Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2021

Project Title:	**VIDE: A Verilog IDE for Novices**
Student:	**Olly L. Larkin**
CID:	**01239375**
Course:	**EIE4**
Project Supervisor:	**Dr T.J.W. Clarke**
Second Marker:	**Dr C. Papavassiliou**

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

Abstract

Verilog is a hardware description language that is typically taught to students studying electronic engineering at a university level. While there are many advanced industry tools, the tools aimed at education are limited. This report details the background, design, implementation, and evaluation of a Verilog IDE with the goal of providing a simple environment in which students can explore the language and build designs. While the end product has some limitations, it succeeds in setting the groundwork for this and heavily simplifies the tasks students typically have to deal with using industry standard software.

Acknowledgements

I would like to give special thanks to my project supervisor, Dr Thomas Clarke, who offered invaluable support throughout the project, as well as the EE department at Imperial College for being so understanding during difficult times. I would also like to give thanks to my friends and family for their continuous support.

Contents

Chapter 1

Introduction

Verilog [14] is a hardware-description-language (HDL) commonly used in industry. HDLs are computer languages used to describe digital logic circuits and, typically, have very different syntax to most programming languages. Typically, in industry, either Verilog or VHDL is used but VHDL is generally considered to be more verbose and difficult to learn, making it less suited to education in digital circuit design. Despite this, there is a lack of tools for Verilog that are simple enough to be considered well suited to education.

The tool that students are currently encouraged to use at Imperial College London on the Electronic and Electrical Engineering (EEE) and Electronic and Information Engineering (EIE) courses is Intel's Quartus Prime [23]. There are three tiers of Quartus Prime (Lite, Standard, and Pro) but all of them are designed to be fully fledged, industry appropriate electronic design automation (EDA) tools, meaning that the feature set goes well beyond what a student needs to learn the Verilog language. This is easily illustrated by searching for "Quartus Prime" on Intel's user guide section of the documentation [25], which shows 46 results. Having so many features can be problematic for people approaching the topic for the first time as it can be difficult to find the specific features that they need. For example, creating a new project in Quartus Prime will prompt you to choose which board or family of devices you are targeting, allowing for features such as timing analysis, which in most cases will not be relevant to a student learning the language for the first time.

In a survey conducted by Marco Selvatici for his final year project, he reported 88% of students tend to find Quartus Prime's menu systems confusing and 63% of students struggle to understand the error messages [26].

Quartus Prime also suffers from platform limitations. It is currently only available on specific versions of Windows and Linux [24]. A study conducted by Vanson Bourne on behalf of Jamf reported that, in a sample of 2244 higher education students, 40% use a device running macOS and 71% would prefer a device running macOS if upfront cost was not an obstacle [17]. While this study was not localised to EEE or EIE students, it is reasonable to assume that a fairly large number of students studying these topics are using macOS, meaning to access Quartus Prime they can choose from

the following options: Use a device provided by the university, install a compatible version of Windows or Linux on their existing device (this is only officially supported if the device has an Intel processor [19], which all new macOS devices are moving away from as Apple is now manufacturing its own processors), or use the program through an SSH connection with the university servers, which can become slow and unresponsive when a lot of people need to access it at the same time, which is generally the case as most students would need to use it during scheduled lab sessions.

VIDE hopes to solve these problems by providing a lightweight, intuitive, cross-platform tool to compile and simulate Verilog, with a simple installation. Reducing the feature sets of industry grade applications down to just editing and simulating Verilog means that students don't have to navigate around tools they do not need or want to use, reducing the mental burden and helping them learn the language and develop designs more easily, with minimal effort and guidance.

This report will provide all the information needed to understand the design and development process of VIDE. The individual chapter details are as follows:

- Chapter 2 will walk through all the relevant background material for this project.
- Chapter 3 will detail the necessary requirements and desirable features for the final produce.
- Chapter 4 will use the background material and requirement capture to construct a high level project design.
- Chapter 5 will discuss the construction of the previously specified design.
- Chapter 6 will outline the testing plan and how functional correctness is ensured.
- Chapter 7 will evaluate the final product.
- Chapter 8 will wrap up the report and provide details on what further work could be completed given more time.
- Chapter 9 will be a user guide, giving detailed explanations of how to install and use the final product.

Chapter 2

Background

This section will go through background material that is relevant to this project. Section 2.1 will give some more details regarding the Verilog language that wasn't covered in chapter 1, while section 2.2 will discuss a few existing products in the same or similar category as this project. Section 2.3 will briefly discuss another product recently designed and built at Imperial College London and how it may relate to this project.

2.1 Verilog

As stated in chapter 1, Verilog is an HDL. This means its syntax differs quite a bit from programming languages that students may already be familiar with when they first encounter Verilog. Multiple versions of the standard exist, but this project will only reference IEEE Std 1364-2005 [14] specifically. This standard was released in 2006, after which, Verilog was incorporated into the superset language SystemVerilog, which is outside the scope of this project. There have not been any major changes to the Verilog language since the 1364-2005 specification was released.

Verilog has a large grammar, illustrated by the fact that the standard is 560 pages, but many language features are unnecessary for this tool, as its purpose is to provide an entry point for students, so they may learn the basics of the language before exploring more elaborate tools. The choice of which language features to support will be explored in section 4.1, where explanations of the different language features will also be given.

As well as language features, the standard also describes the expected behaviour of a Verilog simulator. As Verilog describes digital logic, it is essential that simulation incorporates some form of parallelism. The standard defines this in the form of a set of event queues, where items from the queue can be taken and executed in a non-deterministic order. The two event queues relevant to VIDE's reduced language scope are: The active event queue, and the non-blocking assignment queue. An event is defined to be either an update event, which is when a wire or reg value is updated, or a processing event, which is when a process should be executed, such as the initial block or an always block.

The active event queue contains all the events that should be executed at the current simulation time, and the non-blocking event queue holds all of the update events caused by a non-blocking assignment. Update and processing events are taken from the active event queue in a non-deterministic order, as in hardware these events would be happening in parallel. A processing event should put all the update events it generated in the relevant queues once it has finished. Once the active event queue has been depleted, the non-blocking event queue should be executed in a strictly deterministic sequential order. This means that the order that the user uses for non-blocking assignments will be adhered to by the simulator.

The execution of the update events will trigger always blocks (processing events) if the capture group for the always block contains the updating wire or reg.

The simulator behaviour will be discussed more thoroughly in section 5.2.3.

2.2 Competing Products

There are many existing HDL editors and simulators: The Wikipedia page for HDL simulators lists 23 proprietary simulators and 13 free or open-source simulators, of which 29 support some version Verilog or SystemVerilog [18]. To better understand the existing tools and see if they have desirable features for this project, some of the more notable IDEs will be discussed.

Quartus Prime [23]: Quartus has already been briefly discussed in chapter 1, where an overview of the problems with its platform limitations and verbosity were given. Here, I will outline some of its features and capabilities.

As previously stated, Quartus comes in three different editions: Lite, Standard, and Pro. Each have slightly different feature sets and different prices. The most noticeable difference between the three editions is the FPGAs that they support, with the higher tiers supporting higher-end FPGAs. The Lite edition is free and offers all the features a student is likely to need. The Standard edition has a fixed cost of $2,995, and the Pro edition has a fixed cost of $3,995.

Quartus supports more languages than just Verilog: all three editions also support VHDL and (limited) SystemVerilog, as well as having a high level synthesis (HLS) compiler to allow the use of C++ in digital circuit designs. The Pro edition is the only one that also supports VHDL-2008.

Even the Lite edition of Quartus, which has the least amount of features, has a wide array of inbuilt features and tools. Examples of this include: The chip planner (visual display of chip resources showing placement, connections, routing paths, and more), the platform designer (a graphical system representation to help integrate components into designs — the user interface for this feature is shown in fig. 2.1), and the timing analyser (used to validate the timing of the circuit given the clock speed and device it is targeting).

All three editions have access to design simulation via the ModelSim-Intel FPGA

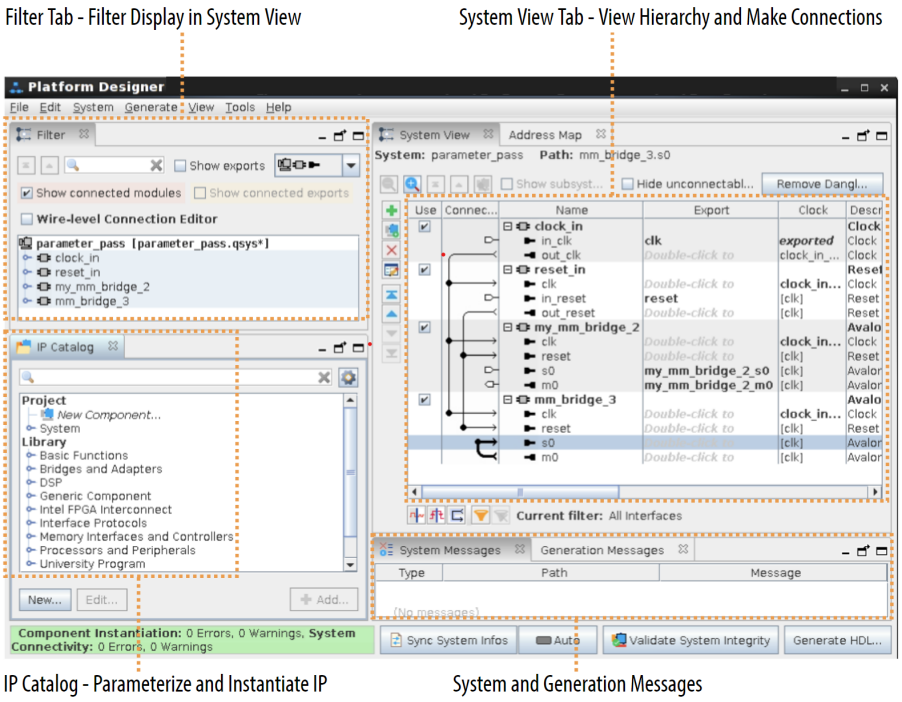

Figure 2.1: User interface for the Quartus Prime feature "platform designer" [15].

Starter Edition or ModelSim-Intel FPGA Edition (requires an additional licence) software. Both are versions of ModelSim [20] targeting Intel FPGA devices, which is an industry suitable digital circuit simulator.

Aldec Active-HDL [1]: Active-HDL is a comprehensive HDL editor, simulator, and verification tool, supporting Verilog, SystemVerilog, and VHDL. It is built for Windows and does not have a macOS or Linux version.

It has a large feature set, boasting "200+ EDA and FPGA tools" [1], operating during design entry, simulation, and synthesis. This is, in part, achieved by interfacing with other vendor tools. A notable feature is that it has multiple options for graphical design entry: block diagram editor, and state machine editor. It also incorporates project management tools with the goal of making an environment capable of handling all the needs of an industry HDL developer team.

The text editor hosts features such as: Cross probing between waveform and editor, templates, keyword highlighting, auto-complete, auto-format, and support for code groups.

A price is not listed on the website, but a link to contact sales is given, which suggests that this product is not expected to be used by individuals; rather by teams working in industry.

Sigasi [27]: Sigasi separates itself from the other tools in this list by the fact that it can be installed as a standalone app or as a plugin to the Eclipse editor, meaning users are able to make use of other Eclipse plugins to change the editor behaviour or look. As it is partially tied to the Eclipse editor, which is cross-platform, Sigasi is also cross-platform and able to run on Windows, macOS, and Linux.

The UI provides multiple views to give different ways to access the information in the project, facilitating navigation, inspection, and editing. It can be configured to fit the users needs, but the default layout has five views: Project explorer, editor, problems, outline, and hierarchy.

