IIA Project
GF2: Software
Initial Interim Report

Oscar Saharoy <os408@cam.ac.uk>,

Eric Song <qs222@cam.ac.uk>,

Max Rose <mhr36@cam.ac.uk>

Introduction

In this project we complete the implementation of a logic simulator for both combinatorial and clocked logic circuits. We are working to a specification laid out by a figurative client under a tight deadline, and will therefore need to work efficiently and leverage our experience within the course and outside it.

In this initial report we describe the setup for the wider project, including establishing effective workflows and defining and understanding the syntax for our logic description language. These choices will lay the foundation for the rest of the project; it is vital that we make robust decisions now to ensure the project proceeds smoothly.

General Approach

After understanding the project handout and completing the preliminary tasks, we set up our development infrastructure and assigned tasks up to the completion of the first interim report. We made a rough plan for the overall project but will only make concrete plans for a week at a time, which facilitates quickly pivoting to new directions as we learn more about the task.

We split the work for the initial interim report evenly, taking care to avoid dependencies between tasks and enable parallel work to expedite progress. After this initial sprint is completed, we will regroup and assign new tasks to bring us to the next deliverable, which will probably be the next report.

Coordination & Planning

We have set up a Github repository for this project which acts as our single source of truth - the code is hosted here as well as links to our reports and the tools we are using to coordinate development. We have set up Github Actions for continuous integration (CI) testing with `pytest` and code style compliance with `flake8`.

We decided to write all our reports and documentation within Google Docs which allows us to edit collaboratively and synchronously. For task allocation and tracking we have set up a Trello board which allows us to add and shelve tasks as needed, move them between states of completion and assign tasks to team members.

Allocation of Tasks

For the tasks up to the initial report, we distributed the work as follows although there was teamwork and collaboration on most tasks. We are aiming to complete this initial report by 20/05/22.

- Define and describe our logic description language using Extended Backus-Naur Form (EBNF): *Max*
- Identify and describe possible semantic errors: *Eric*
- Introduction; description of general approach; teamwork and planning - *Oscar*
- Description of error handling: *Eric*
- Construction of two example input files: *Max*
- Construction of circuit diagrams for example input files: *Oscar*

We agreed that after the initial report is complete, work will begin on the implementation phase of the project with Eric handling the GUI module and Oscar and Max handling the scanner, parser, and names modules. We will all be doing system integration as we progress, including creation of unit tests and pathological input files to test against, and also interactive testing of the GUI. When we feel we are finished implementing, we will write tests for and try to find bugs in each other's code to improve coverage. This phase will begin 21/05/22 and we aim to finish our second interim reports by 1/06/22. This will give us a day to rest or fix bugs before we move to the maintenance phase of the project.

The maintenance phase will bring us up to the final report so should be complete 8/06/22. In this phase we will apply all our learnings throughout the project to optimise our development workflow and deliver the features within the week. Our plans for this phase are flexible to allow us to effectively react to the new requirements.

Syntax Described with EBNF

The syntax for the logic description language is defined in Extended Backus-Naur Form as follows.

```
program         = "START ", devices, connections, outputs, "END" ;

device          = deviceName, "=", type, ";" ;

type            = clock | switch | and | nand | or | nor | dtype | xor ;
clock           = "CLOCK", "(", number , ")" ;
switch          = "SWITCH", "(", number , ")"  ;
and             = "AND", "(", number , ")" ;
nand            = "NAND", "(", number , ")" ;
or              = "OR", "(", number , ")" ;
nor             = "NOR", "(", number , ")" ;
dtype           = "DTYPE" ;
xor             = "XOR" ;

signal          = deviceName, [".", pinName] ;
connection      = signal, "->", signal, ";" ;
output          = signal, "--", displayName, ";" ;

devices         = "DEVICES", "{", device, {device}, "}";
connections     = "CONNECTIONS", "{", connection, {connection}, "}" ;
outputs         = "OUTPUTS", "{", output, {output}, "}";

comment         = "#", {any_character}, "#" ;
```

The correct syntax for a program consists of a list of device definitions, followed by a list of connections, and a list of output assignments, in this order. Comments can be inserted at any point in the code, surrounded by a '#' on each side.

