Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2020

Project Title:	**DEflow: An Extensible Hardware Design Platform for Teaching Digital Electronics**
Student:	**Marco Selvatici**
CID:	**01360752**
Course:	**EIE3**
Project Supervisor:	**Dr Thomas J. W. Clarke**
Second Marker:	**Dr Edward Stott**

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, electronic copies of my final year project report to both Blackboard and the EEE coursework submission system.

I affirm that the two copies of the report are identical.

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

Acknowledgments

I would like to thank my supervisor, Thomas Clarke, for his invaluable guidance and suggestions.

I would like to express my deep and sincere gratitude to my family and friends, and in particular to my parents, whose encouragement supported me throughout my course of studies in an abroad university. Their love and teachings guided me through my academic journey and will continue to do so for the rest of my life.

I am extremely grateful to Karen Sarmiento and her family, for their care and hospitality in these uncertain times of a global pandemic. Their kindness and generosity always made me feel welcome.

Finally, I would like to extend my gratitude to all the students who altruistically spent their time helping me by taking part in the user feedback survey.

1 Abstract

Students learning digital electronics currently have to use complex industrial software targeted to experienced engineers, which introduces extra cognitive load. This dissertation describes a highly extensible open-source cross-platform hardware design application with a strong emphasis on usability and intuitiveness. The application allows students to design, analyse and simulate both combinatorial and synchronous digital circuits. The extensibility and maintainability have been guaranteed by the adoption of a functional reactive programming architecture, implemented in a strongly typed functional language. The usability of the platform has been evaluated by collecting user feedback, by comparing it with alternative tools such as Quartus, and by creating large designs such as a CPU. The evaluation shows that users are much more easily able to familiarize and use this system in comparison to the alternatives.

Contents

2 Introduction

As part of their university studies, first year electronics engineers learn the foundations of digital electronics both in lectures and in practical laboratories. Throughout these laboratories, students design and simulate digital circuits. This enables them to grasp the concepts surrounding the use of combinatorial and synchronous logic, which are the bases for every complex electronic system.

During the laboratory sessions, Imperial College EE students use a hardware design platform called Quartus [56]. Quartus is an industrial software targeted to an audience of experienced hardware engineers. Therefore, it contains an abundance of features, such as circuit timing analysis, which are widely used in industry. However, a large amount of these features are unnecessary and introduce extra cognitive load for first year students. A survey that I conducted among university students showed that 88% of them tend to get confused in Quartus complex menus system, while 63% struggles to understand the error messages provided by the application. Additionally, 75% of the surveyed students agreed that a simpler application would have allowed them to learn digital electronics concepts with less guidance, as well as improving their overall laboratory experience (survey results are analysed in section 8).

The aim of this project is to design and develop an alternative software application to support university students in learning Digital Electronics and Computer Architecture. The application will allow students to design synchronous digital circuits using a graphical circuit diagram editor. It will also enable students to easily analyse, validate and simulate such circuits.

The application aims to improve on programs like Quartus by offering a simpler interface and a more intuitive experience. This, in turn, simplifies students' learning processes by removing the overheads of familiarising with complex hardware design software. In fact, students should be able to use the application without the need of any guide or reference manual. Furthermore, the application should be cross-platform and easy to install, so every student can use it from their personal devices without having to SSH into the university servers (which is currently necessary to use Quartus). This will allow students with an unstable internet connection to be able to work remotely. Additionally, since every desirable feature is unlikely to be implemented within the limited timespan of the project, maintainability and extensibility will also be among the main focuses. This allows the application to be modified in the future as the teaching objectives evolve. Finally, this innovative hardware design platform will be open-sourced so individuals and institutions outside of Imperial College can use or change it as they wish.

This report will first present an analysis of the existing hardware design platforms, and what technology could be used to create a cross-platform, more intuitive and easily maintainable application (section 3). These considerations will converge into a precise set of requirements that the application ought to have in order to be successfully employed for digital electronics teaching (section 4). An analysis of the main design and implementation choices that have been made during the project will be presented in sections 5 and 6. The produced application will then be evaluated in sections 7 and 8 via a number of experiments, such as user feedback collection, an in-depth features comparison against the alternatives and the design of a large digital circuit (a RISC CPU). Finally, a set of future extensions and improvements for the application will be proposed in section 9.1, and guides to get started with it both as developers and users will be presented in appendix A.

3 Background

This section provides the background necessary for a reader to understand how the project fits in the current digital electronics teaching world, as well as the technologies that will be used. We will firstly discuss some of the characteristics of an ideal digital electronics application (subsection 3.1). This will set a framework to analyse existing alternatives to this project's deliverable and understand their shortcomings (subsection 3.2). Lastly, an analysis of the technologies that will be used for the project will be presented, with justifications of the choices and alternatives considered (subsections 3.3 and 3.4).

3.1 Didactic Considerations

This subsection will investigate what characteristics an ideal digital electronics teaching tool should have. This considerations will be guided by the analysis of students experience during digital electronics laboratory sessions (subsection 3.1.1), and by analysing what hardware design methodology is best fit for first year students (subsection 3.1.2).

3.1.1 Laboratory Environment

As mentioned in the introduction, the application targets the academic environment. In particular, it ought to be adopted for the teaching of Imperial College EE Digital Electronics and Computer Architecture (DECA) module [21]. The DECA module involves around 40 hours of laboratories, which aim to strengthen students' understanding of digital electronics concepts. During these laboratory sessions, students are required to build complex computer systems from basic electronics components such as gates. Additionally, students are expected to grasp how data can be represented and manipulated in the form of binary signals.

To reach such objectives, the laboratory sessions cover a large amount of material and this can put students under time pressure. In fact, it is not uncommon for students to work extra hours outside the allotted time to keep up with the material.

In such a scenario, it is important to facilitate students' learning experience as much as possible by providing tools that are intuitive to use and quick to get started with. This, in turn, lets students focus on the didactic objectives of the laboratory sessions, and possibly lead to a more pleasurable overall laboratory experience. The tools currently employed, such as Quartus, do not provide this simplicity and ease of use. In fact, a survey conducted on Imperial College and Cambridge University students revealed that 75% of them agreed that using a simpler tool would have allowed them to learn digital electronics concepts with less guidance, as well as improving their overall laboratory experience (the data are presented in section 8.2).

Ideally, the hardware design application would be so intuitive that no user manual or guide should be necessary for students to perform both familiar and unfamiliar tasks. Furthermore, such an ideal program should allow students to quickly validate and experiment with their designs, receiving instantaneous feedback if they introduce errors in the diagram. This way students can make the most out of laboratory sessions, without wasting precious time waiting for the application's calculations to complete.

	Circuit Diagram	HDL
Easily visible dataflows	✓	✗
Closely relates to hardware	✓	✗
Gives overview picture	✓	✗
Requires no programming skills	✓	✗
Easy to parametrise	✗	✓
Good for large and complex projects	✗	✓

Table 1: Comparison of circuit diagrams and hardware description languages. Inspired from [85].

3.1.2 Schematic Circuit Diagrams vs Hardware Description Languages

In order for students to familiarise with digital electronics concepts, they need to design digital circuits. There exists two main approaches to design a digital circuit: schematic circuit diagrams and hardware description languages (HDL). The former is graphical representation of the digital circuits, which uses standardised symbols to show components and interconnections among them. The latter is a textual representation of the circuit, formed of statements and control structures.

Advantages and disadvantages for both approaches are reported in table 1. Circuit diagrams provide a visual representation which is propaedeutic for learning [96, 72]. Even more importantly, circuit diagrams require no programming skills, which is crucial given that EE students are supposed to get started with digital electronics laboratories as early as the first term of first year. On the other hand, the shortcomings of circuit diagrams (as per table 1) tend to have lesser impact for the size of the designs created during first year digital electronics laboratories, which should not exceed the complexity of a basic RISC CPU.

Overall, circuit diagrams seem to be a better fit for beginners. Once the students develop a solid understanding of electronics concepts, it is reasonable to replace circuit diagrams with HDLs, which are more powerful when dealing with large projects.

Currently, circuit diagrams are used in most first year DECA laboratories. Towards the end of the academic year, students are introduced to a subset of Verilog HDL [82] in order to design some combinatorial circuits like a CPU decoder. According to a conversation with the course lecturer, the reason for this choice is both to ease the process of describing complex combinatorial logic and to introduce the HDL language which will be used during the second academic year. Though this head start is desirable, it is not strictly necessary to meet first year didactic objectives.

3.1.3 Summary

This section discussed some of the characteristics that an ideal digital electronics teaching tool should have. Such a tool should allow students to design circuits via a diagram editor, which, in turn, helps them to get a visual understanding of electronics concepts. Furthermore, the tool should be intuitive in order to remove the overheads that currently negatively affect students' learning experience. These results will be used to inform the requirements capture in section 4.

	CircuitMaker [9]	Quartus [56]	KiCad EDA [46]	SimulIDE [66]	BrainBox [6]
Diagram editor	✓	✓	✓	✓	✓
HDL editor	✗	✓	✗	✗	✗
Simulation	✗	✓	✗	✓	✓
Teaching oriented	✗	✗	✗	✓	✓
Open-source	✓	✗	✓	✓	✓
Actively maintained	✓	✓	✓	✓	✓
Supported platforms	Windows	Windows, Linux	Windows, Linux, MacOS	Windows, Linux	Windows, Linux, MacOS

Table 2: Comparison of some of the existing digital circuit design tools.

3.2 Existing Hardware Design Applications

Digital hardware design applications have been around for many years. This section will analyse some of the currently available tools, discussing their strengths and shortcomings.

There exist several digital circuit design tools, each with a slightly different set of features. Some of the most famous are reported in table 2. CircuitMaker [9], Quartus [56] and KiCad [46] are developed for industrial usage, and tend to focus on an audience of experienced engineers. As such, they offer a variety of features that are not required nor understood by a novice student. Having a large set of features generally means that the user has to navigate more menus in order to achieve their goals. Given the relatively short amount of time that students are expected to spend in the laboratory, it is important to ease their learning process as much as possible by letting them focus on learning concepts, more than learning how to use complex tools.

On the other hand, SimulIDE [66] and Brainbox [6] have been designed primarily as teaching tools. Therefore, they tend to have much simpler user interfaces, and favour simplicity over the amount of features supported.

I decided to further analyse the three tools that fit the largest number of prerequisites from table 2. These tools are: Quartus [56], which is the program currently used in DECA laboratories; SimulIDE [66] and BrainBox [6], which seem to provide a reasonable set of features, while being relatively simple to use.

Quartus is an industrial software which includes countless tools for working with field programmable gate arrays (FPGAs), system on chip (SoCs), and complex programmable logic devices (CPLDs). Functionalities include design, synthesis, optimization, verification, and simulation of digital circuits. Quartus is developed and maintained by Intel, which provides access to it via a paid license. Quartus Lite (which has a subset of Quartus' functionalities) is freely available. Being such a powerful and large application, Quartus is arguably difficult to familiarise with. A clear example is the numerous amount of menus that a user is required to go through in order to set up a new project (see figure 1). A survey conducted among former Quartus users showed that 88% of them get confused while trying to perform unfamiliar tasks with the application, and they have to consult a guide to get unstuck (result presented in section 8.2). Therefore, students have to waste precious laboratory time by having to learn application-specific procedures or repeatedly consulting a user's guide, rather than learning digital electronics concepts.

SimulIDE is an electronic circuit simulator intended for hobbyists or students that mainly focuses on ease of use and simplicity. It allows users to draw diagrams with an intuitive drag and drop interface, and

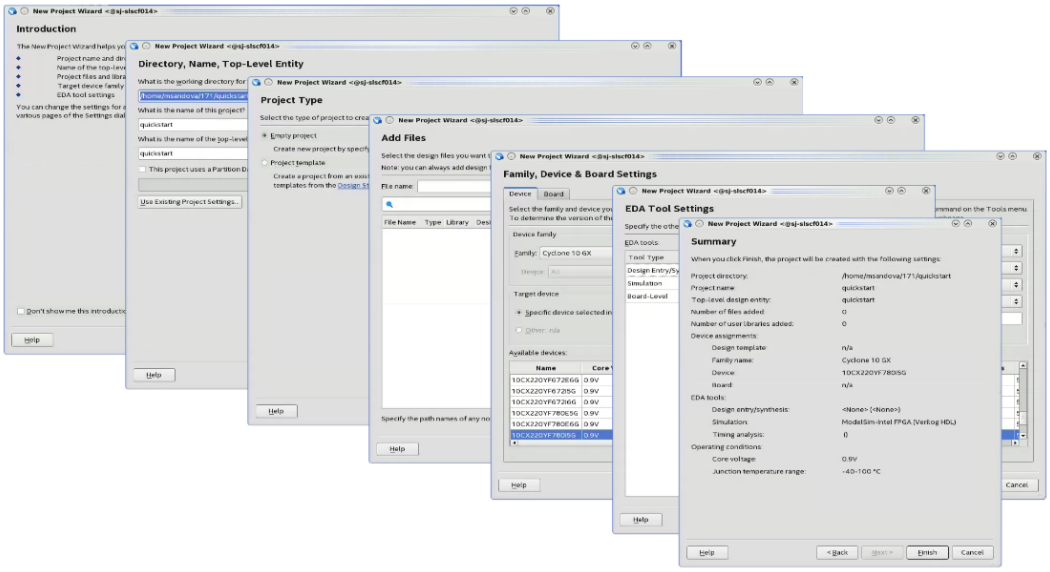

Figure 1: Series of menus in Quartus project setup wizard. Image credits [57].

to run simulations. On the other hand, the application seems to be lacking important features as the ability to pack multiple wires into buses. Another major drawback is that SimulIDE is developed and maintained by a single developer, who has received complaints about how difficult it is to contribute to the project [65]. Furthermore, SimulIDE is mostly developed in C++ and Assembly [47], and extending it to add the required missing features may be a tedious task.

BrainBox is conceptually similar to SimulIDE: it offers an intuitive user interface for creating circuit diagrams, and is primarily designed for students. Unfortunately, also BrainBox lacks key features such as memories and buses. BrainBox is a cloud based application developed in JavaScript with Node.js backend. Therefore, it can be hosted on a server or run locally if the user can set up their own node server. Being written entirely in JavaScript, the project is probably more easily maintainable and extensible than SimulIDE. However, because of its weak type system JavaScript is not the most reliable choice for a large-scale software application (more on this in section 3.3.3).