One notable issue is that, while it provides linting out of the box, for simulation, the user is expected to integrate with an external simulator. This step may prove difficult for students who are unfamiliar with the tooling surrounding HDLs, although they do offer guides to ease this process.

The product licence is subscription based, at £895 a year for Sigasi Studio XL, and £1,620 a year for Sigasi Studio XPRT. The major difference between the two versions is that Sigasi Studio XPRT is able to offer block diagram and state machine diagram graphical views of designs. Students using the tool for education and individuals working on open source projects are eligible for a free licence, as long as they allow the program to report usage data back to Sigasi.

Looking at some existing products in HDL editing outlines the issues that students currently face while trying to use the tools. Each of the programs above have an excessive amount of features for what a student would require, making the UIs feel cluttered, as evidenced by Sigasi's default layout having five individual views and the "platform designer" feature from Quartus Prime shown in fig. 2.1. While these layouts are extremely useful for people familiar with the tools, allowing them to easily gather information on the system and navigate the project, for students learning the language, it is unnecessary clutter that distracts from the task at hand.

Another potential point of contention for students is the designs of these applications. Most of them are not recently built and have opted for a simple but fairly outdated styling (also evidenced by fig. 2.1). This may help to reduce confusion, as the applications are not simple and excessive styling may make them harder to navigate, but most applications from the last few years are beginning to adopt the styling standard that is common in web applications. As students continue to grow more familiar with this simple modern styling, they will grow less familiar with the styling currently used in these applications, making them feel outdated. This could potentially be solved with the option to theme the programs but most do not offer this.

An HDL editor that doesn't suffer from an overbearing number of features is EDA Playground [6]. It is a web application with a fairly simple design and layout. There is a panel on the left giving all the different editor and simulation options, and then (by default) two editor windows open, and a log below. The options may still prove confusing to students who are not familiar with the tool, but there are not many and it should be a small learning curve to become familiar with them. One issue is that

to view a waveform after simulation, rather than just the log output, the user has to add a `$dumpfile("dump.vcd")` or `$dumpvars` command in their design, we may be unexpected and confusing to new users. Additionally, as it is a web application, it has no direct access to the users local file system. This means to load a project from local storage, the user has to upload each file. The "Playground" in the name indicates that this tool is designed to test language components and get a feel for them, rather than building designs.

2.3 Issie

Issie (Interactive Schematic Simulator and Integrated Editor) is a design and simulation tool for drag and drop style block schematic editing [16]. It is actively maintained by Dr. Thomas Clarke at Imperial College London, with the intent to use it to educating students on digital circuit design. While this product does not serve the same purpose as this project, and therefore cannot be considered a competing product, its goal of educating students about digital circuit design is the same.

Issie is built in an electron framework, making it cross-platform and available on Windows, macOS, and Linux, meaning it also follows modern web application styling. The UI is simple and well laid out making it easy to navigate and use for students.

The reason that Issie is of interest to this project is the potential to integrate the two products. Issie provides a block diagram editor, simulator, waveform viewer, and even Verilog file exporting, but it doesn't provide the option to import a module from a Verilog file. The work done in this project to compile and simulate Verilog files could potentially be used to allow editing blocks with Verilog in Issie. How this effects the design decisions in this project will be discussed in section 4.2 and the possible ways to integrate will be discussed in chapter 8.

Chapter 3

Requirement Capture

The aim of this project, as stated in chapter 1, is to provide an environment in which students are able to comfortably learn the Verilog language before going on to use more advanced tools to synthesise their designs and test them on FPGAs. This section will outline the required features to achieve this goal, and then list further features that may improve the user experience.

The project can be broken down into multiple sections, each with different requirements. The distinct sections of the project are: The code editor, the compiler, the simulator, the waveform view, and the user interface (UI). The following sections will outline their respective required and desirable features.

3.1 Overall Project

To make sure that the majority of students are able to use this product, Windows, macOS, and Linux must be supported.

3.2 Editor

3.2.1 Required Features

The features required for the editor are:

- **Must be responsive enough that the text does not feel like it is lagging behind the keypress:** As this project aims to provide a learning environment, it is important that the user is not distracted by behaviour they do not expect. This may be beyond the control of the project if the user's system is too slow, but should be avoidable for the average system by having a responsive editor.
- **Error checking and reporting on file save:** To provide a fluid editing experience to the user, they should not have to perform a deliberate action to view error checking results, such as clicking a "check for errors" button. Running error

checking on save means that the user is provided with error checking when performing a fairly frequent action. This is the minimum frequency at which error checking and reporting should occur. The majority of editors offer error checking as the user types.

- **Light and dark theme:** An article which collected numerous sources to study the effects of light or dark themes in editors came to the conclusion that for people with normal vision, light themes are more productive, but potentially lead to long term eye health issues [3]. It is also common for users to have a preference for dark themes. To make this product appeal to a wide user base, users should be offered the choice of dark or light theme.
- **Code snippets or auto-completion for common language constructs:** Snippets and auto-complete help speed up development, but they can also provide the user with a template of the construct they are trying to use. This can speed up the learning of new grammar for students who are not well acquainted with the language.
- **Syntax highlighting:** Syntax highlighting provides insight into the underlying syntax of the language that is essential for a student learning it for the first time.

3.2.2 Desirable Features

The desirable features for the editor are:

- **Intelligent code navigation:** For example, this could mean navigating to definitions or submodule files when the user CTRL + clicks the corresponding identifier in the code. This is a nice benefit to have and can greatly speed up development as it means the user does not need to spend time trying to find the definitions manually.
- **Info on hover:** When hovering over specific components, the editor could give details about how that component functions or its type. This would be very useful for students first learning the language.
- **Error checking and reporting as the user types:** Type-time error reporting means that the user can see errors they may be making as they go and would be a great benefit.
- **Custom themes:** Allowing user to install custom or 3rd party themes means that they can get the editor looking how they want, making them more likely to use the product.

3.3 Compiler

The compiler must be able to correctly compile the entire Verilog grammar supported by this project down to a netlist type. The speed at which it does this will also determine how feasible it is to have type-time error reporting in the editor, as well as how responsive editing feels.

3.4 Simulator

The main requirement for the simulator is that it is able to simulate any netlist that the compiler may produce while adhering to the behavioural requirements laid out by the Verilog standard [14] as discussed in section 2.1.

The speed of the simulator will also play a large role in how responsive the overall application feels. It is essential that it is able to simulate hundreds of cycles of a simple circuit, such as an 8-bit adder, within a few seconds.

3.5 Waveform Viewer

3.5.1 Required Features

The required waveform viewer features are listed below:

- **Must display correct waveform given simulator output:** Any discrepancy between the simulator output and the waveform view would be misinforming the user, leading them to believe that their designs behave differently to how they may expect.
- **Show the value of a bus:** It is common in electronics projects to want to see the value of the bus if it is being treated as a single number.
- **Break down busses into individual bits:** It is common in electronics projects to not just want to see the bus as a single value, but as a collection of individual wires.
- **Clearly show unknown bits:** Verilog is capable of representing bits as the unknown value `X`. The waveform must be able to reflect this.
- **Show bus values in binary, octal, decimal, and hexadecimal bases:** It is common in electronics projects to need to see output one of the mentioned bases, so the waveform must be able to display them.
- **Zooming:** Depending on the clock frequency the user is using, the default resolution of the wave might not be adequate. If they are using a large clock frequency, then it is reasonable to suggest that they may need to zoom in to get a better

idea of how the signals are changing. If they use a lower clock frequency, they may need to zoom out.

3.5.2 Desirable Features

The desirable waveform viewer features are listed below:

- **Interactivity between waveform and editor:** If the user clicks on a signal, highlighting the relevant register or wire in the editor would be quite useful in aiding comprehension.
- **Save as PNG:** The option to save the waveform as an image would be quite useful for students as it is likely they would need waveform images for lab books or reports.

3.6 User Interface (UI)

As the only user triggered function this project performs is simulating and creating and managing project files, as error reporting is automatic, it is essential that these operations are as simple as possible. A large issue with the majority of IDEs discussed in section 2.2 is their complexity when trying to perform simple actions. This needs to be avoided in this project.

Chapter 4

Analysis and Design

This section will, with inspiration from the products discussed in section 2.2 and the requirements laid out in chapter 3, construct a high level design for this project.

4.1 Verilog Language Scope

An important aspect of this project is the decision of what Verilog language features to support. The Verilog language is large and has many features that students are unlikely to need while first learning the language. Including these features as valid syntax means more work is required to construct the compiler and simulator, but also that the student is more likely to write syntactically correct code that does not do what they expect, without receiving a warning or error. However, care must be taken in choosing the language features so that the students do not find it too limiting.

A key resource is choosing which features should be included was Professor Peter Y.K. Cheung's lecture notes on Circuits and Systems (CAS), for second year EEE and EIE students at Imperial College London [5]. This is the first introduction to Verilog for EEE and EIE students and is the only point during university where it is explicitly taught to them. In following modules, such as the Digital System Design (DSD) course for third and fourth year EEE and EIE students [2], knowledge of the Verilog language is assumed and students are expected to be able to apply it to the concepts being taught. This is illustrated further by the fact that the coursework for DSD does not have Verilog as a strict requirement, but also allows students to submit solutions written in VHDL. It is important that this project supports the majority of features referenced in the lecture notes, as these are features the students are expected to know and use.

Cheung's CAS notes introduce Verilog in lecture 6. They then go in to introduce new language features throughout. The first example of Verilog given depicts a 2-to-1 multiplexer and is shown in listing 4.1.

Listing 4.1: The first example of Verilog code given in Peter Y.K. Cheung's Cicuits

and Systems module in lecture 6 [4].

```
1  module mux2to1 (out, outbar,
2                  a, b, sel);
3
4      output  out, outbar;
5      input   a, b, sel;
6
7      assign out = sel ? a : b;
8      assign outbar = ~out;
9
10 endmodule
```

Listing 4.2: Alternative syntax for specifying port direction.

```
1  module mux2to1 (
2      output out, outbar,
3      input a, b, sel);
4
5      assign out = sel ? a : b;
6      assign outbar = ~out;
7
8  endmodule
```

Listing 4.1 introduces some basic language features. First is the **module** keyword on line 1, which, when paired with the **endmodule** keyword on line 10, defines a new module. After **module**, the identifier **mux2to1** is given. This is the name of the module being defined. After this, a bracketed list of identifiers are given, which are the inputs and outputs to the module. The body of the module starts with line 4 and specifies that the previously declared ports **out** and **outbar** are outputs. Similarly, line 5 specifies that the ports **a**, **b**, and **sel** are inputs. Verilog actually supports multiple ways of specifying port direction, with an alternative shown in listing 4.2. Both methods produce the same results and will be supported by this project. Continuing on to line 7 in listing 4.1, we are introduced to the **assign** keyword. This construct has the syntax **assign [wire] = [expression]**, and creates a continuous assignment, meaning the value of the wire is permanently tied to the expression. Line 7 continuously assigns the ternary expression **sel ? a : b** to the wire **out**, while line 8 continuously assigns the expression **out** to the wire **outbar**. All language features shown here will be required for this project, as well as all expression operators.

The next bit of Verilog syntax introduced is shown in listing 4.3.