Semantic constraints

```
program         = "START ", devices, connections, outputs, "END" ;

device          = deviceName, "=", type, ";" ;


type            = clock | switch | and | nand | or | nor | dtype | xor ;

clock           = "CLOCK", "(", number , ")" ;
```

- The number must be a integer greater than 0

```
switch          = "SWITCH", "(", number , ")"  ;
```

- The number must be a either 0 or 1

```
and             = "AND", "(", number , ")" ;
```

- The number must be a integer from 2 - 16

```
nand            = "NAND", "(", number , ")" ;
```

- The number must be a integer from 2 - 16

```
or              = "OR", "(", number , ")" ;
```

- The number must be a integer from 2 - 16

```
nor             = "NOR", "(", number , ")" ;
```

- The number must be a integer from 2 - 16

```
dtype           = "DTYPE" ;

xor             = "XOR" ;


signal          = deviceName, [".", pinName] ;
```

- The deviceName suggested must be a device claimed in the devices section.
- The pinName must be a valid pin of that device
- If the device that corresponds to the deviceName has more than one output (AND, NAND, OR, NOR, DTYPE), there must be a pinName attribute

```
connection      = signal, "->", signal, ";" ;
```

- The signal name to the left of the "->" symbol must be the name of a device output and the signal name to the right must be a device input.

```
output          = signal, "--", displayName, ";" ;
```

- The signal must be the name of a device output, the displayName must be a string.

```
devices         = "DEVICES", "{", device, {device}, "}";

connections     = "CONNECTIONS", "{", connection, {connection}, "}" ;

outputs         = "OUTPUTS", "{", output, {output}, "}";


comment         = "#", {any_character}, "#" ;
```

Error Reporting

When parsing the file, we'll go through the file first time to detect the syntax errors and then the second time for semantic errors. However, if syntax error is detected, we'll not move forward to semantic error detection.

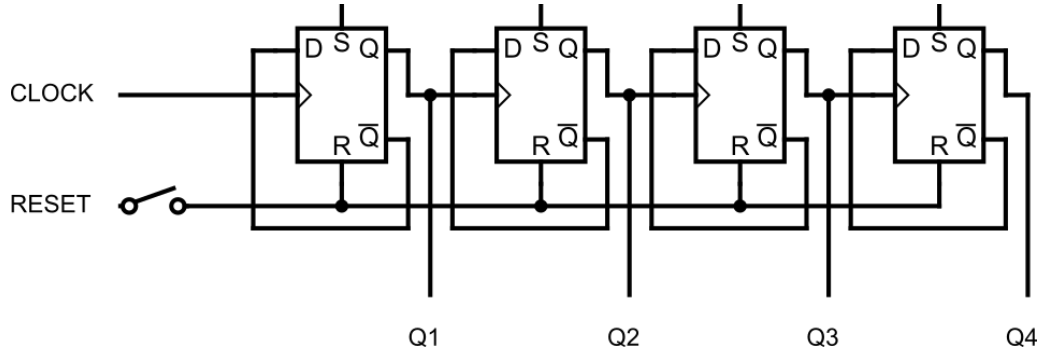
The way we're going to show each error is firstly, print out the script followed by last semicolon/curly bracket to next semicolon/curly bracket. Then print the error pointer on the next line to show the exact location of the error, and then on the third line print out the error type, message, and the suggested way of correcting the error. If there are multiple errors, we put them in a list and print them all together when we finished parsing.

When an error is detected, if it's a syntax error, we will try to continue the parsing from the point after next semicolon or curly bracket (both opening and closing). If the syntax error is about missing semicolon or closing curly brackets, we start directly from the point after the place where we expect a semicolon or closing curly brackets. By doing such we can get as many syntax errors detected as possible before exiting the program.

When we're detecting semantic errors, simply put them in the error list and print them out when finished parsing.

Example Circuits Using the Syntax

Counter Circuit



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);
}
```

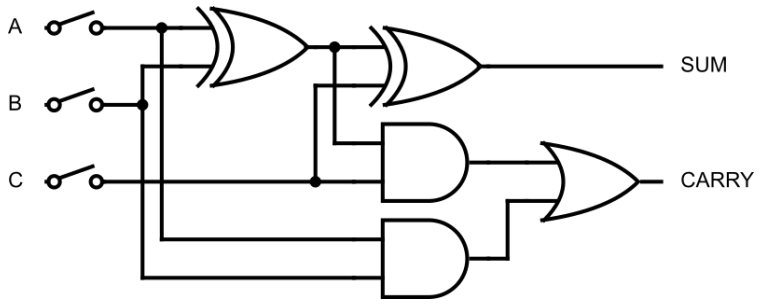
```
CONNECTIONS {
  ff0.QBAR -> ff0.DATA;
  ff1.QBAR -> ff1.DATA;
  ff2.QBAR -> ff2.DATA;
  ff3.QBAR -> ff3.DATA;

  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;
}

OUTPUTS {
  ff0.Q -- "Q0";
  ff1.Q -- "Q1";
  ff2.Q -- "Q2";
  ff3.Q -- "Q3";
}
END
```

Adder Circuit



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 {
  xor1 -- "SUM";
  or1 -- "CARRY";
}
END
```