During this research it was found that, though there exist more intuitive and user-friendly alternatives to Quartus, they tend to be limited in terms of capabilities. These considerations will be used in the requirements capture (section 4) to define an application which aims to have all the features required for introducing students to digital electronics, while being intuitive to use, open-source, cross-platform and easy to extend. Furthermore, an in-depth features analysis of the aforementioned applications will guide the comparison with the final project deliverable in section 8.2.

3.3 Technological Considerations

When setting up a large software project, there are several technological choices that have to be made. In this subsection I will introduce a series of technologies, their benefits and drawbacks. These considerations will ultimately converge into a unique technology stack in subsection 3.4. As mentioned in sections 2 and

3.2, the application should be cross-platform, easy to extend and to maintain. These three pillars will largely influence the stack of technology being chosen.

Firstly, the choice of the software ecosystem will be discussed in subsections 3.3.1 and 3.3.2. Subsection 3.3.3 will discuss the choice for the programming language in the selected ecosystem. Lastly, subsection 3.3.4 will describe the technology chosen to create a maintainable user interface.

3.3.1 Software Ecosystem

One of the most important choices to make when starting the development of a new product is the software ecosystem to use. In fact, the ecosystem largely affects the other technologies of the system such as the programming language or the user interface infrastructure. The ideal ecosystem for this application ought to have the following features:

Cross-Platform The product will likely be used by students and academic staff who own devices running different operating systems. It is therefore important that the product is compatible and easy to set up on all the main platforms (Windows, Linux and MacOS).

Open-Source The product is intended to be distributed with a GPLv3 license [36], to make it open-source and free to use for institutions and individuals. Therefore, it should rely on equally open-source and free technologies.

Powerful Libraries When creating a large software project it is important to be able to rely on mature and maintained libraries to offload part of the implementation burden and focus more on the innovative parts of the application.

Commonly Known As the application will probably benefit from extensions and changes in the future, it is important to make use of widely known technologies to ease the maintenance and extension tasks.

The main candidates are the web ecosystem and desktop ecosystems such as the .NET, Java or C++ ones. Desktop applications tend to be very responsive and lightweight, and can be cross-platform if developed with toolkits like JavaFX [41] or QT [55]. On the other hand, learning how to use these toolkits may be a lengthy process and some of the technologies involved tend to be cumbersome (for example, C++ drawing libraries have complicated API [48]).

The web ecosystem is a good fit for all the requirements. It is cross-platform by design, can benefit from the power and simplicity of JavaScript, and many JavaScript tools and libraries are open-source. Moreover, the relatively recent development of tools like Electron [22] allows developers to package web applications and run them as desktop applications. Hence, it is possible to take advantage of the power of the web ecosystem while deploying desktop applications. This is important since a cloud based web application would be unusable without a stable internet connection, which is undesirable.

On the other hand, the heavy reliance of this ecosystem on the JavaScript programming language poses some concerns regarding scalability and maintainability. In fact, despite being a very powerful and quick solution for small web applications and websites, JavaScript has some fundamental issues related to its weak type system that make it difficult to safely use it for large scale projects. Section 3.3.3 will describe JavaScript limitations and how they can be overtaken while keeping all the advantages of the web ecosystem.

3.3.2 Electron

Electron is an open-source framework developed and maintained by GitHub. It allows developers to build cross-platform applications using HTML, CSS, and JavaScript. This can be achieved because Electron bundles Node.js [52] and the Chromium web browser [8] into a single package. Node.js is a framework that allows one to run JavaScript code outside the browser sandbox, therefore allowing it to access OS level API like the file system ones. Chromium is used as a rendering engine to display content designed with HTML and CSS. Therefore, Electron provides all the advantages of the web ecosystem in a desktop environment. However, it also has a few downsides:

- A single Electron application will be often more than 60 MB in size. This is because, as aforementioned, Electron packages both Node.js and Chromium into a single bundle.
- Like the Chromium web browser, Electron applications require a considerable amount of RAM to run. For example, a simple hello world app needs around 70 MB of RAM, compared to the 20 MB required by an equivalent QT-based counterpart [23].
- The UI may feel less responsive than native desktop applications on less powerful machines. The responsiveness is similar to the one experienced when running web applications in the browser.

A promising alternative to Electron is Proton Native [54]. Proton is conceptually similar to Electron, but instead of using Chromium as a rendering engine, it generates platform specific UI code. Native applications tend to run faster and be more lightweight than their Electron counterparts. On the other hand, Proton is a relatively new (first released in 2018), with a smaller community than Electron. Also, it has never been used for big and successful products such as the Electron-based Visual Studio Code [73], Atom [4], Slack Desktop [67] and WhatsApp Desktop [77].

In summary, the web ecosystem has been chosen for the application. Thanks to the adoption of Electron, the project deliverable can be packaged and deployed as a desktop application while still maintaining the benefits of the web ecosystem, such as being cross-platform. This decision will influence the other technological choices of this project.

3.3.3 Programming Language

Sections 3.3.1 and 3.3.2 provided a rationale for the choice of the software ecosystem being used for the project. This section will analyse the choice of the main programming language to use for the implementation of the application.

The web ecosystem relies heavily on the powerful and versatile JavaScript programming language. JavaScript has several characteristics that justify its ubiquitousness in the current development world, such as:

Beginner-Friendly Syntax The language is notoriously easier to learn and use than other languages like C or Java. Furthermore, no other tool aside from a web browser is required to get started with it.

Interpreted There exist many powerful JavaScript interpreters, such as Google's V8 engine (which powers both Google Chrome and Chromium) [70]. An interpreted language is generally faster for development because it allows developers to manually test the code quickly, avoiding expensive recompilations.

Automatic Memory Management The developer does not have to worry about garbage collection and other low level issues.

Active Community JavaScript gets improved constantly, and new libraries/frameworks are released on a weekly basis. In 2018, more than 1 million Node.js packages were downloaded every day [51].

However, JavaScript has several shortcomings related to its dynamic type system. Dynamic typing means that data types are determined at runtime, instead of compile time. This introduces a class of runtime errors that would be otherwise blocked by the compiler. JavaScript tries to partially mitigate this issue by providing some obscure type conversions. In fact, it is perfectly valid to compare an integer with an array, and the expression [ ] == 0 evaluates to true [42]. Additionally, dynamically typed code is less self-documenting since there are no types in the function signatures. The developer has to manually add them in the function docstring and ensure that they remain consistent in future changes of the code.

More specifically, JavaScript is a prototype-based programming language [89]. Though such languages are very flexible, they tend to be affected by a class of security vulnerabilities called prototype poisoning. Exploiting a prototype poisoning vulnerability can often lead to arbitrary malicious code execution [93]. There exist techniques to mitigate this issue, for example using Defensive JavaScript [93], a typed subset of JavaScript, but they constraint the number of language features you can access.

In recent years, many statically typed languages which can be compiled to JavaScript were created to mitigate its main weaknesses while taking advantage of its strengths. Famous examples include TypeScript [69], ClojureScript [10], Elm [24] and Dart [13].

Similarly, functional programming languages tend to have very strong static type systems, and several of them can nowadays be compiled to JavaScript. The key idea behind functional programming languages is immutability: functional code is composed of a series of pure functions which take an input and transform it into its output without side-effects. This allows developers to clearly separate the data (input) from transformations on the data (functions). In addition to mitigating JavaScript issues related to dynamic typing, functional programming languages provide a series of advantages:

- The code is generally easier to reason about, in comparison to object-oriented code. Since functions transform immutable data, programmers do not have to worry about state maintenance issues.
- The code is easier to test. To test a function one simply needs to provide an input and compare it to the expected output. This means that developers do not have to worry about bringing the system into a certain state and checking that it changes as expected.
- The code is easier to understand and maintain. Usually all that is required to refactor code is to change the types into the new desired ones, and follow the compiler indications about what other parts of the code needs to be changed.
- Static type checking provides stronger code correctness guarantees. The compiler is very strict about making sure types are as expected and all cases are considered in pattern matching. This drastically reduces the possibility of introducing bugs and, the vast majority of the time, the code works on the first attempt.

During the third year High Level Programming course I became familiar with the F# programming language. F# is a hybrid language: it is primarily intended to be used as a functional programming language but it also supports mutable data and Object Oriented Programming constructs such as classes. Nonetheless, the use of mutable data is discouraged by the syntax of the language itself, as the developer has to explicitly mark such variables with the 'mutable' keyword. Finally, F# has a strong Hindley-Milner type system, which allows static type inference [88]. This provides strong code correctness guarantees, as explained earlier.

F# code can be compiled to JavaScript via the powerful Fable compiler [27]. Therefore, this tool allows developers to simultaneously benefit from the power of F# and the versatility of JavaScript. Other strongly typed functional languages such as Haskell [35] and OCaml [12] can be compiled to JavaScript too, but I am not familiar with either. Furthermore, the application will likely be maintained by Imperial College students who can learn F# from the High Level Programming course.

In summary, JavaScript is a versatile language, but its dynamic typing makes it difficult for a developer to ensure code correctness as the codebase grows. On the other hand, functional programming languages such as F# feature a static type system which greatly helps the developer in maintaining a large codebase. Additionally, F# can be compiled to JavaScript via tools like Fable. The choice of F# as the main programming language for the project heavily affects the Implementation of the application (section 5). Finally, section 8.4 will highlight the benefits of such a choice specifically for the development of DEflow.

3.3.4 User Interface Infrastructure

A crucial component of any user-facing application is the Graphical User Interface (GUI or UI), which has the role of transforming user intentions into changes to the application state. It is unfortunately very common to underestimate the importance of writing deeply reasoned and structured UI code, but this mistake leads to the production of anti-patterns that reduce productivity and make the code more prone to bugs [83].

One of the key challenges in creating robust and maintainable user interfaces comes from the inherent event-driven nature of the UI logic. In other words, the UI logic has to handle an unorganised stream of events, and doing so in an organised and clean way is a difficult task.

One of the most promising approaches to UI design is Functional Reactive Programming (FRP). This concept was first introduced in 1997 in the animation field [90], and has since been successfully employed in several other fields such as robotics [95], video games [86] and music synthesis [92]. This paradigm gained widespread attention in the UI domain after the release of the Elm programming language in 2012 [87, 24]. FRP is:

Declarative The developer specifies *what* the desired result of the computation is rather than *how* the computation should be performed.

Reactive Computations are triggered as result of user interactions.

In order to better understand what FRP is and what advantages it can bring in the UI domain, we will consider a simple example where the text content of the `outputTextArea` is dynamically updated to match what the user types into the `inputTextArea` (see figure 2). The pseudocode for a FRP implementation would look like the following:

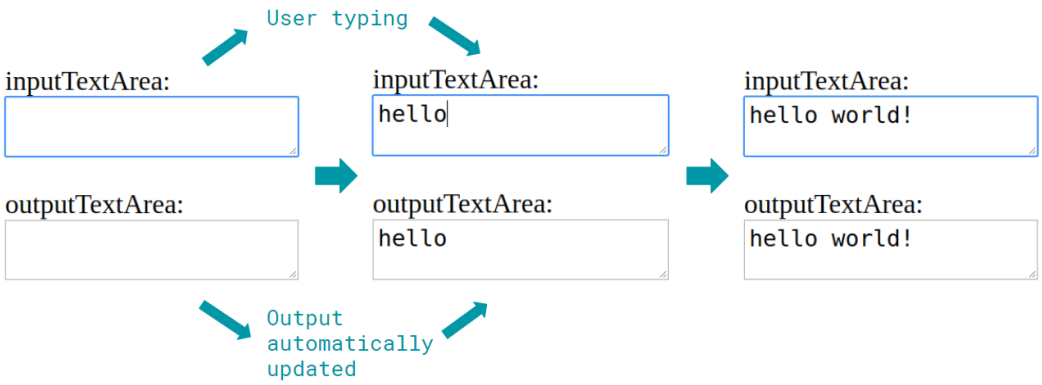

Figure 2: Simple application.

```
inputTextArea (initialValue = "")
outputTextArea (value = inputTextArea.value)
```

The underlying FRP runtime takes care of updating the output upon changes to the input. In contrast, a more traditional event-driven approach would look like this:

```
inputTextArea (initialValue = "")
outputTextArea (initialValue = "")

event handler: user changes value of inputTextArea {
    text = getValueOf(inputTextArea)
    setValueOf(outputTextArea, text)
}
```

It is apparent that FRP moves the focus on the desired result (the `outputTextArea` displays the same text of the `inputTextArea`), leaving out *how* to obtain such a result (the handling of events). The power of FRP becomes more evident when the UI requires many complex interactions.

A more subtle but equally crucial difference is that the FRP approach is 'functional'. Consider the same example as before, but let the `initialValue` of the `inputTextArea` be "abc" (see figure 3). Before any user interaction, the value of `outputTextArea` in the traditional approach will be the `outputTextArea`'s `initialValue`, which does not necessarily match the one of the `inputTextArea`, breaking the semantic of the application. This is because the result of the traditional implementation depends on the *history of past events*. In other words, if the event handler has never been triggered in the past, the content of the `outputTextArea` is not guaranteed to be as expected. This issue can be fixed by introducing the following event handler:

```
event handler: initialisation of outputTextArea {
    text = getValueOf(inputTextArea)
    setValueOf(outputTextArea, text)
}
```

Figure 3: Simple GUI, before any user interaction and with initialValue of inputTextArea set to "abc".

However, having to explicitly handle all of the corner cases leads to more complex and hard-to-maintain code.

On the other hand, the FRP implementation guarantees that, no matter what happened in the past, the value of the `outputTextArea` matches the one of the `inputTextArea`. In fact, the whole UI is fully specified by the current value of the `inputTextArea`. The FRP approach is conceptually similar to specifying a function that, given a certain input, always produces the corresponding output *irrespective of the history of the system*.

In summary, the adoption of a Functional Reactive Programming architecture allows to define UI in a declarative way, abstracting from the low level event handling mechanisms. This, in turn, greatly helps reasoning about the UI code itself, and provides much stronger correctness guarantees. A more thorough analysis of this paradigm will be proposed in the section 3.4, when looking at the specific FRP technology adopted for the project.

3.3.5 Summary

One of the most important choices for a large software project is its technological infrastructure. In this subsection a series of alternatives have been presented, which will be used to define the complete technology stack in section 3.4.