Listing 4.3: Introducing internal wires and gate level modules [4]

```
1  module mux_gate (out, outbar,
2                   a, b, sel);
```

```
3
4       output  out, outbar;
5       input   a, b, sel;
6       wire    out1, out2, selb;
7
8       not i1 (selb, sel);
9       and a1 (out1, a, sel);
10      and a2 (out2, b, selb);
11      or o1 (out, out1, out2);
12      not i2 (outbar, out);
13
14  endmodule
```

Listing 4.3 introduces a few new features. First is the keyword **wire** used to declare the wires **out1**, **out2**, and **selb**. This creates instances of the wires, which are now able to be used as inputs/outputs to other modules, part of expressions, and can be continuously assigned to with the **assign** keyword. Lines 8-12 are all examples of module instantiations. This is when another Verilog module is used as part of your design. In the case of line 8, an instance of the module **not** is being created. It is given the instance name **i1**, and then given **selb** as its first port (which happens to coincide with the output), and **sel** as its second port (which coincides with the input). Wire declarations and module instantiations are both essential aspects of the Verilog language and therefore must be supported by this project. Incidentally, the modules used in lines 8-12 (**not**, **and**, and **or**) are inbuilt modules in the Verilog language representing different gates. These modules will not be made available by this project, as they can easily be recreated by the user, their functionality can be more conveniently accessed through use the of inbuilt operators, and they are not commonly used in practice.

The next set of introduced language features is shown in listing 4.4.

Listing 4.4: Introducing always blocks [4]

```
1   module mux_gate (out, outbar, a, b, sel);
2       output  out, outbar;
3       input   a, b, sel;
4
5       reg out, outbar;
6
7       always @ (a or b or sel) begin
8           if (sel) out = a;
9           else out = b;
10
11          outbar = ~out;
12      end
```

```
13
14 endmodule
```

Listing 4.4 shows a functionally similar implementation as listing 4.1 and listing 4.3, but here uses an **always** block to handle the computation. An **always** block in Verilog is a block of code with an event trigger that contains more C-like statements which run procedurally, such as **if**, **case**, and loops. In this case, the code block on lines 8-11 will run when any change occurs in the wires **a**, **b**, or **sel**. In this case, the trigger is the equivalent of using the syntax **always @(*)**, as this will case any wire or reg evaluated in the code body to become part of the event trigger list, and the only wires or regs evaluated in lines 8-11 are **a**, **b**, and **sel**. This example also introduces the **reg** keyword on line 5. This specifies that the ports **out** and **outbar** are not wires (which is the default), but regs, which is necessary for them to be assigned to during procedural execution in the **always** block. These features are again essential to the majority of Verilog programs and are therefore supported by this project. The decision was made to not include looping statements, however, as this tool is aimed at people likely coming from a software programming background and changing to think about Verilog in terms of programming hardware can be difficult. Therefore, moving away from constructs not commonly used in Verilog but very commonly used in software, such as loop statements, may make this transition easier.

It is also worth noting that there is alternative syntax to that used in line 5 of listing 4.4. While the user is able to separately specify that a port is a reg, they are also able to do it inline with the port declaration, such as **output reg out, outbar**. This syntax will also be supported by this project.

The keywords **posedge** and **negedge** can also be used in **always** block event triggers to indicate that it should only trigger if the lowest bit goes from a 0 to a 1 or a 1 to a 0 respectively. This syntax is demonstrated in listing 4.5.

Listing 4.5: Posedge and negedge demonstration

```
1 // previous code ...
2
3     always @(posedge a, negedge b) begin
4
5 // more code ...
```

Regs and wires in Verilog can also be indexed. So far the examples have all shown examples of single bit wires and regs, but Verilog is capable of having busses which can hold multiple bits. An example of this syntax is shown in listing 4.6.

Listing 4.6: Indexing in Verilog

```
1 module indexing_example(
2     input [7:0] a,
3     output [2:0] b);
```

```
4
5       assign b[0] = a[7];
6       assign b[2:1] = a[4:3];
7   endmodule
```

Indexing in Verilog is inclusive, meaning that the index **[7:0]** results in an 8-bit bus. It also requires the larger number as the first element and the smaller as the second. Listing 4.6 shows indexed declarations on lines 2 and 3, meaning the input and output wires **a** and **b** are busses with 8-bit and 3-bit widths specifically. The body of this module simply assigns bit 7 (0 indexed) of **a** to bit 0 of **b**, and bits 3 and 4 of **a** to bits 1 and 2 of **b** respectively. Indexing is another essential feature that must be supported by this project.

Another language feature not yet introduced is non-blocking assignments in procedural blocks such as an **always**. An example is shown in listing 4.7. The difference in functionality between blocking (previously seen) and non-blocking is that the assignment when using a non-blocking assignment is put onto a delayed event queue, meaning it is not executed until after the block has run. This means to swap two values, the example in listing 4.7 can be used, as the value of the first is not overwritten until all the statements are executed.

Listing 4.7: Blocking vs non-blocking assignments

```
1   // some code ...
2
3       // blocking assignments -> this will result in
4       // both a and b having the original value of b.
5       always @(*) begin
6           a = b;
7           b = a;
8       end
9
10      // non-blocking assignments -> this will result
11      // in c and d swapping values
12      always @(*) begin
13          c <= d;
14          d <= c;
15      end
16
17  // some code ...
```

Another key language feature not yet discussed is **initial** blocks. The syntax for this is given in listing 4.8. In Verilog, the initial block can contain the same logic as an **always** block and is effectively a very similar construct. The main difference is that

while an **always** block triggers when one of the event triggers in its sensitivity list fires, an **inital** block triggers only once at the start of simulation, hence the name "initial". Typically, it is used to initialise register values, but can also be used to construct a test bench. Another language feature not introduced yet are delays, with the syntax **#<number>**, e.g., **#10**. These can only be used inside a procedural block and cause a delay of the specified amount of simulation cycles. A test bench can be constructed using the **initial** construct and delays to change the inputs to the module being tested. The decision was made to not support this functionality. This is because the aim of this tool is to introduce Verilog in the context of digital circuit design. Allowing students to also verify their designs using the Verilog language shifts this focus. As an alternative, an interface will be available to users to specify inputs to the module before simulation, meaning the Verilog language is strictly used for design purposes. **Initial** constructs are still supported but limited: The only type of statement allowed within an **initial** construct is a blocking assignment, as shown in listing 4.8.

Listing 4.8: Initial construct

```
1  // some code ...
2
3      reg a;
4      reg [7:0] b;
5
6      initial begin
7          a = 1;
8          b = 8'b11011110;
9      end
10
11 // some code ...
```

For a full specification of the supported grammar in Backus-Noir form (BNF), please refer to appendix A.

4.2 Framework

There are many options for which framework to use when building this project. Firstly, it could be a standalone application, or it could be an extension or plugin for an existing application, like Sigasi, as mentioned in section 2.2. If standalone, we then have the option of a native desktop application of a hosted web-app.

This section will explore the possibilities and ultimately provide a high level overview of the project framework.

One key aspect in making these decisions was the potential to integrate the final product with Issie. As discussed in section 2.3, Issie is an existing drag and drop style circuit designer and simulator being actively maintained and used at Imperial College

London. While integration with Issie was outside the scope of this project, for it to be possible as future work, there needs to be parallels in the languages or frameworks used to make sure it is compatible.

The majority of Issie is written in F#, which is a functional language running under .NET [10]. It is running in an Electron [7], which is a framework for running JavaScript web-apps natively on desktop. Issie transpiles F# to JavaScript using Fable [11], which is an F# to JavaScript transpiler, to allow this. To build the UI in F#, Issie uses the Elmish F# library [8], which uses the model-view-update (MVU) architecture [28] to render graphical components.

Ideally, this project would match the framework of Issie exactly, meaning integration would be fairly straightforward, but this introduces some technical difficulties which can be difficult to overcome.

The biggest issue with using Issie's framework for this project is F# library compatibility. As all the F# code is transpiled to JavaScript using Fable, only libraries compatible with Fable will work. This means that the F# code cannot make use of libraries written in C#, which is fairly common as it is the more popular .NET language. The place where this presents the biggest problem is in parsing the Verilog language. As the compiling has a constraint on performance, as mentioned in section 3.3, using an existing parsing library to parse the Verilog code is desirable, as it would likely be heavily optimised in comparison to a custom build parser. Ultimately, finding a well documented parser library compatible with fable proved difficult and, due to the time constraints of the project, I made the decision to move ahead and not use the F#/Fable/Electron framework.

While this means that integrating with Issie would not be that straight forward, if the project still used F#, it would still be possible by re-writing the non-fable compatible sections. There are also benefits to using a functional language such as F# beyond Issie integration. The lack of mutable variables and constructs typically used when working in a functional language like F#, compared to a typical object-oriented language, mean that bugs due to program state are significantly less common. These kinds of bugs can be difficult to debug, as state in a program can be hard to properly determine with poorly designed code. Avoiding this is a large benefit. F# also has the benefit of being able to easy interop with C# and C# libraries, of which there are many, so the issues normally plaguing smaller community languages are not present. Due to these reasons, I chose F# to program all the performance focused core logic of the IDE, namely compilation and simulation.

Using F# for the core logic of the project (backend) does not limit the options for building the UI (frontend), as the backend can be called from most frameworks. This would require some work to create an inter-process communication method but is feasible. After dropping fable as a framework requirement, there is not much reason to continue using Electron, as the UI components most likely would need redesigning anyway if being integrated with Issie and should be fairly simple. With this in mind, an extension or plugin for an existing editor would provide a few benefits.

Visual Studio Code (VSCode) [29] is a lightweight general purpose editor with an

extensive API for building extensions. WakaTime is a management tool to record metrics for programmers and they released data for 2020 showing that VSCode was by far the most popular tool among their users [13]. They reported that their users spent roughly 7 times the time in total in VSCode than they did in the next popular editor, IntelliJ. Other popular editors, such as Visual Studio, had upwards of 16 times less time spent than VSCode. This suggests that a large amount of the user-base for this project are likely already familiar with VSCode. This familiarity would work to the advantage of the project, as an intuitive and simple UI is one of the design goals. Another benefit is the wealth of extensions already available, including editor functionality extensions and themes. This means that users are able to customise the IDE as they see fit, even creating their own extensions if one does not already exist that suits their needs. There are restrictions when building an extension as opposed to building a native application, but these are unlikely to cause problems while developing this project as the UI should be as simple as possible. Due to these reasons, this project will be built as an extension to VSCode.

As previously discussed this will result in the need for a communication channel between the VSCode extension and the backend core logic. The details of this will be discussed in section 5.1. To be able to run the backend, there are multiple choices. The first option is to compile down to the each architecture that needs targetting (Windows, macOS, and Linux in this case) and then call the platform specific application from the extension. The other option is to compile the backend down to a `.dll` file, which can then be run, regardless of platform, using the .NET runtime. The second option is more attractive, as it simplifies the build process and means that the extension is compatible with any system capable of running VSCode and .NET, even if it does introduce a .NET dependency. This will be the approach taken in this project.

Chapter 5

Implementation

This chapter will provide the details of the project implementation, refining the designs laid out in chapter 4. Section 5.1 will discuss the communication method built to join the backend and frontend, section 5.2 will walk through the implementation of the backend core logic, namely the parser, compiler, and simulator, and section 5.3 will detail the implementation of the VSCode extension.

5.1 Inter-Process Communication

With the backend being a separate process to the frontend, a method of communication between the two is required. There are a few options for how this communication occurs, such as stdin and stdout (text based input and output on the command-line), or through the use of sockets, with the backend and frontend communicating over a port using the localhost IP address. Both solutions require a message sending protocol so that the front and backend are able to communicate effectively. This protocol only needs to act one way: The frontend must be able to call backend functions but not vice versa.

In its simplest form, this protocol could be an enumerated message type, with different values corresponding to different backend API functions (the functions made available to the frontend), followed by the arguments required to perform the job. For example, the message type could be a string, and the value "COMPILE" could mean to call a `compile` API function available in the backend. However, this results in extra work for future developers and maintainers. As a tool built for education will inherently require updating, as the content taught in constantly changing, it is reasonable to suggest that future updates to this project may require changes to the backend API. If the above approach is used, any update to the signature of an API function would require the communication interface to also be updated to reflect this change. The same is true for if a new API function was to be added. The developer would need to write the function, then introduce a new message type value to represent that function, as well as specify the arguments it is able to take in some way.

To avoid these potential issues, another approach was taken in this project. Rather than having to add a new message type for each API function, the system was designed so that an attribute could be attached to the API function that would expose it as an available backend function and make it callable by the frontend. An example of the syntax is given in listing 5.1.

Listing 5.1: Demonstration of attribute to expose API functions

```
1 [<ExposeMethod>]
2 let add2Numbers (a: int) (b: int) =
3     a + b
```

In listing 5.1, line 1 shows the attribute previously mentioned. Applying it here to the function **add2Numbers** means that the function is callable through the communication interface, and also that its details will be included in a JSON formatted declaration, which can be exported by the communication interface and gives details of all the exposed functions that can be called.

To call this function from the frontend, the frontend must construct a message in a JSON format and send it across a communication channel. The format of such a message is shown in listing 5.2.

Listing 5.2: Format of a request made by the frontend for the backend

```
1 {
2     "id": number,
3     "methodName": string,
4     "parameters": any array
5 }
```

The "id" field is used to make sure that the response from the jobs can be matched up with the correct requests, the "methodName" field refers to the function that is being called, and the "parameters" field is an array containing the ordered parameters supplied to the function. An example job for the function given in listing 5.1 is shown in listing 5.3.

Listing 5.3: Example job to call function in listing 5.1

```
1 {
2     "id": 0,
3     "methodName": "add2Numbers",
4     "parameters": [
5         1,
6         2
7     ]
8 }
```

If this job was send to the backend, the backend would execute the job and respond with the output of the function call. The response message is of the format shown in listing 5.4.

Listing 5.4: Format of a reply sent to the frontend

```
{
    "id": number,
    "reply": any
}
```

The "id" field in the reply will match the "id" value given in the job. The "reply" field must have an **any** type, as it must be able to handle any function call, and holds the value of the function return. For the example job given in listing 5.3 has a value of "1" for function argument **a**, and "2" for function argument **b**, resulting in the response message shown in listing 5.5.

Listing 5.5: Response given by the backend for the example job in listing 5.3

```
{
    "id": 0,
    "reply": 3
}
```

This system was implemented by constructing the class **MethodDispatcher** in the backend. This class requires two functions upon construction: Firstly, a function to receive messages, and secondly, a function to post replies. Once it is constructed, it searches the assemblies in the project for methods with the **[<ExposeMethod>]** attribute. Once it has found them, it validates that they conform to some requirements. Due to the arguments having to be serialised to a JSON representation and back, generic parameters are not supported. This is because to deserialise the parameters when the job is received, it uses the function signature to know what type it should deserialise to, and if the parameter is generic, then the type to deserialise to would have to be inferred from the serialised object. This was considered an advanced feature that was unnecessary for this implementation. Another requirement is that exposed API functions cannot have the same names. The reason for this is that if multiple exposed functions have the same name, then the function being called when the name is used is ambiguous. One approach to solving this would be to use the full function signature, including the parameter names and types, to identify the function. This would mean that only duplicate function signatures would be invalid, rather than function names, but would ultimately make the jobs much less human-readable. For this reason, I opted to provide the option to expose the function using a different name. The syntax for this is shown in listing 5.6.

Listing 5.6: Optionally specifying a different name to expose a function

```
1 [<ExposeMethod(name="plus")>]
2 let add2Numbers (a: int) (b: int) =
3     a + b
```

Using the declaration in listing 5.6 would mean that the API function **add2Numbers** is only available over the communication channel using the "plus" as the "methodName". Trying to use the old name "add2Numbers" would cause an error. An example job to call this newly declared function is shown in listing 5.7.

Listing 5.7: Example job to call newly declared function in listing 5.6

```
1 {
2     "id": 0,
3     "methodName": "plus",
4     "parameters": [
5         1,
6         2
7     ]
8 }
```

The **MethodDispatcher** class has an internal declaration of exposed functions that holds a map of function names to a reference of the function that can used to execute said function when a job is sent. It also produces an external declaration serialised in a JSON format that can be requested and provides details of the exposed API functions, as mentioned previously, which can be used by the frontend as documentation on which API functions are available to call.

The class has a public method **Start()**, which, when called, enters a loop of processing incoming jobs that is only exited if an exception is encountered during this process. Firstly, it calls the **getJob()** function, which is passed to the class when constructed as previously mentioned, which returns the next job. This job is then processed in an asynchronous process, meaning that the loop is able to immediately continue and wait for the next job. This is beneficial for the case where this system is used with a continuous stream of input jobs. When the function call returns, the response is serialised and added to a thread-safe queue to be dispatched by a single process. This means that both the receiving of jobs and sending of replies is strictly only handled by an individual thread, meaning there is no opportunity for thread related bugs.

5.2 Backend

The backend of this project is considered to be the parser, compiler, and simulator. All is implemented in F# to produce a single console application that can be invoked by the frontend. Each will be discussed in its own respective subsection.

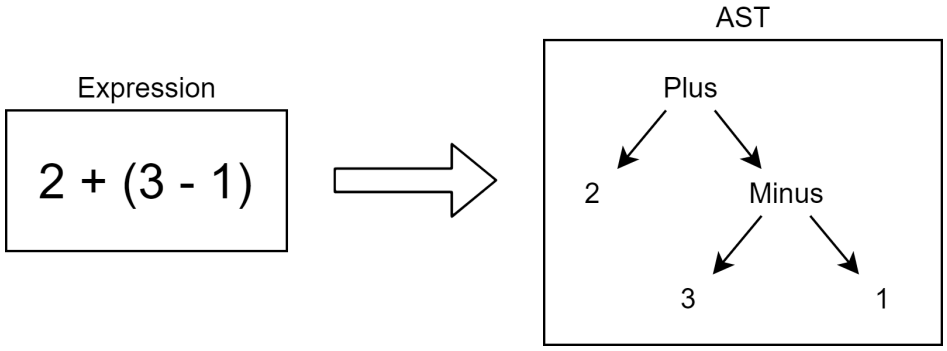

Figure 5.1: An example of an AST for a simple expression

5.2.1 Parser

Parsing, in this project, is defined as taking the source text from a Verilog file and transforming it into an abstract syntax tree (AST). An AST is a representation of the source code and is unaware of any context or language semantics, only its syntax. For this project, the AST will line up very closely with the BNF Verilog grammar given in Appendix A An example of a simple AST representing an expression is shown in fig. 5.1.

To achieve this goal, the library FParsec [12] was used. This is a library built for F# that provides a set of parser combinators, as well as some primitive parsers. A parser combinator is an operator that joins two or more parsers together for different results. The library has many but a few notable ones are:

- **`par1 >>. par2`**: Apply **`par1`** followed by **`par2`** and return the result of **`par2`**.
- **`par1 .>> par2`**: Apply **`par1`** followed by **`par2`** and return the result of **`par1`**.
- **`par1 .>>. par2`**: Apply **`par1`** followed by **`par2`** and return the result of both as a tuple.
- **`choice [par1; par2]`**: Apply **`par1`** and return the result if it succeeds. If it fails, apply **`par2`**.

By default, the operators do not allow backtracking. This is when the parser consumes some input and then fails, that input is not available when trying a different parser. In a lot of scenarios, backtracking is avoidable, and by having the default behaviour not allow backtracking, the developer is more likely to consider an approach that would work without it, which is more efficient. If backtracking is required, it can still be used by adding a question mark to the end of the operators.

The primitive parsers that FParsec provides are built to parse small expressions, such as **`pstring`**, which will try and parse a specific string and returns the string if successful. Using the available combinators and primitives, it is possible to create complex grammars.

Parsing expressions required using a slightly different approach to the other language constructs, as it required the use of left-recursive grammars, due to the left associativity of some operators. Fortunately, FParsec provides the tools for dealing with this. They have a specific class of parser called **OperatorPrecedenceParser** aimed at parsing expression type grammars, which also allows the developer to define the order of precedence of the operators. This greatly simplified the work involved by allowing me to define the parser for an expression primitive, and then add operators with their relevant precedence.

While the parser returned line and column information by default, this information was lost when the parser constructed the AST in the initial implementation. This meant that any semantic errors recognised by the compiler did not have relevant line and column information to display the error in the editor. The way around this problem was to construct a function that took a parser as input and return a parser that would also record line and column information. This is shown in listing 5.8.

Listing 5.8: Demonstrating how line and column information can be recorded using FParsec

```
1  type WithPos<'T> =
2      { file: string
3        start: Position
4        finish: Position
5        value: 'T }
6
7  let withPos
8      (p: Parser<'T,UserState>)
9      : Parser<WithPos<'T>,UserState> =
10     pipe4 getUserState getPosition p getPosition
11     <| fun file sPos value fPos ->
12         { file = file
13           start = sPos
14           finish = fPos
15           value = value }
```

The function **withPos** (starting on line 7) uses the FParsec function **getPosition** to return the position before applying the parser **p**, and then again afterwards. This gives the bounds of the object parsed so that if there is an error during compilation, it is able to report this information. It also records the file name from the user state, as the compiler will flatten all components into a single netlist, as will be discussed in section 5.2.2, meaning it needs this information to determine which file the errors should be displayed in.

One issue encountered when constructing parser for the Verilog subset was dealing with comments. It was fairly straightforward to make the parser accept whitespace or comments after any construct, which did mean that comments were ignored, but this

resulted in undesirable error messages. FParsec produces default error messages in the event of a failing parse, which are very descriptive, saying what language features it would have expected instead of what it received. When allowing for comments after every item, the error messages would then specify that it expected a comment or other language construct. This goes against what a user would typically expect from an error message, as comments are not normally considered a language feature. To remedy this problem, instead of allowing the parsers to absorb comments after every construct, the comments were removed in a preprocessing step. This needed to be done with care, however, as otherwise the line and column information reported in the error would be incorrect if there had been a comment which was removed, as this would change the rest of the file layout. The solution was to replace the comment with whitespace equal in length, and preserve the newlines used in multiline comments, so that the rest of the file remains unchanged.

5.2.2 Compiler

The role of the compiler is to convert the AST, mentioned in section 5.2.2, into a netlist that can be used for simulation. In this project, the netlist is defined as a further representation of the source code that is aware of semantics and is suitable to be simulated. The type for the netlist is given in listing 5.9.

Listing 5.9: Netlist type

```
1 type Netlist =
2     { varMap: VarMap
3       initial: AssignItem list
4       alwaysBlocks: IndexedAlwaysBlocks
5       modInstNames: IdentifierT list }
```

In this, **varMap** holds all the information regarding the inputs, outputs, regs, and wires in the system. If they are continuously assigned, this assignment is recorded in the map. Next, **intial** holds a list of assignments that occur in the **initial** block. Verilog allows any statement within an **initial** block but, as mentioned in section 4.1, this project only supports using blocking assignments, meaning the contents of the **inital** block can be fully represented with a list of said assignments. The **alwaysBlocks** field holds information regarding the **always** blocks in the system. There are indexed, as this is required by the simulator and will be discussed in section 5.2.3. It also holds the event trigger list, as well as the statements in the body of the block. Lastly, **modInstNames** is simply the names of the modules in the system so far, mainly used for validating that the names are not re-used for other components.

The decision was made to have a single netlist for the whole system, rather than individual netlists representing each module used. This ultimately would result in more components in total if the same module is used multiple times in the project, but simplifies some logic when compiling and simulating. To avoid issues with the same

name being used for wires and regs in submodules, which is acceptable if they are not being repeated within the same module, when the netlists are merged, submodule names are all prefixed by the instance name of the module they are a part of, along with the ":" symbol, which is not a valid character to have in a name in Verilog. This means that a name could only be considered a duplicate across modules if the name of the module instance was also the same, which is already caught as an error by the compiler, eliminating this possibility.