3.4 Technology Stack

Subsection 3.3 described a series of technologies, including the web ecosystem, the F# programming language and the Functional Reactive Programming paradigm. In this subsection, these technologies and concepts will be used to define the technology stack for this project.

As briefly mentioned in subsection 3.3.3, it is possible to make use of the F# functional programming language together with the web ecosystem by using the Fable compiler. Fable transparently transforms F# code into a human readable and modern JavaScript code, which follows the recent language standards and does not make use of obscure language features. The use of both F# and JavaScript in the same application, implies that there will be two types of dependencies, one for each language. Luckily, Fable is a mature and commonly used tool, so several solutions exist to hide the complexity given by the interoperation of the two languages. For example, Femto [31] is a package manager that allows developers to handle both JavaScript and F# dependencies in the same way.

Furthermore, Fable is extremely well integrated with Electron. As mentioned in section 3.3.2, Electron is a tool that allows developers to package web applications and deploy them as cross-platform desktop applications, therefore allowing them to access the file system and other OS level utilities.

Making use of the F# programming language, allows developers to rely on the Elmish library [25]. Elmish lets developers take advantage of the power of Functional Reactive Programming when designing and creating UIs (see 3.3.4). As the name suggests, Elmish closely follows the FRP concepts introduced by the Elm programming language, but exposes them as a library for a much more powerful and general purpose language, F#. More details about Elmish will be given in the example below.

Together, this set of technologies forms the Fable-Elmish-Electron technology stack. This stack allows to build FRP, cross-platform application with F#. This technology stack may feel complex, but it is very developer-friendly thanks to the existence of tools (such as the aforementioned Femto) which do an excellent job in hiding the complexity from the developers. This architecture is heavily functional, therefore it provides the benefits described in section 3.3.3, such as type safety and strong correctness guarantees provided by the compiler. The benefits of this technology stack will be further emphasized in section 8.4.

We will now examine an example Elmish application which can be compiled with Fable and run on different platforms thanks to Electron. Elmish (like Elm), implements the FRP concepts via the *Model-View-Update* architecture (MVU). The main components of this architecture are:

Model An immutable data structure that represents the application's state at every given instant. In the same way that a video is composed by a series of immutable frames, the application interaction is a succession of immutable models.

View A pure function which takes the model as input, and generates a new UI layout. This F# function, uses a renderer library such as React to declaratively build a UI [60].

Update A pure function that takes the current model and a new command as input, and outputs a new model.

In the Elmish library, the commands that trigger updates are called messages. A message represents, for example, the action of clicking a button. When a message is dispatched, it gets passed to the update function, which produces a new model. This new model is then rendered by the view function. Figure 4 represents this process schematically.

To better understand how this loop of operations works, we will analyse a simple application that keeps track of how many times a button is clicked (see figure 5). The model is an integer, which is initialised to zero:

```
type Model = { Count : int }
let init () = { Count = 0 } // Returns the initialised model.
```

We need to define what type of messages are expected. In this application, we require a single Increment message:

```
type Messages = | Increment
```

The view function needs to display the count and a button:

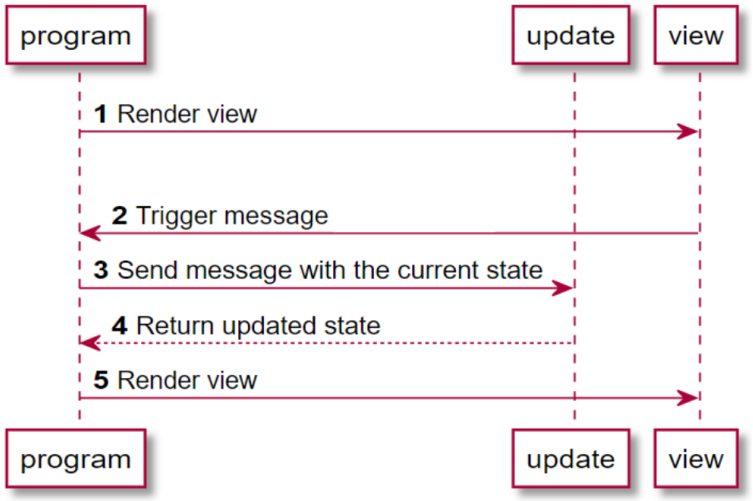

Figure 4: Model-View-Update interactions in the Elmish architecture. Image credits: [25].

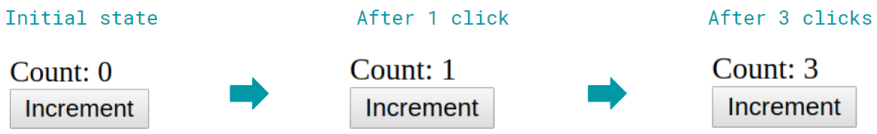

Figure 5: Simple Elmish application.

```
let view model dispatch =
    // The function takes as input:
    // - the current model.
    // - a dispatch function, which can deliver messages to trigger updates.
    // Creates an HTML div element containing some text and a button.
    div [] [
        text "Count: %d" model.Count
        button [ OnClick (fun _ -> dispatch Increment) ]
               [ text "Increment" ]
    ]
```

Finally, the update function handles incoming `Increment` messages:

```
let update message model =
    match message with
    | Increment ->
        // Creates and returns a new model with the incremented count.
        { Count = model.Count + 1 }
```

Upon launching the application, the UI gets rendered by calling `view` with the initial model (with `Count` set to zero). This is step 1 in figure 4. When the user clicks the "Increment" button, the `Increment` message

is dispatched (step 2). The Elmish library then triggers the `update` function (step 3), which produces the updated model (step 4). Lastly the new model gets displayed by calling the `view` function (step 5), and the application is ready for a new interaction.

It is interesting to note how the interface layout is generated in the view function. Elmish allows to define HTML and CSS elements directly from F#, which provides two major advantages. First of all, as every part of the F# code is type checked, the compiler ensures that the generated HTML code will not contain common mistakes, such as missing closing tags or invalid attributes. Secondly, the CSS itself can be defined and dynamically updated from F# itself. Traditionally, dynamically changing the CSS code would require the use of CSS global variables and the insertion of a series of DOM selectors and event handlers, which make the code more prone to bugs and less understandable (similarly to handling events in section 3.3.4). The use of dynamic CSS in Elmish applications will be further discussed in section 5.2.2.

3.5 Summary

This section presented a series of considerations necessary to understand the topics that will be discussed in the following sections. Subsection 3.1 concerned the analysis of what characteristics an ideal digital electronics teaching tool should have, such as being intuitive. Subsection 3.2 provided an high-level analysis of some of the currently available hardware design applications, determining that none was found to satisfy all the ideal criteria. Additionally, a series of technologies have been described in subsection 3.3, which converged into the Fable-Elmish-Electron technology stack described in section 3.4.

All the considerations made in this section will guide the definition of the requirements for the project (section 4), as well as impacting both the implementation (see 5) and the evaluation (see 8) sections.

4 Requirements Capture

In this section, a clear and comprehensive specification of the requirements of the project will be laid out. These requirements are the foundation for all the design choices that will be analysed in the subsequent sections.

As previously stated, the project concerns the creation of a hardware design platform to improve the digital electronics learning experience of first year university students. The application is primarily targeted to Imperial College EE students, but it is hoped that it will be adopted also by other teaching institutions or even individuals interested in learning electronics topics. This information is of crucial importance, as it is unrealistic to plan the creation of a fully fledged industrial hardware design application in less than two months.

According to Imperial College EE Digital Electronics and Computer Architecture (DECA) course syllabus [21], first year students are expected to learn about combinatorial and synchronous logic, as well as being able to use block diagrams to design computer systems containing RAMs, ROMs, registers, adders, multiplexers and state machines. To fulfil these requirements, the application must have the following essential features:

E1 Include a graphical interface which allows users to design synchronous digital circuits containing both combinatorial logic and clocked components.

E2 Include a catalogue containing a set of builtin digital logic components, such as gates, flip-flops, registers, ROMs, RAMs, multiplexers and demultiplexers. The user can use such catalogue to add components to their designs.

E3 Allow the user to define their own components and reuse them in successive designs. For example, the user could create an adder from basic combinatorial components and use it to build an ALU.

E4 Allow the user to package multiple wires into buses.

E5 Allow the user to insert probes and inputs into the circuit to test and simulate the logic.

E6 Provide an interface that allows users to simulate the circuit behaviour with user-specified inputs. The user should also have the ability to step through the simulation one clock cycle at a time.

E7 Include an analysis tool that validates the circuit diagram and reports errors.

E8 Give the possibility to save and load designs, and handle design dependencies (for example using a previously created adder to build an ALU).

E9 Be actively maintained. This is necessary as every large software will lack some useful features or contain some bugs, and maintainers can act upon them.

These functionalities can be found in other programs like the aforementioned Quartus. The application ought to distinguish itself via the following desirable features:

D1 Having a clear, responsive and intuitive user interface. A survey conducted on a group of university students demonstrated that 88% of them get confused when using a complex system like Quartus (details provided in section 8.2). 75% of the surveyed students agreed that their learning experience

would have benefited from having used a simpler application. In fact, Quartus offers an abundance of advanced features which are well beyond the laboratory scope and navigating through them adds unnecessary complexity to the learning process.

D2 Providing helpful circuit design functionalities such as wires auto-routing and automatic bus width inference.

D3 Providing helpful and thorough feedback for errors, which carefully explains the issues that may be present in the design. Additionally, errors ought to be visualised directly on the circuit diagram, so to maximise the amount of guidance given to the users. Applications that are targeted to experienced hardware engineers often tend to assume a deep understanding of electronics concepts (which is usually not the case for students at the beginning of their university education) and present errors and warnings in a less explanatory fashion.

D4 Being relatively easy to extend, in particular by third year EE Imperial College students. This way, the application can be easily adapted to reflect future changes in the module syllabus.

D5 Have a good documentation that allows maintainers to quickly familiarise with the main ideas and design choices of the application.

D6 Being open-source.

D7 Being easy to install and run on all main platforms (Windows, Linux and MacOS). This is not the case for Quartus, for example, which requires some level of expertise to be installed on Linux [78], and cannot be run natively on MacOS (requires installing Windows on a virtual machine [1, 64]). All students can therefore effortlessly install the software on their personal devices, hence being able to use it without having to physically access the EE laboratory or connecting to EE servers via SSH.

This set of requirements and desirable features provide a comprehensive framework to guide both the implementation and the evaluation of the application and its alternatives. Desirable feature D2, in particular, leaves space for countless extension possibilities, some of which will be analysed in section 9.1.

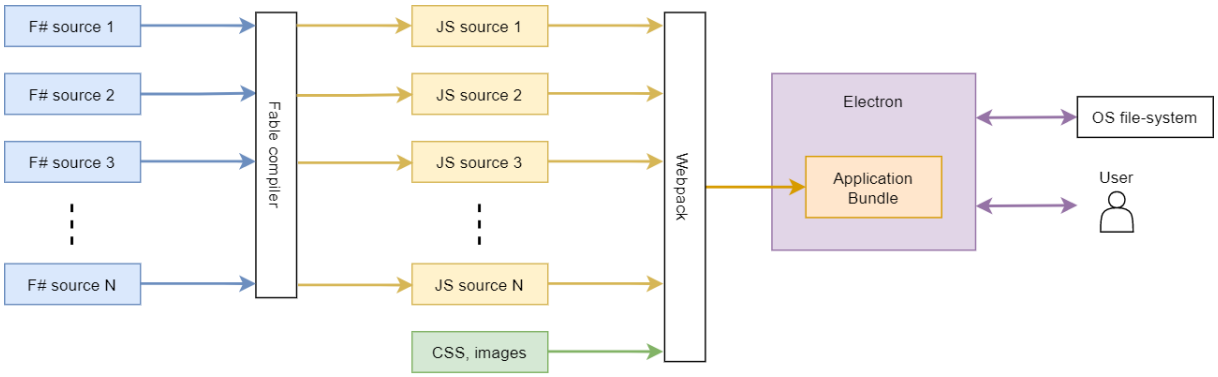

Figure 6: Compilation process for F# code to be run on Electron.

5 Implementation

The implementation of DEflow presented numerous challenges, both from an algorithmic and a software engineering point of view. In this section I will discuss the most interesting aspects of this process.

First, the high level overview and main design choices of the application will be presented in subsection 5.1. Then, subsections 5.2 and 5.3 will discuss how the application interface has been created and what extensions had to be made to the chosen circuit diagram drawing library to tailor it to the digital electronics domain. Finally, the following subsections will focus on the algorithmic part of the application, such as diagram analyses and simulation.

5.1 Analysis and Design

Since DEflow is a large application (more than eight thousand lines of code), it is important to understand how the code has been designed to manage such complexity. This subsection will first present how F# code can be compiled into a cross-platform application, which will allow us to understand the reasons of DEflow codebase architecture.

As per requirements, the system must be relatively easy to maintain and extend. For these reasons, it was decided to develop the application using the F# functional programming language, whose benefits are covered in the section 3.3.3. The process of compiling F# code into a cross-platform desktop application requires the use of several technologies (see figure 6). First of all, the Fable compiler compiles F# code to JavaScript. All the produced JavaScript modules (and other static assets such as CSS files) are bundled together using Webpack [76]. This application bundle is then packaged with Electron, which produces a platform specific executable. As explained in section 3.3.2, Electron uses Node.js and Chromium to run and render the application. In particular, the use of Node.js rather than the Chromium JavaScript engine allows the application to access OS level API such as the file-system.

The application code structure is presented in figure 7. One of the most crucial design choices in DEflow, is the separation of the user interface logic core from the core application logic and the testing logic.

The core application logic comprises the algorithms for analysing, simulating and inferring the widths of wires in the circuit diagrams. The core logic is entirely written in F# and it is completely decoupled from the Fable-Elmish-Electron technology stack (described in section 3.4). This allows it to be run and tested

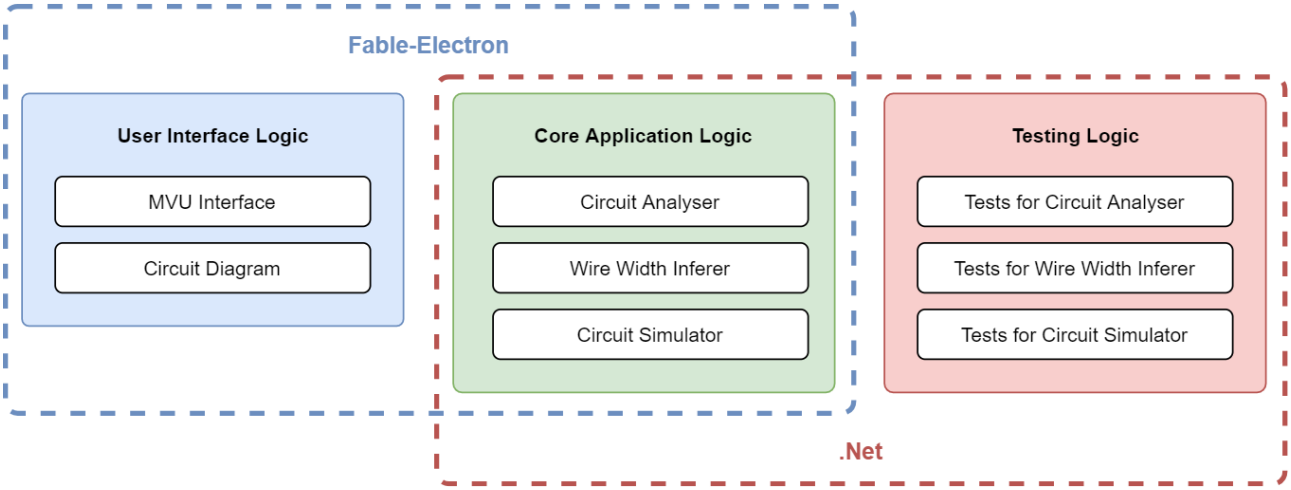

Figure 7: High level overview of the code structure.

Asset	Type	Description	Lines of code
`src/Common/*`	F# project	Collection of types and utilities used by all other F# parts of the application.	137
`src/WidthInferer/*`	F# project	Logic to infer the width of wires in a circuit diagram.	451
`src/Simulator/*`	F# project	Logic to analyse and simulate the behaviour of a circuit diagram.	1997
`src/Tests/*`	F# project	Tests for the logic in `WidthInferer` and `Simulator`.	2106
`src/Renderer/*`	F# project	User interface logic, draw2d F# wrapper and utilities to interact with the file-system via Node.js API.	2601
`src/Main/*`	F# project	Boilerplate code to configure and start the Electron application.	78
`app/public/lib/draw2d_extensions/*`	JS files	Extension of the draw2d JavaScript library to support digital components and connections.	1023
`app/*`	Web assets	Static web assets including stylesheet, icon and application markup.	N/A

Table 3: Structure of the code in the repository. The colour coding matches the one in figure 7. In the interest of space, only the most important folders are listed. The number of lines of code reported is accurate as of 18/06/2020.

under the .NET framework.

The testing logic provides a comprehensive set of tests for the core application logic. Like the core application logic, it is completely written in F#. In order for the testing infrastructure to be versatile and fast, it relies on the Expecto testing library [26], which cannot be compiled by Fable. Therefore, the testing logic can only be run under the .NET framework. The testing infrastructure will be further investigated in section 6.

The user interface (UI) logic is responsible for allowing the user to interact with the core application logic. It is mainly written in F#, which interoperates with JavaScript via modules provided by the Fable compiler. Because of this interaction with JavaScript, this logic cannot be run under the .NET framework. The user interface logic adopts a Model-View-Update (MVU) architecture, making use of the F# Elmish library. The benefits of using a Functional Reactive Programming architecture such as MVU, have been thoroughly discussed in sections 3.3.4 and 3.4. Additionally, the UI logic relies on the Fulma style library [33], which is an F# port of the famous Bulma CSS library [7]. Relying on a style library partially lifts the burden of having to worry about creating an appealing and consistent interface. In fact, Fulma contains a lot of pre-styled elements such as buttons and menus, which make the development much faster. The UI logic will be further discussed in section 5.2.

The separation between core, testing and interface logic has two major benefits. Firstly, testing is made much easier as it is possible to test the core application logic directly under .NET, using powerful libraries such as Expecto [26]. Secondly, the separation reflects the conceptual disrelation of the various parts of the application. Different modules communicate among each other via a set of well-defined high level API, hiding the complex logic that lies underneath them. This, in turn, helps to better understand the application itself and guarantees that changes in a module will not break others (as long as the module API is not changed).

An alternative to this modularised design is a monolithic architecture. Such architecture tends to be easier to set up, as no explicit effort is made to separate different modules, but it is a poor choice on the long term since it provides none of the guarantees above mentioned. A monolithic architecture would make it impossible to run and test part of the code under .NET, as some libraries specifically designed to work with Fable will not work in Microsoft's framework. Furthermore, it would not be easy to guarantee that changes in a part of the codebase will not affect other parts, making it difficult to change and improve the application.

This conceptual separation is reflected in the code structure, which is presented in table 3. As it is possible to see from the table, the application is mostly implemented in F#, with some JavaScript being necessary to extend the drawing library (more on this in section 5.3).

In the next subsections, the most interesting and challenging aspects of the implementation of DEflow modules will be thoroughly analysed.

5.2 MVU User Interface

As stated in section 5.1, Functional Reactive Programming (FRP) was chosen to develop a responsive and maintainable UI. The advantages of a declarative approach for UIs have also been thoroughly described section 3.3.4. FRP is possible in F# thanks to the Elmish library, which implements the MVU architecture described in section 3.4. Elmish uses the React JavaScript framework [60] to render the user interface. React is a framework developed and maintained by Facebook, which allows to handle the HTML DOM in a

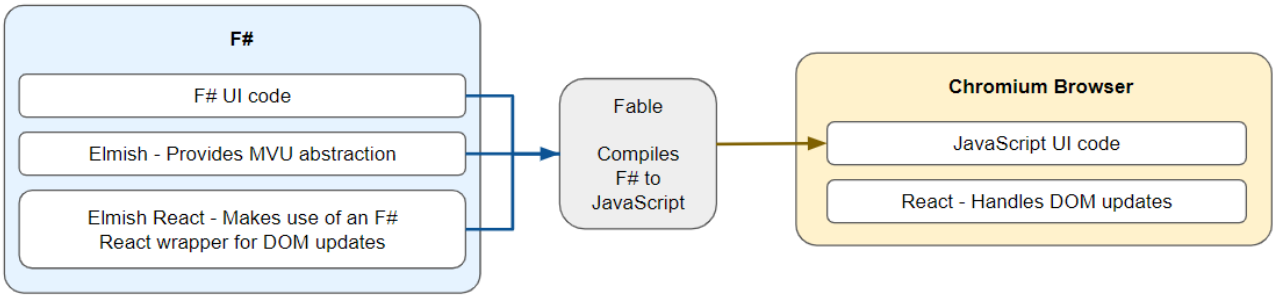

Figure 8: Compilation process of the user interface code.

declarative fashion. Figure 8 shows how these various technologies interact together in DEflow.

DEflow, in accord with the Model-View-Update (MVU) architecture, has a *model* which keeps the state of almost the entire application. This gets transformed into an actual interface by the various *view* functions. There exist, for example, a view for the catalogue from where the user selects the components they want to add to the circuit, one for the editor to change the content of ROM and RAM memories, and so on.

Each view exclusively uses the information in their relevant part of the model in order to build the interface. Therefore, the different views are isolated from each other, and changing something in one piece of the interface is guaranteed to not have any consequence on the rest. This, in turn, greatly simplifies the development process and increases the confidence in making changes.

Finally, views may contain interactive elements such as buttons or text-boxes. Upon user interaction, these elements can dispatch messages that will *update* the application model, which will be re-rendered via the view functions. If done naively, this re-rendering process is expensive. Luckily, React only re-renders DOM elements that actually changed (which usually form a very small part of the whole interface), massively improving the MVU performance.

To better understand the MVU loop in DEflow, consider the example where a user is populating a text-box with a number that is supposed to be two bits wide. Every time the user updates the content of the text-box a message is dispatched and the application model is correspondingly updated. In case the user is inserting an invalid number (for example a number that requires more than two bits to be represented, like 5), the *notification view* will generate a popup to notify the user of the error. As soon as they insert a valid number the notification view will remove such popup. Therefore, the MVU loop allows DEflow to have an extremely responsive UI, which can immediately provide feedback to the users. This type of immediate error feedback is also employed when the user performs other invalid actions, to promptly guide them toward the resolution of the issues.

5.2.1 Stateful Library in a MVU User Interface

The MVU architecture works extremely well if the whole application state can be captured in the *model* data structure. Unfortunately, this is not always possible for complex applications that rely on external libraries which were not designed with a functional architecture in mind.

As mentioned in the requirements capture (requirement E1 in section 4), the application necessitates a graphical interface which allows users to design digital circuits. A considerable amount of effort would be

	Features supported out of the box						Other characteristics				
Library	Draw lines	Draw polygons	Create connections	Auto route connections	Extract diagram state	Load diagram state	Is open-source	Is extensible	Is maintained	Size (MB)	GitHub stars
InteractiveSVG [40]	✓	✗	✗	✗	✗	✗	✓	✓	✓	0.05	17
Raphael [58]	✓	✓	✗	✗	✗	✗	✓	✓	✓	1.1	10.6k
Fabric [30]	✓	✓	✗	✗	✗	✗	✓	✓	✓	2.4	15.6k
jsPlumb [45]	✓	✓	✓	✓	✓	✓	✗	-	✓	-	-
draw2d [18]	✓	✓	✓	✓	✓	✓	✓	✓	✓	6	367

Table 4: Comparison of some common JavaScript 2d graphics libraries.

required to design and implement a graphic library from scratch, therefore I decided to rely on an existing library. Table 4 shows a comparison of several 2d graphics JavaScript libraries. Given its set of features and its extremely helpful documentation [17], draw2d was selected for this project.

In order to create an interactive diagram component, one must instantiate the diagram object provided by the chosen library and attach it to an HTML element rendered in the page. The library then modifies the HTML element and effectively transforms it into the diagram UI. The diagram component is therefore a JavaScript object that maintains its own state, which is mutable and separate from the MVU model. This state contains information about what figures are displayed in the diagram and how they are connected. Having a state which is updated independently from the MVU model breaks the MVU semantic of the application itself. Nonetheless, there exist two main approaches that allow the usage of stateful libraries without having to renounce to the MVU architecture:

- The first and most natural solution would be to reimplement the diagram component and expose its state so it can be stored in the MVU model. This way, the diagram component would be no different from other stateless components: every change in the diagram would trigger a message, update the model and re-render the new view. Although this approach is nice in theory, it is really expensive in practice in terms of both development time and application performance. In fact, every update in the diagram would have to pass through an MVU iteration, possibly triggering costly re-rendering operations. For example, the process of dragging an element in the diagram would trigger hundreds of re-renders of a complex HTML object, degrading performance sensibly.

  This is a known problem of declarative approaches to user interfaces. A clear example of this issue comes from the Atom code editor, which was rewritten in 2015 without the declarative abstraction introduced prior in 2014 [50], because the responsiveness seemed to suffer [38].

- The second solution is to allow the diagram component to maintain its own state and manipulate its own HTML elements, while creating a wrapper to interact with it in a controlled manner [79, 39]. Such

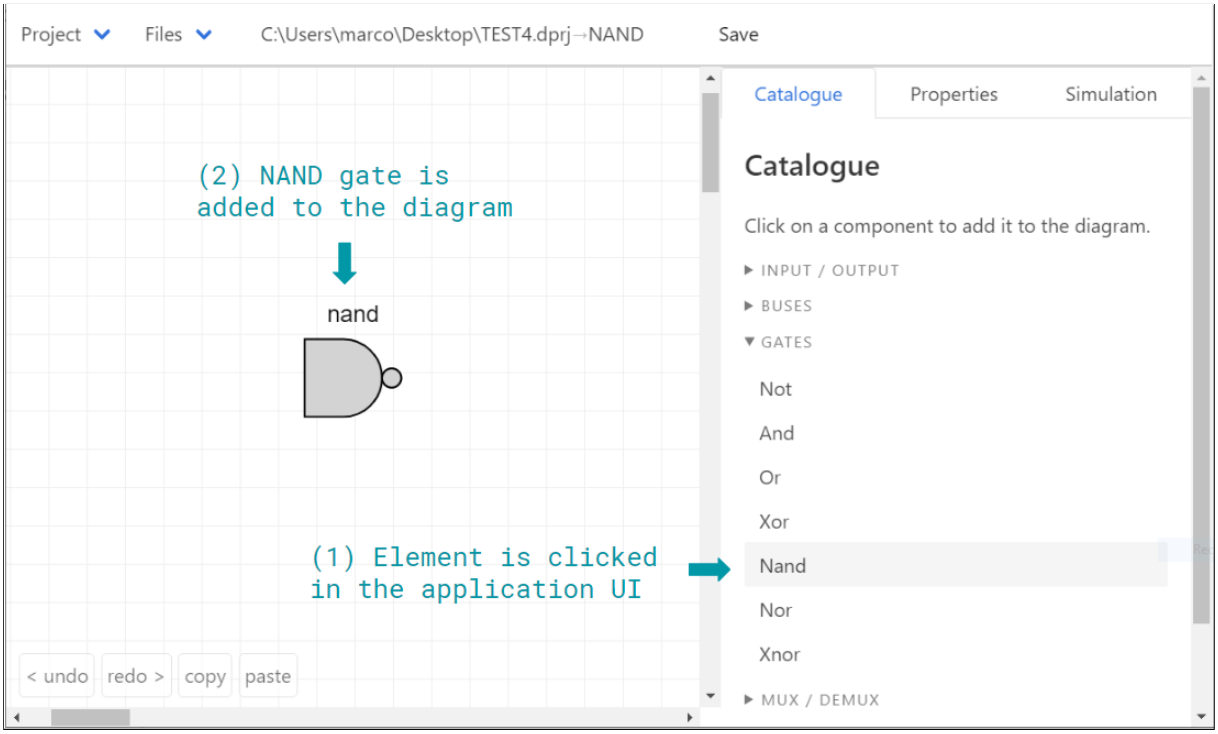

Figure 9: Clicking a menu element in the main application UI triggers a change in the diagram.

a wrapper is much quicker to implement, but it has to be handled carefully in order to preserve the functional semantic of the application as much as possible.

In this approach, the state of the application remains split among two independent data structures, agnostic of each other. In other words, the MVU application state is rendered and updated independently from the diagram state, and vice versa. This is optimal for performance, as the diagram component works the way it was designed to, without any extra MVU operation. The wrapper has the role of mediating the interaction between the MVU application and the diagram component. This interaction can happen in two directions:

The MVU application changes the diagram state. For example, if the user clicks a button to add a figure to the diagram (see figure 9), a method of the wrapper is called. This instructs the diagram to add the specified component, while the MVU model remains unchanged.