The compiler is fairly large and complex, and an issue that became apparent quite quickly was how to effectively deal with errors and warnings throughout compilation, as they could be found at any stage. If an error was encountered, often it would be while iterating over a few independent constructs, and in this case, it is better to continue trying to compile the other constructs and generate multiple errors. If a warning is generated, then compilation should continue with the warning returned at the end with the final result. This system means that, while the parser is only able to recognise a single error (where the parsing failed), the compiler is able to return multiple errors and warnings. This system could also very quickly lead to messy and unreadable code if care was not taken. To effectively handle errors and warnings, the types in listing 5.10 were used.

Listing 5.10: CompResult types

```
1  module CompResult =
2
3      type CompErrors = WithPos<string> list
4
5      type CompWarnings = WithPos<string> list
6
7      type CompRes<'Result> =
8          | Succ of 'Result
9          | Warn of 'Result * CompWarnings
10         | Fail of CompErrors
```

Here, **`WithPos`** is the same type shown in listing 5.8 and is an instance of type string in both cases. This means all errors and warnings hold a file name and position as well as the string error message. This is a generic type, meaning the **`'Result`** type can be whatever type is necessary for the situation. The values **`Succ`**, **`Warn`**, **`Fail`** represent success, warnings, and errors respectively. These types allow the propagation of errors and warnings, but does not mean that the code will be clean and easy to read or write. Just using this type means that every time something that can cause an error or warning is returned, it must be unwrapped before it can be used. To combat this, and make the code neater and more readable, some helper operators were constructed. These are shown in listing 5.11.

Listing 5.11: CompResult helper functions

```
module Utils =

    /// A bind operator for the CompRes type
    let (?>) (r: CompRes<_>) f =
        match r with
        | Fail e -> Fail e
        | Succ v -> f v
        | Warn (v, w) ->
            match f v with
            | Fail e -> Fail e
            | Succ v' -> Warn (v', w)
            | Warn (v', w') -> Warn (v', w @ w')

    /// Similar to bind except 'f' does not
    /// need to return a CompRes, just a value.
    /// Cannot generate errors or warnings
    let (?>>) (r: CompRes<_>) f =
        match r with
        | Fail e -> Fail e
        | Succ v -> Succ (f v)
        | Warn (v, w) -> Warn (f v, w)
```

The bind operator **?>** allows the developer to chain commands together that return **CompRes** types. It works by first checking the output of the last command (**r** in this example) to see if it was an error, success, or warning. If it was an error, then it does not run the second function and returns the error. If it was a success, then it returns the result of applying the function to the value of the successful result. If it was a warning, it now checks the output of applying the function to the previous value and concatenates the warnings if another is generated.

For convenience, the other operator **?>>** was created that allows the function **f** to not return a **CompRes** type, and simply only apply the function **f** if the result **r** was not an error.

While these operators help a great deal with tidying up the code and removing many manual unpackings, they do not provide a solution to the compounding of error messages previously mentioned. The compounding of error messages is only really possible when iterating over a list of separate constructs in the AST, such as a list of statements inside an **always** block. This implies that to allow this, a new **map** function for iterating over lists should be created that allows the collection of errors or warnings. This is shown in listing 5.12.

Listing 5.12: CompResult list map function

```
let rec compResMap mapper list =
    match list with
```

```
3       | [] -> Succ []
4       | head::tail ->
5           match mapper head with
6           | Fail e ->
7               match compResMap mapper tail with
8               | Fail e' -> Fail (e @ e')
9               | _ -> Fail e
10          | _ as r ->
11              r ?> fun pHead ->
12                  compResMap mapper tail
13                  ?>> fun pTail -> pHead::pTail
```

The logic of this function is fairly complex, so I will explain it at a high level. **CompResMap** takes a **mapper** function as well as a list that it is mapping. It is implicitly typed so that the mapper function returns a **CompRes** type. If the list is empty, then return a success with an empty list. If the list is not empty, then deconstruct the first element. Next, if applying the **mapper** function to this element results in an error, then continue with the result of list and concatenate with any other errors generated. If it did not result in an error, then continue with the rest of the list. In the case of a warning being generated, the warning messages will be automatically collected due to the use of the **?>** and **?>>** operators demonstrated in listing 5.11.

5.2.3 Simulator

Simulation is defined in this project as the act of taking the compiled netlist for the system and providing inputs for a certain number of simulation cycles, where a simulation cycle is when the system has settled after input for that cycle is applied. The output of this process is the values of specified inputs, outputs, regs, or wires at the specified simulation cycles.

The Verilog standard [14] outlines a simulation reference model which should be followed by any Verilog simulator, which was briefly outlined in section 2.1. The reference model uses an event system, where events can be either update or evaluation events. An update event is a change in value of a wire or reg. An evaluation event is when a process (always block or initial block) should run. When an update event triggers, the sensitive processes (always blocks with updated wires or regs in its trigger list) are scheduled as evaluation events. Any updates caused by the process when it runs should be scheduled as update events.

There are multiple event queues, but not all are relevant due to the reduced language grammar, so only the relevant ones will be discussed. These queues are: Active events, inactive events, and non-blocking assign update events. Different types of assignments result in adding events to different queues, as will be discussed below.

Continuous assignments: A continuous assignment uses the `assign` keyword. The left-hand side of the equality sign must be a wire, and the right-hand side must be an expression. For the sake of simulation, this is considered a process which is sensitive to the variables used in the expression, meaning an event to update the left-hand side value is scheduled whenever one of the sensitive variables changes.

Blocking assignment: A blocking assignment (BA) is an assignment used in an initial or always block that has a reg on the left-hand side and an expression on the right-hand side with the `=` operator between. When a BA is encountered, the right-hand sign is evaluated and assigned to the register, adding any triggered processes to the inactive event queue.

Non-blocking assignment: A non-blocking assignment (NBA) is an assignment used in always blocks that has a reg on the left-hand side and an expression on the right-hand side with the `<=` operator in between. When an NBA is encountered, the right-hand side is evaluated but not assigned to the register. Instead, an update event is added to the non-blocking assignment update queue. Any events that would be triggered by this update will be added to the relevant event queues when the NBA event is executed.

The processing of events is as shown in fig. 5.2. Start to finish in this diagram is defined as a simulation cycle. On a new cycle, the initial events will be the evaluation events caused by the change in inputs. For the first cycle of the simulation, the initial block is also an active evaluation event. The algorithm works in a loop, which ends when there are no more events to process. If there are active events, one is randomly selected and processed, adding any newly generated events to the relevant queues. In the case of no active events, if there are inactive events, they are moved to the active event queue and the loop is resumed, otherwise, if there are NBA events, they are moved to the active event queue and the loop is resumed.

The random selection of the next active event to process means the behaviour is non-deterministic. This is an essential aspect of Verilog, as these processes would be executing in parallel when synthesised to hardware. What is required to be deterministic, however, is the order in which the NBAs are processed. These should match the order of definition in the always blocks, which is not demonstrated in the flow chart in fig. 5.2.

From the above algorithm, it seems that an always block could modify a reg in its trigger list, which would trigger the always block again, potentially creating an infinite loop. I could not find details of this in the Verilog standard so, for consistency, I used the behaviour of existing simulators. To test this behaviour, I used EDA Playground [6] and tested using the Icarus Verilog 0.10.0 and Aldec Riviera Pro 2020.04 simulators. I found that always blocks cannot trigger themselves from blocking assignments, but they could from NBAs. I also found that if two always blocks that are running in parallel both cause a change to variables that trigger a third always block, the third

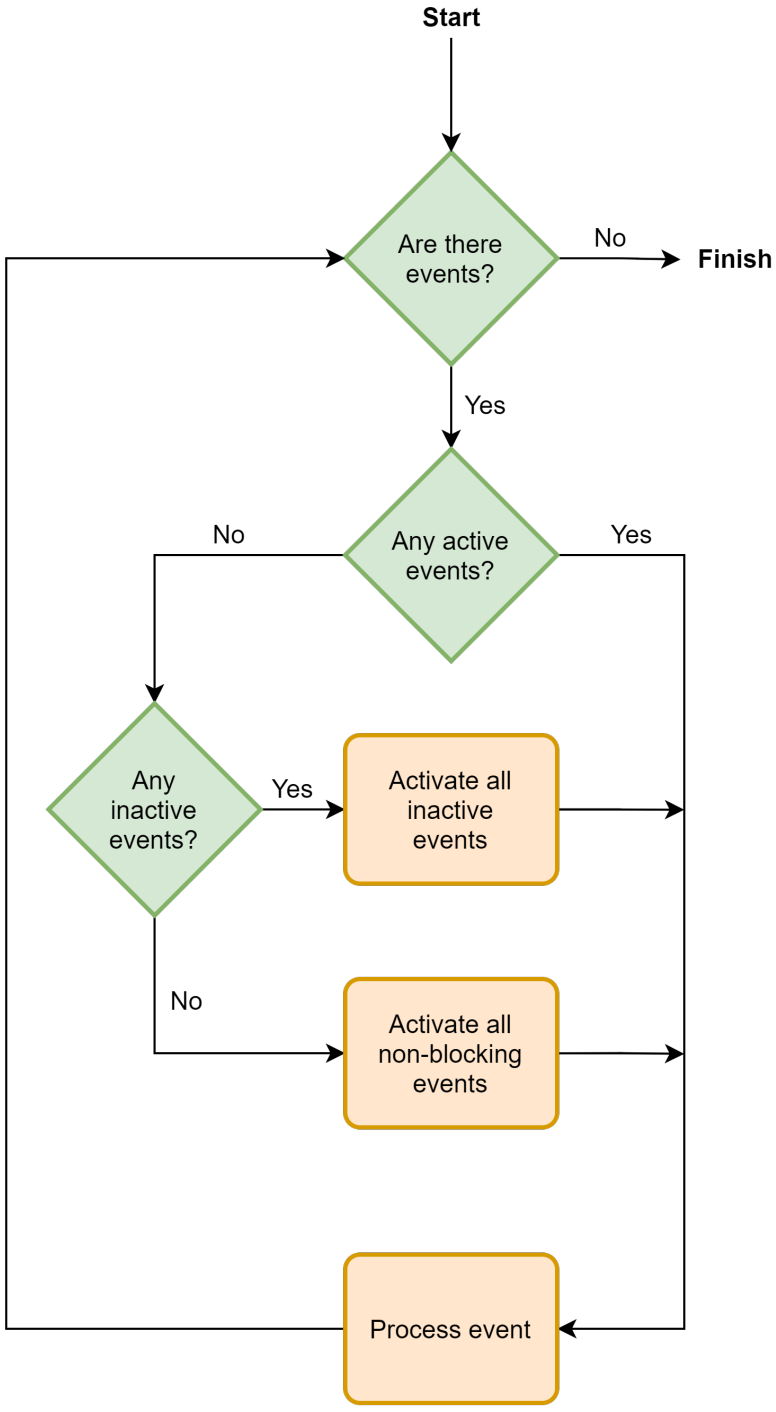

Figure 5.2: The algorithm for processing events in a Verilog simulator.

always block will only be run once. This suggests that an always block that is already scheduled to be executed cannot be scheduled again until it has been executed. To allow for this behaviour, the compiler assigned each always block in the system a unique numeric ID.