The diagram component changes the state of the MVU application. This may happen if the user selects a component in the diagram and its properties get displayed in the main application UI (see figure 10). In this scenario, the diagram component will dispatch an MVU message with the information that needs to be displayed. This message will be handled in the update function, which will produce an updated MVU model. The new model will then be rendered via the relevant view functions. In this scenario, the wrapper has the role to provide the diagram component with a function to dispatch an MVU message.

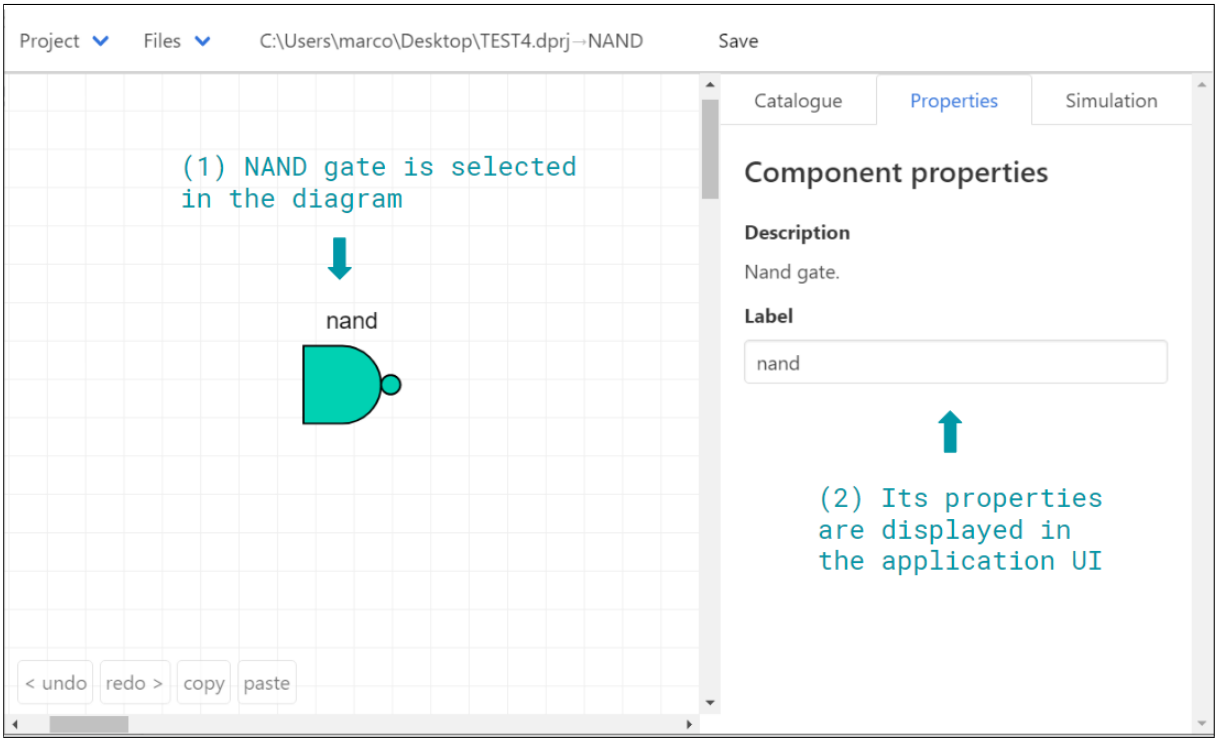

Figure 10: Selecting a component in the diagram dispatches an MVU message and updates the main application UI.

Given its advantages, the wrapper approach has been chosen for DEflow. Clearly, this choice imposes a tradeoff between functional-style and efficiency. Having a small, carefully wrapped non-functional part of the application logic is unideal, but it is a necessary tradeoff that arises from the need of efficiently using stateful libraries.

5.2.2 User Interface Styling

As mentioned in section 5.1, DEflow relies on the Fulma styling library to generate cohesive, appealing and informative interfaces. Fulma is an F# port of Bulma, which is a modern, modular and open-source CSS library. Fulma provides a vast catalogue of pre-styled elements such as buttons and menus, which have been created by UI experts. Therefore, using such a library can highly increase both productivity and deliverable quality.

Additionally, the developer can also decide to override the styles provided by the library. This can be done by defining CSS properties directly in the F# code. Since CSS properties are just normal F# data, it therefore becomes very easy to dynamically change the style of the elements in the interface. For example, when a user modifies a diagram, the circuit diagram should be the most prominent component of the UI, and hence it should cover a large area of user interface (see figure 11). On the other hand, when running a simulation, the simulation menu should cover a large amount of the interface, as users will have numerous interactions with it (see figure 12). Dynamically updating the CSS properties of the various elements on the page allows the developer to introduce such a behaviour.

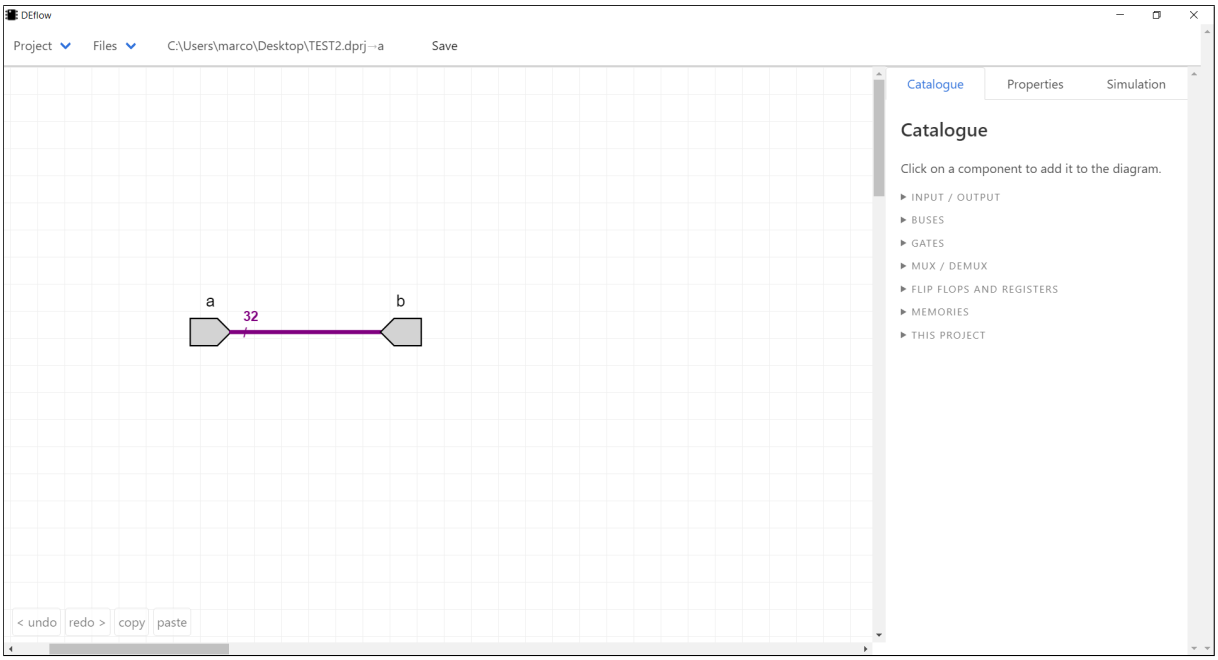

Figure 11: The diagram occupies a large area of the interface as the user is likely to have numerous interactions with it during the circuit design phase.

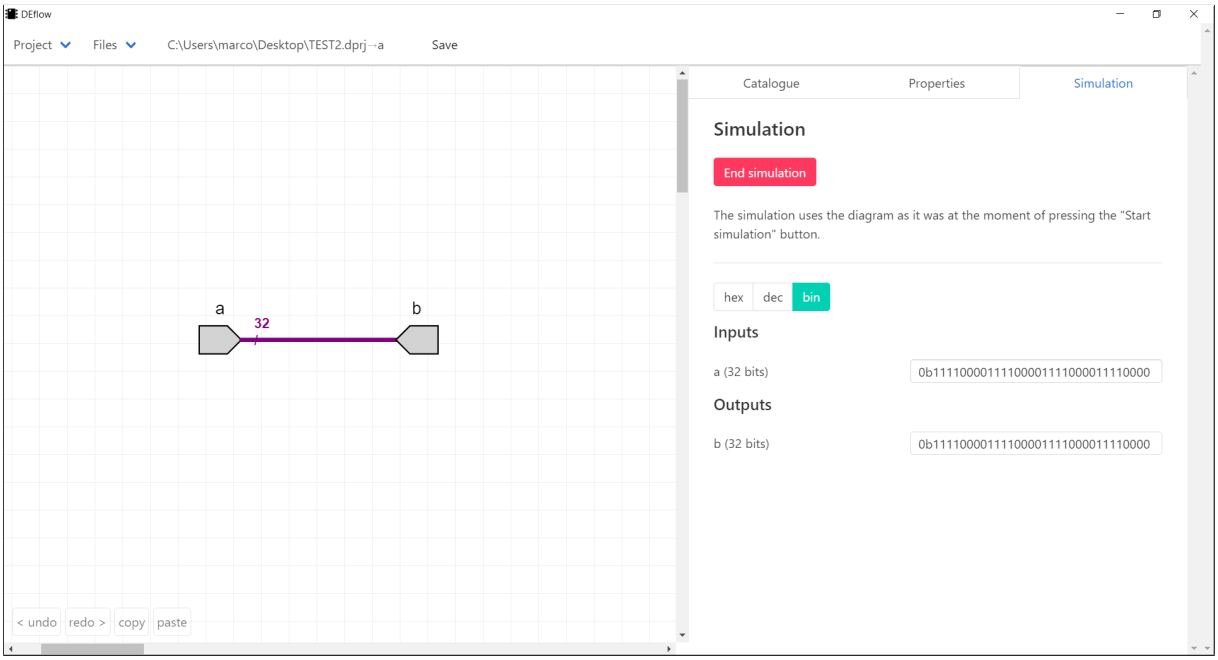

Figure 12: The simulation menu is enlarged to allow the user to interact with it better. This way , large binary inputs and outputs can be viewed.

5.3 Extensions to the Drawing Library

In order for the application to be intuitive for the users, it must make use of standard digital electronic notations such as common gate shapes [81], and thicker labeled connections to represent buses. The draw2d library provides a vast set of functionalities, but unfortunately support for digital electronics components is not among those. This subsection will present how this issue has been overcome.

In order to provide the necessary visualisations, draw2d has been extended with several modules. Such extensions can be found in the `app/public/lib/draw2d_extensions` folder. A series of JavaScript classes have been defined which allow users to visualise gates and other common electronics components with their standard shapes. The choice of using the 'old-fashioned' gate figures has been made as those are the ones currently used in the DECA laboratories. Nonetheless, this can be changed with a minimal amount of effort in case of necessity.

The extension classes derive the builtin `draw2d.SVGFigure` class in order to benefit from all of the functionalities of the library. One extension also introduces the support for the dynamic creation of diagram components, given a list of component inputs and outputs. This extension is necessary for the creation of hierarchical components (described in section 5.7). Additionally, a series of functions to draw and label wires have been added.

While developing such extensions, two bugs in the draw2d library have been found and reported to draw2d developers [15, 16].

5.4 Serialisation and Deserialisation of the Circuit Diagram

A very important feature for the application is to give users the ability to save and load diagram designs (requirement E8 from section 4). To do so, it is necessary to obtain a textual representation of a circuit diagram, which can be written and read from a file. The draw2d library offers functions to serialise the diagram object into a string and to subsequently reload such serialised state into a diagram. Unfortunately these functions have several characteristics that make them not adapt for the application:

- The information that gets handled by these functions is just a subset of the entire diagram state. For example, such serialisation would discard information about components' labels.
- The serialization output tends to be considerably memory intensive, even for small diagrams. For example, the serialisation of the circuit in figure 13 produces a 12 Kilobytes output.
- The output produced by the draw2d serialiser requires the specific draw2d deserialiser logic to be parsed properly. Moreover, the deserialiser logic automatically populates the diagram with the deserialised content. This is highly undesirable as DEflow often necessitates to access a circuit state directly from the file without loading it into an actual circuit diagram.

To compensate for the shortcomings of this library functionality, I decided to implement my own serialisation and deserialisation functions. The custom serialiser first transforms the JavaScript diagram object into an F# data structure. This data structure is then serialised to a Json object and dumped to a file. The custom deserializer reads the desired file and parses its content back into the F# data structure. This data structure can then be visualised in the circuit diagram, or used for other means. This approach is superior

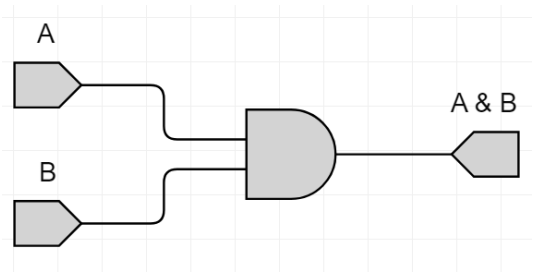

Figure 13: Simple circuit with an AND gate connected to two inputs and one output.

to using the draw2d functions since it stores all the necessary information in an easily parsable format, while achieving a much smaller file size. In fact, the circuit in figure 13 only requires 2 Kilobytes of memory, just 17% of the size produced by the draw2d functions.

The logic that transforms the JavaScript diagram object into an F# data structure will also be used to provide the input to the diagram analysis and simulation algorithms, described in subsections 5.5 and 5.6.

5.5 Diagram Analyses

As per specifications, one of the crucial features of the application is its ability to detect a variety of errors that the user may introduce (requirement E7 from section 4). In order to do so, a set of analysis is run on the circuit diagram. Most of these analyses will be presented in this subsection. In particular, high relevance is given to the algorithm for the inference of the width of the wires in a circuit diagram, as it is of elevated algorithmic interest.

5.5.1 Wires Width Inference

In order to facilitate the design of complex logic circuits, the application allows the user to package multiple wires into buses. A bus can carry an arbitrary number of bits. The number of bits carried determines the *width* of such a bus. This subsection will analyse the reasons why inferring such information is important, as well as providing the algorithm required to do so.