With these details and the algorithm given by the Verilog standard, a simulation algorithm was constructed and is shown in fig. 5.3. The user provides the system with how many simulation cycles to run for, the values of the inputs at each cycle, and which wires or regs should be given in the output. As the whole algorithm must loop, the first step is to check if it should terminate, which is when all the requested cycles have been processed. If it should terminate, the requested reg and wire values are returned to the user for the given cycles. If not, in the case of the first cycle, run the initial block. Next, all the triggered always blocks are collected. At this stage, the always blocks would have either been triggered by a change in input or a change in value due to the initial block. This collection of always blocks is considered a batch. The contents of a batch must be unique, meaning it cannot contain the same always block multiple times. If the batch is not empty, then the always blocks are run. This may cause other always blocks to trigger, but only always blocks not in the same batch are recorded. This means that always blocks cannot trigger each other when they're in the same batch, which matches the behaviour of simulators on EDA Playground. If the batch is empty, then a check to see if there are any NBAs queued by the always blocks is run. In the case where there are NBAs, these assignments are processed and then it loops back to collect a new batch of triggered always blocks. If there were no NBAs, then the values of the requested reg and wire values are recorded before looping back to the beginning and moving to the next cycle.

Running an always block consisted of interpreting the contained statements line by line and updating the values of the regs when they were assigned to using a blocking assignment, and queuing the updates to regs when they were assigned to using a non-blocking assignment, which would be run afterwards, as shown in fig. 5.3.

Verilog allows the user to specify the size of a reg or wire, as well as allowing some bits to be considered unknown. To represent this during simulation, a `VNum` class was made that could hold a value in a `uint64`, which is a 64 bit unsigned integer, as well as an array specifying which bits were unknown. This approach limits the maximum size of a bus to 64 bits. An alternative approach would be to store each bit as a separate value in an array. This would allow the representation of 0, 1, or unknown for each bit without limiting the bus size to 64, but was ultimately decided against as performing operations, such as addition, on an array of bits, rather than a `uint64`, was found to be significantly slower. The limit of 64 bits is unlikely to cause problems for the majority of Verilog projects, so the trade-off was considered acceptable.

5.2.4 API

All the previously discussed functionality of the backend is packaged into a single console application so that it can be easily interfaced with by the front end. The

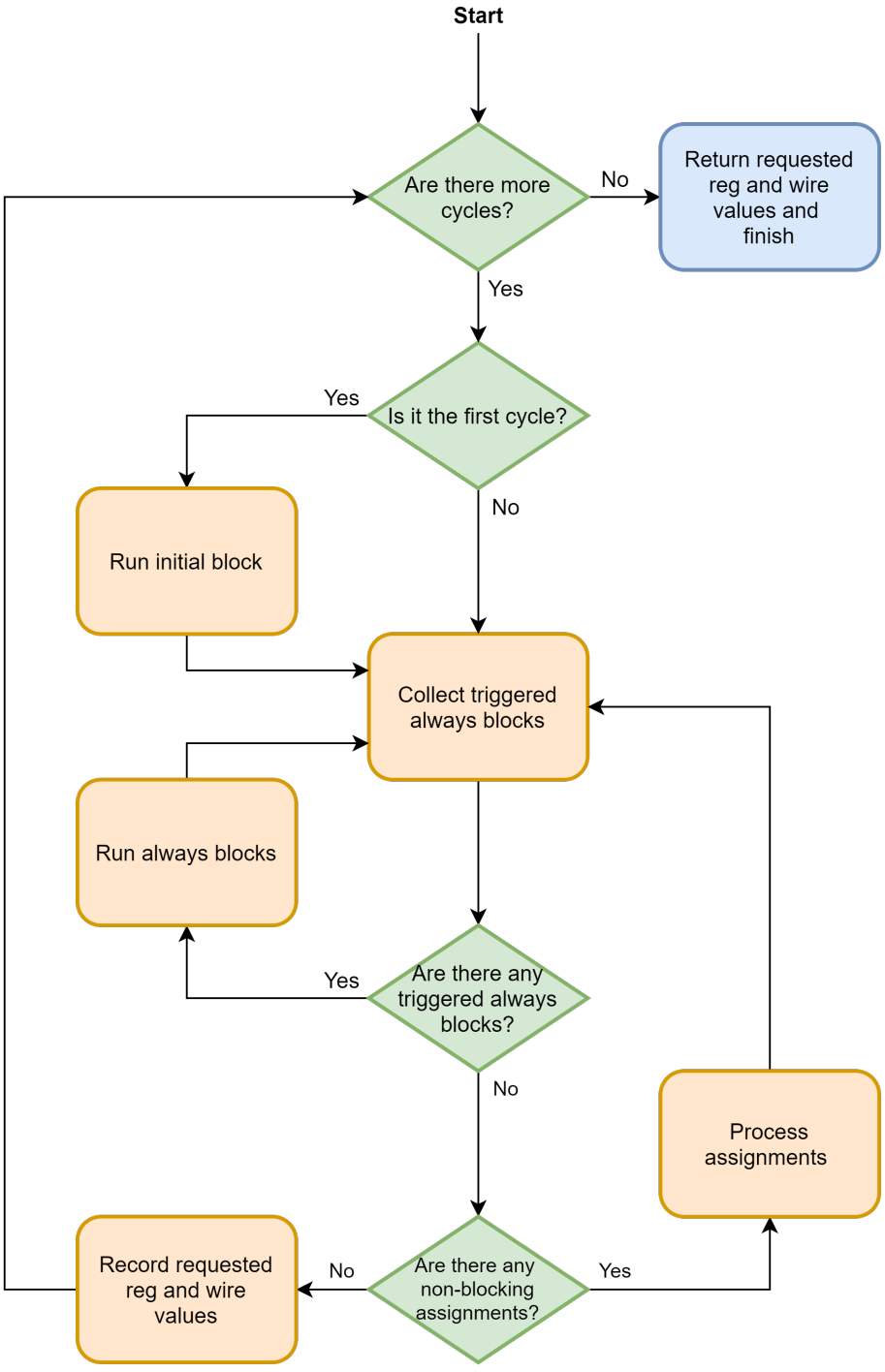

Figure 5.3: The algorithm for simulation.

`MethodDispatcher` class is used to call any API functions, but still requires the `getJob` and `postReply` functions, as discussed in section 5.1. To handle this communication, a backend command line interface (CLI) was created. The different command line options are given below:

`-d`: Providing a command line argument of `-d` to the backend application will print out the exposed function declaration in JSON format to stdout.

`-j <job>`: Providing a command line argument of `-j <job>` to the backend application, where `<job>` should be replaced with the function call job that the user wants to process, will run the job and output the response to the stdout console in JSON format.

5.3 Frontend

As previously discussed in section 4.2, the front-end of this project will be implemented as a VSCode extension. VSCode is built using Node and the VSCode extension creation tool allows setting up a project using JavaScript or TypeScript. As it is running in Node, it is likely that any language that can compile down to JavaScript would be available to use, given the correct build process, but TypeScript is a very capable language delivering type safety, making it suitable for this project.

The default set-up for a VSCode extension includes a file called `extension.ts` in the project route, which contains the functions `activate` and `deactivate`. The activate function runs when the extension is activated (extension activation is specified by the developer) and the deactivate function runs when the extension is deactivated, which is when the user closes VSCode or the workspace.

There is also an additional entry in the `package.json` file called `contributes`. This section is where the developer declares all of the static entities that are introduced by the extension, including any commands, panels, key bindings, and language support, among others. The `package.json` file is also where the developer is able to specify the activation conditions of the extension. In the case of this project, the extension is activated if there are any files in the workspace with a `.v` file extension.

To allow for error checking, VSCode provides a `Diagnostic` class in their API that allows attaching errors to specific files. When the extension is activated, a listener is added for change of active file in the editor, which compiles the project by interfacing with the backend. When the result is returned, it checks if there are any warnings or errors and attaches them to the file in question using the VSCode APIs. This will run when there is any change to the current file, so errors are reported as the user types.

To simulate their Verilog project, the user has to either click a small “play” button that appears in the upper right of the editor when they are in a Verilog file. This is made possible by declaring it in the `contributes` section of the `package.json`. To simulate, the user needs to provide the number of simulation cycles, the inputs, and

the regs and wires that they would like to be in the outputs. To make sure that the user did not have to re-enter this information, it is declared in a JSON file with a **`.simconfig.json`** file extension. As this file is edited by the user, it is important that they are aware of how it should be edited. A JSON schema is used to allow type checking and validation on the file, which also provides tool tips when hovering over relevant keywords to provide more information about what they do. To ease this process even further, if the user tries to run a simulation before the **`simconfig`** file has been made, the user is prompted to create a default one, which will use the details of the Verilog project to give default values for the inputs and requested variables.

When the simulation is run, the output is presented to the user as a waveform with the regs and wires requested. This is made possible by using WaveDrom [30], which is a JavaScript script which allows the construction of a waveform given a basic JSON object. The output of the simulator is transformed to the correct format for the WaveDrom script and then the wave is rendered in a **`WebView`**, which is a way of display web content in a panel in the VSCode window. To allow horizontal zooming of the wave, the property **`hscale`** can be edited and the wave is re-rendered. A problem quickly encountered with the method was that the values quickly expanded past the bounds of the boxes when the wave was zoomed out. To combat this, an equation to determine the maximum number of characters per value given a specific **`hscale`** value. If the number of characters is above this amount, the value is replaced with "..." and the user has to zoom in to see the actual value.

Chapter 6

Testing

An important aspect of any project is ensuring that it functions as expected. This section of the report will go through the testing techniques used for different aspects of the project and the results. To ensure that changes did not break previously working features, all pushes to the main branch would run all the tests on GitHub. All tests can be found in the GitHub repository, listed in appendix C.

6.1 Backend

The backend is written entirely in F#, meaning the NUnit [22] testing framework is available. While this framework is fairly capable, it can also become quite verbose depending on the quantity of tests that need to be run. To combat this, the testing library Expecto [9] was used. The library is a popular testing framework for .NET applications, with over 500,000 downloads [21]. It allows the developer to reduce the previously mentioned verbosity by providing a clean way to collect lists of tests. Simple helper functions can be constructed that allow the user to pass in a list of tuples containing the expected result and the test to run, and it will report the results the same as if they had been declared individually. The only downside to using Expecto over NUnit is that, as it is not built into .NET, the tests cannot be run using the `dotnet test` command, which, when using NUnit, will run all the tests in the project. Instead, the developer is required to navigate to the directory of the tests and then run it using the `dotnet run` command, which is only a minor difference.

The parser, which converts the source text to an AST, is made up of a lot of smaller parsers that are connected via combinators. To test the functionality of the parser as a whole, the individual smaller parsers can be tested. Each keyword parser is tested against the keyword that it should be able to parse. If the keyword is successfully parsed by its respective parser, then it is functioning as expected. The same is true for symbols, such as `?` or `;`. All tests written for this purpose pass.

Along with keywords and symbols, identifiers and number literals are also considered primitive parser tokens. Identifiers are used to name different constructs, such

as the name of a wire or reg. There are strict rules in the grammar regarding which characters an identifier can start with and which characters it can contain. To test this behaviour, some tests were made that should parse, which all pass, and some were made that started with acceptable body characters but not starting characters with the expectation that parsing should fail, which also all pass. Number literals in Verilog can be expressed in more ways than most programming languages, as they can be written in binary, octal, decimal, or hexadecimal bases, as well as letting the user specify any unknown bits. They also allow the user to specify the number of bits that should be used to represent the number. Tests were created to check each individual case and confirm that the correct value was captured by the parser. Tests were also constructed to check if the specified unknown bits were correctly captured by the parser, by checking the output value against the number with those bits set to 0 and also set to 1. All of these tests pass.

While the parser will return errors when the syntax is not matched, meaning that there are an extremely large number of inputs that can result in errors, the compiler will only produce errors when the input is parsed successfully, but there is then a semantic error, meaning there are significantly fewer inputs that may result in errors, making them much easier to comprehensively test. The compiler error messages were tested after the construction of the front end, meaning that as they were created it was fairly simple to test that they were produced correctly by manually writing a program that should produce the error. All errors were found to trigger correctly using this method.

To test simulation, tests were constructed that took a Verilog program as a string along with input values, and then the output of simulation was compared with some expected output values. Using this method, sample programs were written to test that the simulator could run, posedge and negedge function correctly, the initial block runs at the beginning of simulation, always blocks don't trigger themselves via blocking assignments, and that non-blocking assignments are scheduled for after the always block has finished running. All tests pass.

6.2 Frontend

As the frontend is a VSCode extension, it is difficult to test the functionality programmatically. Manual testing was therefore used to ensure the correct functionality. This included testing that the extension only activated when it was meant to, checking that the commands were only available to the user when the extension was active, and checking that the UI buttons to simulate and zoom the wave only appeared when relevant.

The frontend also allowed for easy manual testing of the frontend-backend communication via `MethodDispatcher`, as well as the API functions. These were tested by writing test Verilog projects and deliberately making mistakes, resulting in red lines being drawn and error messages displaying the relevant error. The simulation API

function was tested by simulating valid Verilog projects and printing the results to the console. The waveform generation was tested by comparing the simulator output to the wave to see if the values correctly matched up. All behaviour was as expected.

Chapter 7

Evaluation

This chapter of the report will evaluate the project. This will largely be an evaluation of how well the project has achieved the requirements laid out in chapter 3. The only overall project requirement was that it must run on Windows, macOS, and Linux, which was successfully achieved, as the project is a VSCode extension with a .NET dependency, both of which are available on all three platforms. The following sections will evaluate the relevant sections of the project.

7.1 Parser and Compiler

The only requirements on these aspects specified in section 3.3 was that they should be able to correctly parse and compile the source text into a netlist in a reasonable time frame. The speed will be evaluated when evaluating the editor in section 7.3. The functional correctness of the parser and compiler was explored in section 6.1 through both programmatic and manual testing. While this did determine the correctness of a few key aspects, due to time constraints and the amount of constructs that need testing, it is not comprehensive and further testing is required to ensure full functional correctness.

Another aspect of evaluation of these components is the error messages they produce. Due to the time constraints on the project, this aspect is still fairly underdeveloped and has large room for improvement. The parser currently outputs the default error messages produced by FParsec in the event of a failed parse, which are fairly descriptive in the majority of cases but do not provide detailed explanations of what the user has done wrong, only what the parser expected in place of the token it received. In the event that the parser backtracks and still produces an error, errors from both parser paths are reported which may be confusing to the user.

The compiler error messages are more descriptive, giving the user some indication why what they wrote is incorrect and how they may go about fixing it. One potential area of improvement is the warning messages. The system has the functionality to record these, but currently none are generated. A useful warning could be when the

user implements non-deterministic behaviour, such as changing a reg value in multiple always blocks. Also, infinite loops are currently not caught by the compiler. The simplest example of this would be an always block that triggers itself using a non-blocking assignment but other examples also exist. This is an unlikely occurrence, but ideally the user would receive an error if a program would definitely fall into an infinite loop, or a warning if it was possible.

7.2 Simulator

The requirements for the simulator were functional correctness with a loose requirement on speed. As with the parser and compiler, programmatic and manual testing was used to determine functional correctness. While this confirmed the functionality of some aspects, it is also not comprehensive and further work is required to achieve this.

The speed requirement for the simulator specified that it must be able to "simulate hundreds of cycles of a simple circuit, such as an 8-bit adder, within a few seconds". For this to bear meaning, it should represent the experience the user would have while simulating, so the following test was conducted using the frontend extension with the backend serving the simulation. The specification of the device used can be found in appendix B. To test if the simulator lives up to the requirement, using the extension a sample Verilog program was written which took two 8-bit wires as input, added them together, and assigned the value to an 8-bit output. The inputs were randomly chosen values that looped. The time was measured from when the "simulate" command was activated (either through a button press, key-binding, or running the command directly) to when the waveform was displayed to the user. To simulate 100 cycles, the program took 1.265s, and to simulate 1000 cycles, the program took 1.624s. This shows that there is a set overhead cost regardless of the number of cycles, but that the simulator is very much capable of the performance requirement laid out in section 3.4.

7.3 Editor

The decision to use VSCode as the framework for this project meant that a few of the requirements laid out in section 3.2 are achieved by default. The VSCode editor facilitates responsive typing and custom themes, which was a desired feature. The other required features (error checking, code snippets, and syntax highlighting) were added manually. Error checking was originally only conducted on file save, but on the machine I was developing on (specification given in appendix B) the compilation was fast enough that errors could be displayed as the user typed, which was one of the desired features. Syntax highlighting is provided by a TextMate grammar. Code snippets are also provided for always blocks with common layouts, as well as begin/end being considered brackets by VSCode and therefore autocompleting.

The other desirable features, such as code navigation or informative tool tips, were

unfortunately not implemented. Given more time, these features would be worth adding.

7.4 Waveform Viewer

The requirements for the waveform viewer are given in section 3.5. The required features were all successfully implemented. One small issue with the waveform viewer, though, is the zooming. While it works as expected, because it has to redraw the wave SVG when the zoom is changed (it only zooms horizontally while the vertical scale is fixed), if there are many cycles, it can slow down and feel more sluggish. While this may not be a common problem, as typically it only happens with hundreds of cycles on my computer, it may be a larger issue for lower end computers.

The desired features (interactivity between wave and editor, and save as PNG) were unfortunately not implemented as the script used to generate the wave did not offer that sort of customisability.

The output waveform is a simple SVG and, therefore, contains less functionality than a tool such as ModelSim, which is used for simulation in Quartus. This was expected, as creating a fully interactive waveform viewer would be a large undertaking.

7.5 User Interface (UI)

The requirements for the UI laid out in section 3.6 specify a requirement for simplicity when compared with competing products. This project succeeds strongly in some ways but less strongly in others. The actions a user may take while using this project are creating a Verilog project and simulating it, both of which will be discussed independently. For comparison with this project, Quartus with ModelSim for simulation will be used, as this is the primary tool currently used by Imperial College EEE and EIE students.

To create a new Verilog project, Quartus has a project creation wizard. This walks the user through specifying the project name and directory, the name of the top-level design entity, the project files and libraries included, the target device family and device, and the EDA tool settings. While not all of these are mandatory for the user to complete, as they have default values, there is an undeniable level of complexity for a user who may not even wish to target any device. In this project, to create a new project the user simply has to open a VSCode window in the desired project directory and create a file with a `.v` file extension by either clicking the new file button or right-clicking and selecting "New File" in the file explorer. This is the default process for creating files in VSCode and will be familiar for many users. It is much simpler than the Quartus' project wizard.

Running a simulation in this project is as simple as pressing the green play button that appears in the top right of any editor for a `.v` file or simconfig file. The user also

has the option of using the key-binding "ctrl + R" on Windows and Linux, or "cmd + R" on macOS. This action is as simple as it could be and certainly achieves the requirement.

Where this project is slightly less simple is the configuration of the simulation. This is done using the **`.simconfig.json`** files, which let the user specify the number of simulation cycles to run for, the inputs to the module, and which regs or wires they wish to see in the waveform along with how those values are displayed. As it is a JSON file, there is no dedicated UI to specify these things. To make it more usable, a schema is given that means the user will be informed when the enter something incorrectly and the tool tips will explain what each thing does. To ease the process a little further, if the user tries to simulate without a **`.simconfig.json`** file, they will be asked if they would like to create one, which will create a template file with information filled out corresponding to the Verilog module they are simulating. Simulating in Quartus requires the use of ModelSim, which is a separate tool that Quartus is able to load and interface with. Inputs can be provided by the user in the form of a **`.do`** file, which can be difficult to understand without instruction, or by creating a test bench in Verilog that runs the module. While the simconfig files are the most complex part of the user interface, they are still much easier to understand without instruction than the methods required for Quartus.

Chapter 8

Conclusion and Further Work

As discussed in chapter 7, this project mostly succeeds in its goal to give students learning Verilog a simple and intuitive environment in which to do so. The largest challenges were in creating the backend simulator algorithm. The documentation in the Verilog Standard was difficult to understand and, from what I could tell, not comprehensive. My initial approach to simulation would not have worked due to unexpected behaviour, and it took a few tries to get it right. The performance of the project is hard to evaluate, as I only have access to a single computer to test on. The specification of this is given in appendix B.

The project does have a few shortcomings that may make students continue to lean more towards the existing tool sets, such as Quartus Prime, but if more time was available or this project is worked on past the deadline of this coursework, these reasons may be mitigated.

A good feature to implement, given more time, would be a custom editor for the simulation configuration files. VSCode provides a well documented API for adding custom editors for specific files that could provide the user with a graphical user interface to edit the configuration. This would heavily reduce the complexity of this aspect while also making it possible for the user to only enter correct information. With the current system, the user is able to request non-existent variables, but with a GUI, they could be forced to only add variables through a drop-down menu only containing what is available.

The next feature that would be a great benefit is the option to leave feedback from within the extension. It would be possible to have a button in the extension that would allow the user to leave feedback, which could leave an issue on the GitHub repository for the project. This would help combat a couple of issues. Firstly, the functionality of the backend was not fully verified and further verification is needed, but an option to leave feedback would allow the user to report any bugs that they encounter, speeding up this process. Secondly, while the choice of language features was based on the existing materials taught at Imperial College, this may not line up with what is taught at other academic institutions or with what may be taught in the future at Imperial College. Allowing the user to leave feedback would also allow them to request new

language features, which could then be implemented and released by the developer.

An interesting possibility would be to combine it with the existing tool ISSIE, which would then allow users to create designs using block diagrams and Verilog code. ISSIE is running in an Electron framework, meaning it would be able to run the backend the same way as the VSCode extension, meaning it would have access to the same features. ISSIE has a simulator for circuits, but it likely does not follow a similar algorithm due to the event queue system used in Verilog simulation, meaning that to integrate them, ISSIE's current simulator would have to be used for circuit diagrams, but the simulator for this project would still be required for anything written in Verilog.

Chapter 9

User Guide

9.1 Installation

The project has been published to the Visual Studio Code (VSCode) extension store under the name "VIDE" from the publisher "olly-larkin". To install the extension, the user must open VSCode and navigate to the extension store and search for "VIDE". The extension should be shown in the list of results with the name and publisher as specified above. The user should click the install the extension and then reload VSCode, either by closing the application and reopening it, or by pressing "ctrl + shift + P" on Windows or Linux, or "cmd + shift + P" on macOS to open the command palette, where the "Reload Window" command can be executed.

The extension has a dependency on .NET runtime version 5.0 or higher, so this should also be installed. The extension should prompt the user to install this if they have not already when they open a folder containing a Verilog file.

9.2 Usage

9.2.1 Creating a Project

Creating a project is done using VSCode's native features. The user must open a VSCode window in the directory they wish to initialise their project in, and then create a Verilog file by either clicking the new file icon or right-clicking and selecting "New File" in the file explorer.

9.2.2 Simulator Configuration

Simualtion is configured using files with the extension `.simconfig.json`, where the name of the file should match the name of the Verilog file it is configuring. To create one already filled with default information, click the "simulate" button in the top right

of the editor with the Verilog file open. It is a green play button. This will prompt the user to create a new simulator configuration file.

The different fields of the file are discussed below:

- **`cycles`**: The number of simulation cycles to run for. This is a single positive **`int`** value greater than 0.
- **`requested vars`**: The variables that the user wishes to see in the output waveform. This is a list of JSON objects with the following fields:
  - **`name`**: This is the name of the reg or wire. If the user wishes to track a reg or wire that belongs to a module instance in the top level module, they should prefix it with the module instance name and a colon.
  - **`breakdown`**: This should be **`true`** if the user wishes to also display the individual bit values of the reg or wire in the waveform or **`false`** otherwise (only applicable if the reg or wire is a bus).
  - **`format`**: This can take the values **`bin`**, **`oct`**, **`dec`**, or **`hex`**, and controls the format of the value in the wave (only applicable if the reg or wire is a bus).
- **`inputs`**: The inputs to the module being simulated. This is a list of JSON objects with the following fields:
  - **`name`**: This is the name of the input.
  - **`repeating`**: This should be **`true`** if the input values should loop or **`false`** otherwise. If this is set to **`false`** and the number of values provided is less than the number of simulation cycles, then the last value will stay constant once it is reached.
  - **`values`**: This is a list of the values of the specified input at each simulation cycle. The values should be given in string binary format, only consisting of the characters "0", "1", "x", or "X", where "x" and "X" both represent an unknown bit.