Inferring the width of the wires in a diagram is necessary for two reasons. First, it is needed to ensure that the user is not creating invalid designs. For example, an inattentive user may connect the output of a 3 bits register to an AND gate input. This is, of course, invalid and the user has to be warned of such a mistake (figure 14). Secondly, inferring wire widths allows the application to label wires with their corresponding widths, making it much simpler to develop complex digital circuits (figure 15).

Alternatively, the application could ask the user to specify the width of each wire they insert. Such a decision would negatively impact the usability as users would have to spend time providing information that could be automatically deducted. Furthermore, modifying a design may become cumbersome since changing a component or a connection could lead to the need of manually updating several other bus widths in the circuit diagram.

This issue is similar to the problem of type inference in some programming languages, including F#. In fact, in order to favour usability, the compiler automatically deduces the types of the variables being used, so

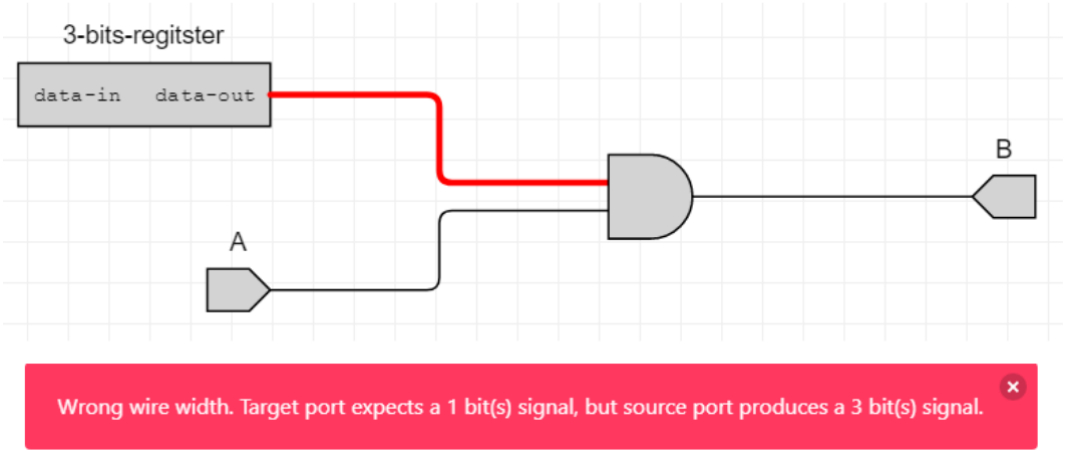

Figure 14: Error notification shown upon unsuccessful wires width inference.

the developer does not have to explicitly annotate them. Unfortunately, inference adds an extra complexity to the design of a compiler or, in our case, a hardware design application.

Inferring the width of wires is trivial in some cases (for example, a wire connected to a two bits register will carry two bits), but complex in others. Consider the example in figure 15. There, four wires with different widths are merged into a single ten bits wide bus. Determining the width of the final wire is only possible by recursively inferring the width of the intermediate wires. The width inference can be performed with the following graph traversal algorithm, which can be seen as an enhanced depth first search [91]:

```
Variables:

wiresWidth: a map that keeps track of the inferred width of each wire in the diagram.
graph: a graph data structure representing the circuit diagram. The children of a graph node are the nodes
       connected to its outgoing wires.

Algorithm:

1.  Initialise each connection in wiresWidth to "Not-Inferred"
2.  For each input node in the graph:
3.    Extract the widths of all the incoming wires (if any) for the current node, using the wiresWidth map
4.    Are the incoming wires width valid?
5.      Yes: Proceed
6.      No: Return error and terminate algorithm
7.    Does the current node have enough information to determine the width of its outgoing wires?
8.      Yes: Set the widths of the outgoing wires in the wiresWidth map
9.           For each child node, if the wire leading to it is "Not-Inferred", repeat the algorithm
             from step 3, using such a child as current node
10.     No: Cannot proceed with recursion, return
```

The algorithm starts from the input nodes, and recursively tries to populate a map that keeps track of the width of each wire in the diagram. It is possible that the user creates invalid diagrams, for example by connecting the output of a 3 bits register to a single-bit AND gate input (figure 14). Such a problem will

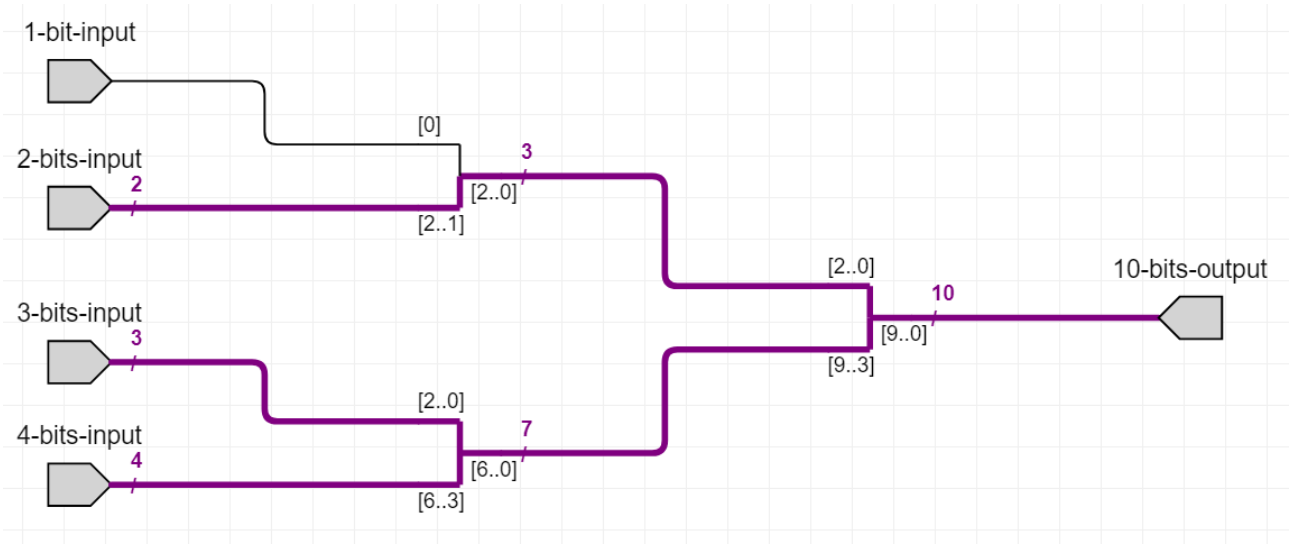

Figure 15: Circuit that requires non-trivial wires width inference.

be spotted by the algorithm with the check at step 4. In fact, the algorithm will ensure that an AND gate receives only two single-bit inputs, otherwise it will return an appropriate error message and terminate.

In some cases a node may not have enough information to determine the width of its outgoing wires, hence the recursion branch will not be explored further. This behaviour is reflected in step 10 of the algorithm. Such a scenario may arise, for example, if the user only partially connects a 'MergeWires' node. A 'MergeWires' node is a diagram node that, taken two wires of width `N` and `M` produces an output wire with width `N+M`. If one or both inputs are missing, it is impossible for the algorithm to determine the width of the outgoing wire. There are three 'MergeWires' components in the diagram in figure 15.

The proposed algorithm is optimal in terms of time complexity. In fact, its asymptotic time complexity is $O(N + E) = O(E)$, where N is the number of nodes and E is the number of edges (in essence, wires) in the graph. This property is guaranteed by the check at step 9 of the algorithm, which ensures an edge is never 'explored' twice.

Nonetheless, the algorithm could be further optimized by only re-inferring the widths of the wires affected by the user changes. This can be achieved by passing to the algorithm the `wiresWidth` map previously obtained, together with the information specifying which connection has been changed. The algorithm is then started from the node at the source of such a connection, and it will recalculate only the necessary widths. Note that extra care has to be taken at step 9: in fact, a wire with an old inferred width has to be re-explored, whilst wires with widths inferred during the current iteration have to be skipped. These two scenarios can be distinguished through the use of a flag that keeps track of whether a wire width has been recently re-inferred.

This optimization has not been implemented as the performance of the algorithm was measured to already be significantly below the threshold of human perception (see section 8.3), so priority has been given to other DEflow features. Nonetheless, it could be valuable to implement it in the future as it requires only minor changes to the current algorithm.

It is interesting to further consider how the algorithm avoids getting stuck in loops that may be present

in the diagram logic. This is possible because an edge width is inferred at most once, thanks to the check performed at step 9 of the pseudocode.

The width inference logic is implemented in the `WidthInferer` module. This module has a very simple API: a single function called `inferConnectionsWidth` which receives in input a circuit diagram and returns a map where each connection is associated to its inferred width, or an error if there is a flaw in the diagram that made the inference process impossible.

The UI module calls the inference function with the current diagram state and handles its result. In case the inference process was successful, the buses in the diagram are highlighted and their width is annotated on top of them (see figure 15). Otherwise, errors get visually presented both by highlighting the faulty wires and with a notification popup explaining the issue (see figure 14). This error reporting method turned out to be extremely effective, with 100% of the users reporting that they agreed that such errors "guided them well toward fixing the problems in the design" (details in section 8.1.2).

5.5.2 Errors Detection

Once the user created their design they can decide to run a set of in depth analysis in search for potential logic flaws. DEflow provides a series of diagram validations, including:

- Ensure that every component is fully connected. In other words, ensure that no component in the diagram has floating inputs or outputs.
- Ensure that every net node is driven by precisely one signal. In fact, DEflow does not support tri-state logic, as it is not part of the syllabus of first year EE Digital Electronics teaching.
- Ensure that input and output components have unique labels. This allows users to have a non-ambiguous interface during the simulation (see section 5.6).
- Ensure that all wire widths can be successfully inferred. This is achieved by running the width inference algorithm previously described, with the additional check that all wires must have an inferred width.

Two further validations will be described in sections 5.7.2 and 5.7.3, as they are related to the concept of hierarchical components presented in section 5.7.

If any of the analysis discovers a flaw in the circuit, the issue is reported via an error message and by highlighting the affected components and wires in the diagram.

These analyses are conceptually independent from each other, therefore it is extremely simple to add further validations or change the existing ones. Each analysis function have the following interface: it accepts in input a representation of the circuit state and optionally returns an error. All the analysis algorithms have linear time complexity in the number of components or wires (depending on the analysis), which is optimal.

5.6 Diagram Simulation

One of the core DEflow features is the ability to simulate digital circuits (requirements E5 and E6 from section 4). Crucially, this allows users to experiment with their designs and learn the related Digital Electronics concepts. This subsection will analyse the algorithms necessary to perform both combinatorial and

synchronous circuit simulations. Additionally, subsection 5.6.2 will show how these algorithms are exposed to the rest of the codebase.

In order to simulate the logic they implemented, a user can inject signals in the circuit using *input* components. The user can also attach *output* components, whose role is to accept a signal without propagating it further. The signal reaching an output component can be then visualised in the simulation interface. Output components can therefore be used to probe the signal on specific wires of the circuit.

The process of simulating a circuit has the following steps:

1. The state of the diagram has to be extracted and transformed into an F# data structure (like it was described in section 5.4).
2. The various analyses described in section 5.5.2 are run to guarantee the validity of the design.
3. The extracted and validated state is transformed into a simulation graph.
4. A further set of validations is run on the simulation graph (described in sections 5.7.2 and 5.7.3).
5. The user feeds input signals in the simulation graph via input components and visualises the corresponding changes via output components.

A user interface takes care of facilitating the task of feeding inputs into the simulation graph and visualising its outputs. It is important to stress that such user interface is completely decoupled from the core simulation logic itself, and can be independently modified and improved. In fact, the simulation logic itself is contained the `Simulator` F# project, which can be run and tested under the .NET framework, without any user interface (details about code modularisation can be found in section 5.1).

We already covered the ideas behind diagram state extraction (section 5.4) and diagram validation (section 5.5). The next step of the simulation process is to build a *simulation graph*. A simulation graph is a graph data structure composed by a series of nodes, each having a *reducer* function and a set of *children* nodes. The reducer is a function specific to each type of circuit component, whose purpose is to take the inputs of a node (the signals on the incoming wires) and transform them into the outputs of that node (the signals that will be put on the outgoing wires). For example, the reducer for an AND gate will take two single-bit inputs and produce a single-bit output. The children of a node are the set of nodes that are connected to its outgoing wires.

Creating a simulation graph is of great help for the simulation process. In fact, the original diagram state is a flat data structure which simply lists all the components and connections in the diagram. Such flat data structure does not represent the flow of information in a digital circuit. For example, in order to look up what components are connected to a given component A, it is necessary to search among all the connections the ones that have component A as either source or destination. As this operation has to be repeated many times during the simulation, this approach becomes prohibitively expensive for complex diagrams. More importantly, this cost is completely avoidable by transforming the flat data structure into a simulation graph.

With such infrastructure in place, the process of feeding an input signal into the simulation graph can be performed with a recursive algorithm. This algorithm is an enhanced depth first search:

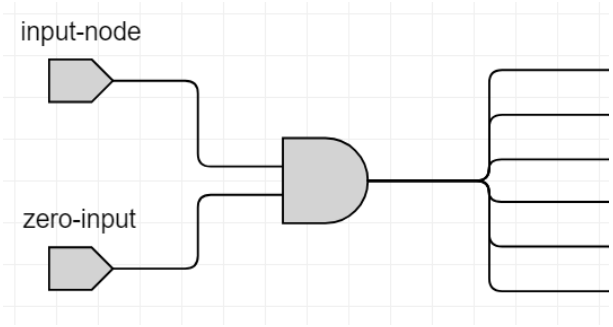

Figure 16: Simple AND gate with grounded input and many outgoing wires.

```
Variables:

graph: the simulation graph data structure representing the circuit being simulated

Algorithm:

1. Start from the selected input node in the graph
2. Obtain the current input signals for the node
3. Use the node reducer to calculate the new output signals. Was any output produced?
4.   Yes: For each child of the node, update its input signals and repeat the process from step 2
5.   No: Return
```

Note that a component may not produce any output (step 5). This is the case for synchronous components such as D-flip-flops. These components, in fact, only produce outputs on a clock tick (more on this in section 5.6.1). In this scenario, the inputs of the children nodes remain unchanged. The fact that only combinatorial components produce new outputs, together with the assumption that the graph contains no combinatorial loops (ensured by one of the analysis previously performed) guarantees the termination property of the algorithm. This can be intuitively seen by visualising the simulation graph as a tree, where the root node is the input node used to feed the new user input, and the leafs are the output nodes. Since there are no combinatorial cycles and synchronous components do not generate any output, the propagation of the new input will flow unidirectionally towards the leafs.

The algorithm has time complexity $O(N + E) = O(E)$, where N is the number of nodes and E is the number of edges in the simulation graph. This directly follows from the no-combinatorial-cycles assumption, together with step 4 of the algorithm: we may explore each graph node at most once, and for each node we may explore each of its outgoing edges at most once.

The algorithm has been further optimised by propagating the outputs of a reducer only if they changed. Consider for example the diagram in figure 16. Assume the signal produced by the input node `zero-input` consists of a single-bit permanently set to zero. This implies that, no matter what signal is generated by `input-node`, the output of the AND gate will always be a single-bit signal set to zero. This means that all AND gate's outgoing connections will never be 'explored' as a result of a change in `input-node`. This optimization makes the algorithm optimal in the sense that no extra work is performed to try to update signals that will not change.

5.6.1 Synchronous Components

Particular care must be taken when handling synchronous components. Such components, in fact, do not produce a new output every time one of their input changes, but only on a positive or negative edge of a clock signal.

Like most FPGAs [32], DEflow adopts the concept of global clock. A global clock is a unique clock that can be accessed by all components in a circuit. Furthermore, in DEflow, all synchronous components are implicitly connected to the global clock. The biggest advantage of this decision is that it greatly simplifies the simulation logic, since it introduces the notion of *discrete time*. In other words, the simulation time only advances discretely at every clock tick, when each of the synchronous components updates their outputs. This allows to eliminate the need of modelling signal propagation delays, which would have otherwise been incorporated in the logic. Since DEflow is mainly targeted to Imperial College EE first year students, and such timing issues are not in first year syllabus, it was decided to avoid this extra complication and focus efforts elsewhere. Moreover, such concepts are not easy to familiarise with, and would probably end up confusing students who approach digital electronics concepts for the first time.

On the other hand, having a global clock implicitly connected to all synchronous components blocks the user from designing valid and relatively simple circuits such as ripple counters. Ripple counters are covered in the first year EE DECA course, even though they are not present in the laboratory syllabus.

It is also arguable that first year students should explicitly handle and connect clock signals, instead of having them implicitly connected by the application. This decision has no impact on the simulation logic, as it would not change the fact that each synchronous component is connected to the global clock. Hence, this is implementable by introducing an explicit 'global clock generator' component that can be added to the diagram and connected to the clock ports of the synchronous components. Such component and its connections can be then easily removed from the diagram state that gets passed to the simulation logic, which will assume each synchronous component is implicitly connected to the global clock. In the interest of time, I decided to focus on more essential features.

It is possible to simulate a global clock tick with the following algorithm:

```
Variables:

graph[t]: the simulation graph after t clock ticks

Algorithm:

1. Take a snapshot of the state of the simulation graph at the end of previous time step: graph[t-1]
2. For each synchronous node in the simulation graph:
3.   Feed the node reducer with the inputs as they were in graph[t-1] and obtain its new outputs
4.   Feed the produced outputs into the combinatorial logic as non-clock-tick inputs (using the simulation
     algorithm previously described)
```

Taking a snapshot of the simulation graph before the clock tick ensures that all the synchronous components are fed with the signals that were present on the wire just before the clock tick (step 1). This is equivalent to having a clock signal that reaches each component precisely at the same time, so there is no possibility of timing issues.

After a synchronous component generates its new outputs and updates its internal state (step 3), such outputs are propagated through the combinatorial logic similarly to how an input is fed in the graph (see beginning of section 5.6).

The time complexity of the algorithm is $O(S*E+N)$, where S is the number of synchronous components, E is the number of edges and N is the number of components (usually S is much smaller than N). This follows from the fact that each of the S synchronous components produces an output which has to be fed into the combinatorial logic. Feeding a signal into the combinatorial logic has $O(E)$ time complexity (see beginning of section 5.6). Finally, taking a snapshot of the circuit diagram requires iterating through all the components, hence it has $O(N)$ time complexity.

5.6.2 Simulation API

The simulation and analysis logic is contained in the `Simulator` F# project. This logic is exposed to other parts of the application via some high-level API, which are described in table 5. These intuitive API hide the complexity of the underlying algorithms. Any sort of user interface can be built on top of these, as they provide a comprehensive set of functionalities to manipulate and extract the state of a simulation graph.

An example usage of these API would be:

1. Call `prepareSimulation` and obtain a simulation graph. When running this step, the simulator logic will analyse the circuit running the algorithms described in section 5.5, and then transform the circuit from the 'flat' representation into a simulation graph.

2. Call `feedSimulationInput` to inject a signal into the circuit and obtain the updated simulation graph. This step will trigger the algorithm described at the beginning of section 5.6.

3. Call `extractSimulationIOs` to extract the new values of the simulation outputs and display them to the user.

4. Call `feedClockTick` to advance the global clock in the simulation and obtain the updated simulation graph. This step will trigger the algorithm described in section 5.6.1.

5. Call `extractStatefulComponents` to get the updated state of all the stateful components and display them to the user.

5.7 Hierarchical Components

Another crucial DEflow feature is the possibility of packaging a circuit diagram and reusing it as an individual component in another circuit (requirement E3 from section 4). Such a component is a so-called *hierarchical component*. For example, figure 17 shows how a half-adder can be used to implement a full-adder. This feature allows the user to manage the design complexity by incrementally building more and more complex functionalities on top of existing ones. Hierarchical components also avoid duplicate circuit logic. In fact, using multiple half-adder components is arguably much cleaner than replicating the half-adder logic.

Nonetheless, the introduction of hierarchical components presents a series of challenges. The most interesting ones will be analysed in this subsection.

API function	Description
`prepareSimulation`	Takes a circuit diagram, the name of the design, and a list of loaded dependencies. Runs all the circuit validations and returns a simulation graph representing the circuit. Returns an error if any of the validations failed.
`feedSimulationInput`	Takes a simulation graph, the ID of the input component to use to feed the input signal, and the input signal itself. Recursively feeds the input in the simulation graph and returns the updated simulation graph.
`feedClockTick`	Takes a simulation graph. Feeds a global clock tick to all synchronous elements in the graph and returns the updated simulation graph.
`extractSimulationIOs`	Takes a simulation graph and a list of IDs of input or output components. Returns the signal currently produced or received by such components.
`extractStatefulComponents`	Takes a simulation graph. Returns all the stateful components in the graph (hence their current states).

Table 5: API provided by the simulator module.

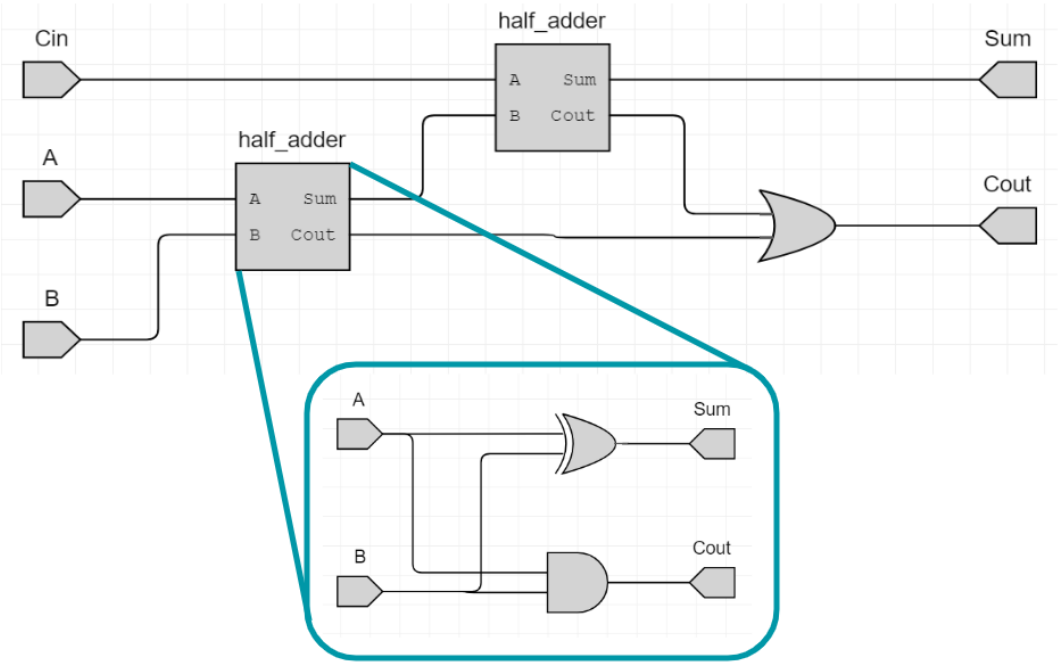

Figure 17: Full-adder designed using two half-adder hierarchical components.

5.7.1 Hierarchical Components Simulation

Simulating hierarchical components requires the simulator to be aware of the logic they contain. There are two main approaches to allow this. The first is to transform the simulation into a recursive process, while the second is to 'flatten' all the hierarchical components into a single large simulation graph. This subsection analyses the benefits and drawbacks of the two approaches.

While all other components have well defined behaviours (for example an AND gate or a D-flip-flop), hierarchical components behaviour depends on the logic they implement. This means that, when creating a simulation graph, it is not possible to assign a predefined reducer function to the simulation node. There exists a clean and recursive solution to this problem: since a hierarchical component is itself a circuit diagram, it can be transformed into a simulation graph, and this simulation graph can be used as the reducer for the hierarchical node. This yields that the simulation graph is now a recursive data structure, where certain nodes can contain other simulation graphs. To better understand this concept, consider again the full-adder diagram in figure 17. When a new input is fed via the input component `A`, it gets propagated to the leftmost half-adder. The half-adder hierarchical component feeds the new input into its internal simulation graph and extracts the corresponding new outputs. These outputs are then propagated 'outside' the half-adder component.

This recursive design flawlessly scales to an arbitrary number of nested components, keeping the overall simulation logic mostly unchanged. Furthermore, it makes the simulator aware of the internal structure of the circuit. This, in turn, makes it possible to create a simulation interface that allows the user to 'zoom' into the specific hierarchical components they are interested in. Therefore, users can selectively analyse and debug the behaviour of such components.

An alternative design is to avoid the recursion in the simulation graph by merging the simulation graphs of all the hierarchical components into a single, larger simulation graph. This is the approach chosen by Quartus. In the diagram in figure 17, the half-adder components would therefore be replaced by the actual half-adder circuit. This approach has been implemented and found to have the following downsides:

- The resulting simulation graph tends to grow very large and have a lot of duplicated logic. This is highly undesirable when running analysis, because the same logic gets needlessly analysed over and over, harming performance. For example, if you analyse the half-adder component once and find out it contains no combinatorial cycles, there is no need to re-analyse its logic again.

- The 'flattening' algorithm is complex. In fact, merging together different simulation graphs requires the creation of extra connections that do not exist in the diagram, and the deletion of others. Furthermore, each node in the simulation graph is associated with a unique ID, so it is also necessary to explicitly change all of the IDs to guarantee unicity. Such a complex algorithm is harder to understand and modify by future maintainers. Not less importantly, it makes it more difficult to prove its correctness.

- A flat simulation graph maintains no information about the original circuit structure. This means that it would not be possible for the user to explore the behaviour of specific hierarchical components of their circuit.

Given its major advantages, the recursive approach has been chosen. This decision will affect the algorithm to detect the implementation of combinatorial cycles proposed in section 5.7.3.

5.7.2 Dependencies Analysis

The introduction of hierarchical components requires changes in the analysis process. This subsection will present how the content of hierarchical components is analysed, and how DEflow ensures that hierarchical components do not form dependency cycles.

Consider again the full-adder example (figure 17). In order for DEflow to be able to simulate the circuit diagram, it must have access to the half-adder diagram. Therefore we define the half-adder to be a *dependency* of the full-adder. During the analysis process, a module called `DependencyMerger` tries to resolve each dependency by looking it up among the diagrams in the currently open project. If the required dependency is found, it is analysed in search of design flaws as described in section 5.5.2, otherwise an error is returned. As the dependency itself may have its own dependencies, this process is recursive. For performance reasons, DEflow keeps track of what dependencies have been found and analysed, so no dependency is ever analysed twice.

Another issue that has to be considered is that an inattentive user may introduce a dependency cycle. For example, if circuit `A` contains hierarchical component `B`, while circuit `B` contains hierarchical component `A`, then it is impossible to simulate either circuit. To detect these issues, DEflow builds a dependency graph, and analyses it in search of cycles. If a dependency cycle is found, DEflow produces a descriptive error message with information about it.

5.7.3 Combinatorial Cycles Detection

One of the analyses performed by DEflow is checking for the presence of combinatorial cycles. These are not allowed, as they are an advanced topic that requires the knowledge of the concept of propagation delays, which is not in the DECA first year syllabus. Note that cycles including synchronous logic (for example a state machine) are perfectly valid and possible in DEflow.

To better understand how DEflow can detect combinatorial loops, it is helpful to start with a simplified scenario without hierarchical components. In such a scenario, it is straightforward to determine whether a component is combinatorial or synchronous just by looking at its component type (for example, an AND gate is combinatorial while a D-flip-flop is synchronous). What follows is the algorithm to determine the presence of combinatorial cycles in this simplified scenario (backtracking of cycle components have been omitted for clarity). The algorithm is largely based on existing methods for cycle detection in a directed graph [14]:

```
Variables:

graph: the simulation graph
visited: a set containing the visited nodes, initially empty
stack: a set containing the nodes visited in the current recursion stack, initially empty

Algorithm:

1. For every node not inside the visited set:
2.   If the node is synchronous it cannot be part of a combinatorial cycle: return
3.   If the node is present in the stack, a cycle has been detected: terminate the algorithm with an error
4.   If the node is among the visited ones, we have already explored it and determined it is not part of any
     cycle (otherwise the algorithm would have terminated already): return
5.   Add the current node to the set of visited nodes
```

	Disallow HC from being part of cycles	**Use heuristic to determine if HC is safe in cycle**	**Keep track of combinatorial paths inside HC**
Restrictions	Blocks every design that would have HC as part of a cycle	Blocks some legitimate designs such as Mealy state machines	None
Performance	No extra cost with respect to the baseline algorithm	No extra cost when algorithm runs, but requires some precalculations	Negligible extra cost as the algorithm also keeps track of ports being visited for HC, but requires some precalculations
Implementation	Simple, treats HC as combinatorial components	Complex, but can be seen as a composition of simpler algorithms	Most complex, but can be seen as a composition of simpler algorithms

Table 6: Comparison of three alternative approaches when determining whether a hierarchical component is part of a combinatorial cycle. HC stands for 'Hierarchical Components'.

```
6.   Add the current node to the stack
7.   For each child of the node, repeat the algorithm from step 2
8.   Remove the current node from the stack and return
```

This algorithm has $O(N + E) = O(E)$ time complexity, where E is the number of edges in the graph and N the number of nodes. This directly follows from the fact that a node (hence its outgoing edges) are visited at most once, thanks to step 4 of the algorithm.