The user is also able to add a requested variable by clicking the plus icon which appears in the top right of the editor containing the simulator configuration file. This will let the user choose from a list of available variables as well as specify if they wish to break down the bus and specify the format.

9.2.3 Simulation

Simulation is begun by clicking the green play symbol that appears in the upper right of editors for either the Verilog file of the simulation configuration file. If a configuration file does not exist, the user is prompted to create one. If it does exist, the simulation is run and the results are shown in a waveform, which can be horizontally zoomed by using the zoom symbols in the top right of the waveform view.

Bibliography

[1] *Active-HDL*. URL: https://www.aldec.com/en/products/fpga_simulation/active-hdl (visited on 06/20/2021).

[2] Christos Bouganis. *Digital System Design Lectures*. Course material. Imperial College London. 2020.

[3] Raluca Budiu. *Dark Mode vs. Light Mode: Which Is Better?* Feb. 2020. URL: https://www.nngroup.com/articles/dark-mode/ (visited on 06/21/2021).

[4] Peter Y.K. Cheung. *ELEC50001 Circuits and Systems Lecture 6*. Course material. Imperial College London. 2020.

[5] Professor Peter Y.K. Cheung. *ELEC50001 Circuits and Systems (Oct - Dec 2020)*. URL: http://www.ee.ic.ac.uk/pcheung/teaching/E2_CAS/ (visited on 06/22/2021).

[6] *EDA Playground*. URL: https://www.edaplayground.com/ (visited on 06/21/2021).

[7] *Electron*. URL: https://www.electronjs.org/ (visited on 06/22/2021).

[8] *Elmish*. URL: https://elmish.github.io/elmish/ (visited on 06/22/2021).

[9] *Expecto*. URL: https://github.com/haf/expecto (visited on 07/06/2021).

[10] *F#*. URL: https://fsharp.org/ (visited on 06/22/2021).

[11] *Fable*. URL: https://fable.io/ (visited on 06/22/2021).

[12] *FParsec Documentation*. URL: https://www.quanttec.com/fparsec/ (visited on 06/22/2021).

[13] Alan Hamlett. *WakaTime 2020 Programming Stats*. Jan. 2021. URL: https://wakatime.com/blog/43-wakatime-2020-programming-stats (visited on 06/22/2021).

[14] "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[15] *Intel Quartus Prime Pro Edition User Guide: Platform Designer*. URL: https://www.intel.com/content/www/us/en/programmable/documentation/zcn1513987282935.html (visited on 06/20/2021).

[16] *Issie*. URL: https://tomcl.github.io/issie/index.html#Verilog (visited on 06/21/2021).

[17] Jamf. *The Influence of Student Device Choice on the Modern Workplace*. URL: https://www.jamf.com/resources/e-books/the-influence-of-student-device-choice-on-the-modern-workplace/ (visited on 06/20/2021).

[18] *List of HDL simulators*. URL: https://en.wikipedia.org/wiki/List_of_HDL_simulators (visited on 06/20/2021).

[19] *Mac Bootcamp Support*. URL: https://support.apple.com/en-gb/HT201468 (visited on 06/19/2021).

[20] *ModelSim*. URL: https://eda.sw.siemens.com/en-US/ic/modelsim/ (visited on 06/20/2021).

[21] *Nuget: Expecto*. URL: https://www.nuget.org/packages/Expecto/ (visited on 07/06/2021).

[22] *NUnit*. URL: https://nunit.org/ (visited on 07/06/2021).

[23] *Quartus Prime*. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html (visited on 06/19/2021).

[24] *Quartus Prime Operating System Support*. URL: https://www.intel.com/content/www/us/en/programmable/support/support-resources/download/os-support.html (visited on 06/19/2021).

[25] *Quartus Prime User Guides*. URL: https://www.intel.com/content/www/us/en/programmable/documentation/lit-ug.html (visited on 06/19/2021).

[26] Marco Selvatici. "DEFlow: An Extensible Hardware Design Platform for Teaching Digital Electronics". MA thesis. Imperial College London, 2020.

[27] *Sigasi*. URL: https://www.sigasi.com/ (visited on 06/20/2021).

[28] *The Elm Architecture*. URL: https://guide.elm-lang.org/architecture/ (visited on 06/22/2021).

[29] *Visual Studio Code*. URL: https://code.visualstudio.com/ (visited on 06/22/2021).

[30] *WaveDrom*. URL: https://wavedrom.com/ (visited on 07/05/2021).

Appendices

Appendix A

Verilog Language Specification

A.1 Key

- { x } - 0 or more x
- [x] - 0 or 1 x
- ‘x’ - x is a literal
- R‘x’ - x is a regex pattern

A.2 Grammar

```
source_text ::= { module_declaration }

module_declaration ::=
      ‘module’ identifier list_of_ports ‘;’ { module_item } ‘endmodule’
    | ‘module’ identifier [ list_of_port_declarations ] ‘;’
           { non_port_module_item } ‘endmodule’

list_of_ports ::= ‘(’ identifier { ‘,’ identifier } ‘)’

list_of_port_declarations ::=
      ‘(’ port_declaration { ‘,’ port_declaration } ‘)’
    | ‘(’ ‘)’

port_declaration ::=
      input_declaration
    | output_declaration

module_item ::=
```

```
      port_declaration ';'
    | non_port_module_item

non_port_module_item ::=
      module_item_declaration
    | continuous_assign
    | module_instantiation
    | initial_construct
    | always_construct

module_item_declaration ::=
      net_declaration
    | reg_declaration

input_declaration ::= 'input' [ 'wire' ] [ range ] list_of_identifiers

output_declaration ::=
      'output' [ 'wire' ] [ range ] list_of_identifiers
    | 'output' 'reg' [ range ] list_of_identifiers

net_declaration ::= 'wire' [ range ] list_of_identifiers ';'

reg_declaration ::= 'reg' [ range ] list_of_identifiers ';'

list_of_identifiers ::= identifier { ',' identifier }

range ::= '[' constant_expression ':' constant_expression ']'

module_instantiation ::= identifier module_instance ';'

module_instance ::= identifier '(' list_of_port_connections ')'

list_of_port_connections ::=
      [ expression { ',' expression  } ]
    | [ named_port_conenction { ',' named_port_connection } ]

named_port_connection ::= '.' identifier [ '(' [ expression ] ')' ]

constinuous_assign ::= 'assign' list_of_net_assignments ';'

list_of_net_assignments ::= net_assignment { ',' net_assignment }

net_assignment ::= lvalue '=' expression
```

```
initial_construct ::=
      ‘initial’ blocking_assignment ‘;’
    | ‘initial’ ‘begin’ { blocking_assignment ‘;’ } ‘end’

always_construct ::= ‘always’ timing_control statement

blocking_assignment ::= lvalue ‘=’ expression

nonblocking_assignment ::= lvalue ‘<=’ expression

seq_block ::= ‘begin’ { statement } ‘end’

statement ::=
      blocking_assignment ‘;’
    | case_statement
    | conditional_statement
    | nonblocking_assignment ‘;’
    | seq_block

statement_or_null ::=
      statement
    | ‘;’

timing_control ::=
      ‘@’ ‘(’ event_expression ‘)’
    | ‘@’ [ ‘(’ ] ‘*’ [ ‘)’ ]

event_expression ::=
      ‘posedge’ expression
    | ‘negedge’ expression
    | event_expression ‘or’ event_expression
    | event_expression ‘,’ event_expression

conditional_statement ::=
      ‘if’ ‘(’ expression ‘)’ statement_or_null
           [ ‘else’ statement_or_null ]

case_statement ::=
      ‘case’ ‘(’ expression ‘)’ case_item
          { case_item } ‘endcase’

concatenation ::= ‘{’ expression { ‘,’ expression } ‘}’
```

```
constant_concatentation ::=
      ‘{’ constant_expression { ‘,’ constant_expression } ‘}’

constant_expression ::=
      constant_primary
    | unary_operator constant_primary
    | constant_expression binary_operator constant_expression
    | constant_expression ‘?’ constant_expression ‘:’ constant_expression

expression ::=
      primary
    | unary_operator primary
    | expression binary_operator expression
    | expression ‘?’ expression ‘:’ expression

constant_primary ::=
      number
    | constant_concatenation
    | ‘(’ constant_expression ‘)’

primary ::=
      number
    | identifier [ range ]
    | concatenation
    | ‘(’ expression ‘)’

lvalue ::=
      identifier [ range ]
    | ‘{’ lvalue { ‘,’ lvalue } ‘}’

unary_operator ::=
      ‘+’
    | ‘-’
    | ‘!’
    | ‘~’
    | ‘&’
    | ‘~&’
    | ‘|’
    | ‘~|’
    | ‘^’
    | ‘~^’
    | ‘^~’
```

```
binary_operator ::=
      '+'
    | '-'
    | '*'
    | '/'
    | '%'
    | '=='
    | '!='
    | '==='
    | '!=='
    | '&&'
    | '||'
    | '**'
    | '<'
    | '<='
    | '>'
    | '>='
    | '&'
    | '|'
    | '^'
    | '^~'
    | '~^'
    | '>>'
    | '<<'
    | '>>>'
    | '<<<'

number ::=
      decimal_number
    | octal_number
    | binary_number
    | hex_number

x_digit ::= 'x' | 'X'

decimal_number ::=
      unsigned_number
    | [ unsigned_number ] decimal_base unsigned_number
    | [ unsigned_number ] decimal_base x_digit

octal_number ::= [ unsigned_number ] octal_base octal_value
```

```
binary_number ::= [ unsigned_number ] binary_base binary_value

hex_number ::= [ unsigned_number ] hex_base hex_value

unsigned_number ::= decimal_digit { ‘_’ | decimal_digit }

octal_value ::= octal_digit { ‘_’ | octal_digit }

binary_value ::= binary_digit { ‘_’ | binary_digit }

hex_value ::= hex_digit { ‘_’ | hex_digit }

decimal_base ::= ‘\’’ [ ‘d’ | ‘D’ ]

octal_base ::= ‘\’’ [ ‘o’ | ‘O’ ]

binary_base ::= ‘\’’ [ ‘b’ | ‘B’ ]

hex_base ::= ‘\’’ [ ‘h’ | ‘H’ ]

decimal_digit ::= R‘[0-9]’

octal_digit ::= R‘[0-7]’ | x_digit

binary_digit ::= ‘0’ | ‘1’ | x_digit

hex_digit ::= R‘[0-9a-fA-F]’ | x_digit

identifier ::= R‘[a-zA-Z_][a-zA-Z0-9_$]*’

comment ::=
      ‘//’ { AnyASCIICharExceptNewLine } ‘\n’
    | ‘/*’ { AnyASCIICharsExceptClosingComment } ‘*/’
```

Appendix B

Device Specification

The laptop used for development and testing:

- Device: ASUS Zenbook Pro Duo
- Processor: Intel(R) Core(TM) i9-9980HK CPU @ 2.40GHz
- RAM: 32GB
- Graphics Card: Intel(R) UHD Graphics 630 and NVIDIA GeForce RTX 2060
- Operating System: Windows 10 Home 20H2 19042.1083

Appendix C

Project Git Repository

https://github.com/OllyLarkinFYP/FYP