Reintroducing the concept of hierarchical components breaks step 2 of the algorithm. In fact, there is no way of knowing a-priori whether using the hierarchical component in a cycle may lead to a combinatorial cycle or not. Consider the case where a hierarchical component is used as part of a combinatorial cycle (in essence, all other components of the cycle are known to be combinatorial). If such a hierarchical component is, for example, a half-adder the cycle should be detected as combinatorial and an error should be returned. On the other hand, if the hierarchical component contains a two bits register the cycle should not be considered combinatorial and should be allowed by the application.

The problem of determining whether a hierarchical component can be used in a cycle or not can be approached in three different ways. The simplest way is to consider all hierarchical components as combinatorial components, since they *might* be. Another approach is to use an heuristic to determine if the hierarchical component could never lead to a combinatorial cycle (in which case the component is safe to use in a cycle), or if it might lead to combinatorial cycles, though it might not. In particular, the heuristic considers the hierarchical component to be combinatorial if and only if there is at least one combinatorial path from one of its inputs to one of its outputs. In other words, if the diagram of the hierarchical component has a path which connects an input to an output without encountering any synchronous component, then it is considered combinatorial. Such information can be precalculated before running the cycle detection algorithm. Finally, there exists a third approach which requires a different precalculation: for each input in each hierarchical component, keep track of what outputs are connected to it via a combinatorial path. When the cycle detection algorithm reaches an input port of the hierarchical component, it only proceeds exploring the children of the output ports that are 'combinatorially connected' to the input port. This approach requires the algorithm to keep track of the ports being visited, as well as the nodes. Table 6 proposes a comparison of these three approaches.

The heuristic approach was initially chosen, as it seemed to strike a good balance between detection

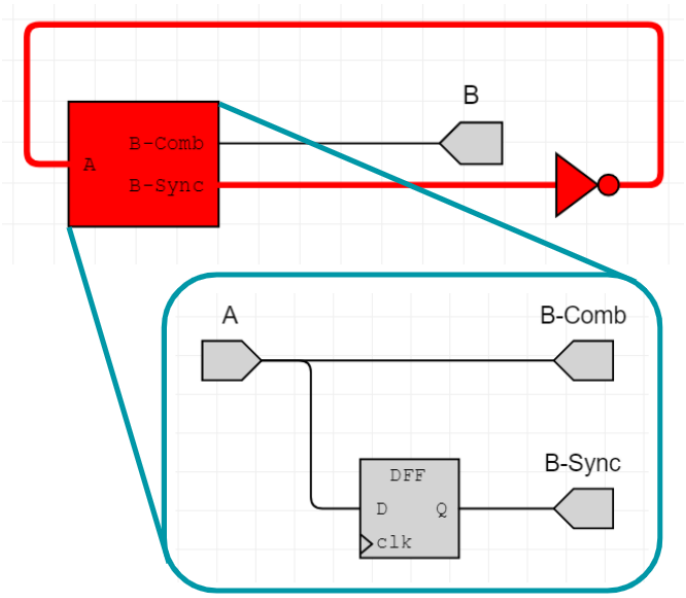

Figure 18: A non-combinatorial cycle, considered invalid in the old version of the algorithm, as the heuristic detected the hierarchical component as combinatorial.

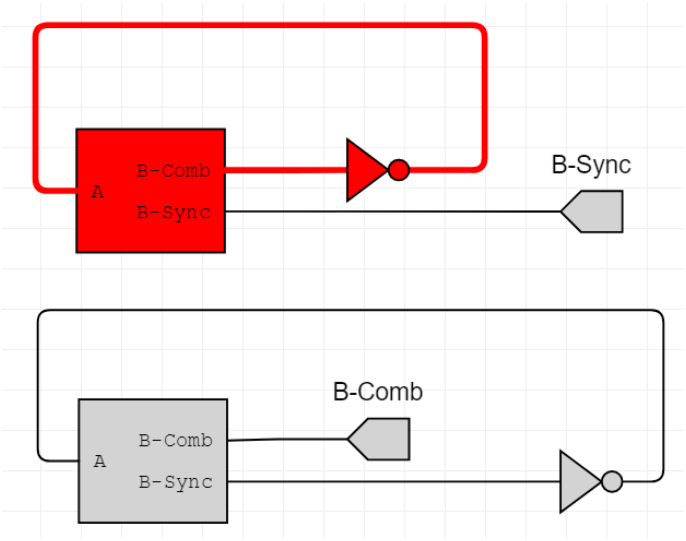

Figure 19: Final combinatorial cycle detection algorithm distinguishing between the two cases. The first diagram contains a combinatorial cycle, while the second does not.

accuracy and implementation effort. On the downside, that choice imposed restrictions on the diagrams that were allowed in the application. For example, consider the diagram in figure 18. The design contains no combinatorial cycles. The hierarchical component is considered combinatorial since it contains a combinatorial path from input `A` to output `B-Comb` which might lead to combinatorial cycles. Therefore DEflow highlighted the suspected combinatorial cycle in red, and considered the diagram invalid.

The heuristic was subsequently decided to be insufficiently accurate, as it blocked legitimate and useful circuits such as Mealy finite state machines [94]. The last approach was instead implemented, which is more complex but it allows the application to precisely detect all the combinatorial cycles without blocking any valid design. This approach requires precalculating, for each hierarchical component, which inputs are 'combinatorially connected' to which outputs. An input is 'considered combinatorially' connected to an output if there is at least one path from the input to the output encountering no synchronous components. Such information can be determined with an enhanced depth first search if the hierarchical component does not contain nested hierarchical components. If there are such nested components, instead, the precalculation algorithm has to be run recursively inside them first. Therefore, the overall precalculation algorithm can be seen as a composition of a post-order traversal of the dependency tree, together with a depth first search in each node of the dependency tree. DEflow optimises this process by keeping track of what dependencies have been explored to avoid having to re analyse them. Figure 19 shows how the final approach manages to distinguish between combinatorial and synchronous cycles with hierarchical components.

5.8 Mutability and Performance Optimisations

Throughout this chapter, a series of algorithms have been analysed. Each one of them have been proposed via a non-language specific pseudocode, and their asymptotic time complexities have been analysed.

All of these algorithms have been implemented using F#. As mentioned in section 3.3.3, F# is a hybrid language which also supports mutable data types. Carefully using mutability may sometimes be beneficial for performance or readability, and this subsection will explore how this concept applies in the context of this project.

In order to achieve the optimal time complexities of the algorithms proposed, it is necessary to use F# dictionaries, which are based on mutable hash maps with $O(1)$ lookup and insertion time complexity. These dictionaries are actually a direct port of a mutable C# collection, and do not support useful functions such as `map` and `fold`. F# maps, on the other hand, are immutable and based on a balanced binary search tree. This means that every map insertion and lookup is $O(logN)$ where N is the number of elements of the map itself. Therefore, in this case, mutable data would allow to reach optimal time complexity while immutable data add an extra logarithmic factor to each term (for example $O(E)$ would become $O(E * logE)$).

Nonetheless, it is arguable that a diagram will hardly have over a hundred components or wires, so the logarithmic factor can be approximated to $\Theta(log100) = \Theta(7) = \Theta(1)$. This approximation yields that the expected time complexity of the algorithm remains $\Theta(E)$. Furthermore, though using hash maps allows to achieve the optimal asymptotic time complexity, it will likely not increase the real-world performance of the algorithms as hashing is an expensive operation and a search in a small binary search tree is generally faster [5].

Because of these reasons, it was decided to stick to a purely immutable implementation. This allows the developer to better benefit from the functional programming advantages mentioned in section 3.3.3.

Additionally, having no mutable variables allows the Expecto testing library to safely run all the testcases in parallel (see section 6).

5.9 Summary

This section covered the main implementation challenges that have been encountered during the development of DEflow. Each challenge led to design choices, which alternative approaches have been carefully analysed. Additionally, a thorough analysis of several graph algorithms have been carried out, with considerations about possible optimizations.

The implementation directly affects the testing of the application (section 6), as well as the considerations about code quality presented in section 8.4.

6 Tests

One of the main emphasis of the project is to deliver an application that can be easily maintained and extended (requirement D4 from section 4). In this section, we will discuss how DEflow testing plays a key role in ensuring such properties.

As already discussed in section 5.1, the F# non-testing logic is split into two main sections: the user interface logic and the core application logic. The former has the role to create the visible layout for the user to interact with, while the latter contains all the algorithms that allow to analyse and simulate a circuit. Crucially, the core application logic is written in pure F# and can therefore run under the .NET framework. On the other hand, the user interface logic can only run when compiled to JavaScript via the Fable compiler.

6.1 Core Application Logic

The core application logic has been extensively tested with the `Tests` module (see table 3). The testing infrastructure is based on the Expecto library, which can run under the .NET framework but cannot be compiled to JavaScript via Fable. Expecto provides very intuitive API to configure and run tests on the codebase. Furthermore, Expecto runs all tests in parallel by default, which greatly speeds up the testing process. It is completely safe to use this parallelisation as the core logic being tested contains no mutable variables that could interfere in concurrent tests.

Having a comprehensive set of tests for core logic offers two major benefits. Firstly, tests ensure the functional correctness of the algorithms implemented. Such algorithms can sometimes be a few hundreds of lines of code long, which makes it difficult to guarantee their correctness just by reading the source code. Secondly, tests allow developers to change the algorithms and have an easy way to assert they did not break existing functionalities. This type of testing is often referred to as *regression testing*. Regression testing is crucial when the codebase grows, as it would become more and more time consuming to manually ensure the correctness of every algorithm every time a change is made. This, in turn, massively improves the maintainability and extensibility of the application.

Tests have been built in such a way that they do not rely on assumptions regarding how the underlying algorithms work. In other words, the logic is tested via the exposed API, rather than with module specific functions. This ensures that a large amount of unit tests does not have to be rewritten if the internal logic of an algorithm is changed, for example, due to an optimisation. In fact, no change at all should be required in such a scenario since an optimisation should not change the behaviour of the algorithm itself.

On the other hand, tests have to be carefully designed in order to cover all the possible behaviours of the algorithms exposed by the high level module API. This is a non-trivial task that requires a deep understanding of the code being tested. Nonetheless, once this comprehensive set of tests is in place, it provides strong guarantees against regression, and it does not unnecessarily slow down the development process by requiring the change of many small tests.

Testing in a functional programming language requires three main components: an input, the function to test and the expected output. No other element is needed, as pure functions just transform data without any side effect. In DEflow, the inputs are always represented by circuit diagrams hardcoded in the testing logic, with some additional parameters such as dependencies for circuits containing hierarchical components.

More than 70 different circuits are used in the tests. These circuits are generally small (usually less than

10 components), as they aim to test specific behaviours of the API with the minimal amount of complexity possible. This greatly facilitates understanding what is going wrong when tests do not pass. Furthermore, a developer can easily load the circuit in DEflow itself and debug its behaviour via the visual interface, which is often helpful. Nonetheless, among the test inputs there are a few complex circuits, such as a two bits adder. This aims to ensure that the logic works for relatively big inputs, but they should not be used as the primary testing method, as they make it more difficult to understand what part of the algorithm is failing.

Overall, DEflow has more than 140 tests, which cover all the behaviours of the `WidthInferer` and `Simulator` modules logic.

6.2 User Interface Logic

The user interface logic has been extensively manually tested. It was decided that programmatically testing this logic was unnecessary and undesirable because of the following reasons:

Logic is Modularised The Model-View-Update architecture allows the separation of the UI logic in independent views. Each view manages their own part of the model, without affecting the others. Hence, upon changing part of the user interface it is enough to manually test that the desired changes took place as expected, while the architecture will ensure that other parts remain unchanged.

Lengthy Setup The UI logic interoperates with JavaScript via modules provided by the Fable compiler. Therefore, it cannot be run and tested under the .NET framework. This, in turn, means that another separate testing infrastructure would have to be set up. Such infrastructure would probably be based on the Jester library [29]. This library is an F# port of Jest [43], which is the testing tool recommended by the authors of the React framework itself. It allows you to run tests with a simulated browser DOM. However, setting up a further testing infrastructure is a very time-consuming task, and priority has been given to adding more features to the application itself.

Constant Evolution Testing the UI logic means generating an HTML view on a virtual DOM and comparing it with the expected result. This means that, in order for a test to pass, every element of the interface has to precisely match the ones encoded in the testcase. Given that the UI often requires small changes such as rewording an error message or restyling a button, the process of updating the UI tests becomes very time consuming. This, in turn, would make extending the application cumbersome, going against one of the requirements of the project itself.

6.3 Summary

In summary, DEflow algorithmic logic has been extensively tested to guarantee correctness and non-regression upon future changes. DEflow features over 140 tests on more than 70 circuit diagrams. The user interface relies on the solid MVU architecture in order to guarantee that changes are compartmentalised and cannot negatively impact separate features.

The considerations about the testing logic presented in this section will be used to inform the evaluation of the code quality (section 8.4).

7 Experiment Methodologies

In this section, I will be describing the experiments that have been carried out on DEflow and its alternatives. The results of these experiments will be used in section 8 to inform the evaluation of usability and performance of the application.

Three types of experiments will be discussed in this section: a user feedback survey that has been conducted among university students (subsection 7.1); a comparison against alternative hardware design platforms that I performed (subsection 7.2); and the design of a large electronic circuit I carried out using the application itself (subsection 7.3).

7.1 User Feedback Collection

User feedback is a crucial part of the evaluation process of every user-facing application. It allows developers to get a different point of view on the product they are creating. In fact, certain parts of the application logic can appear obvious to the engineer that created them, but may not be intuitive to users. Collecting user feedback allows developers to get this sort of valuable information which can be used to improve the product itself.

As soon as the core functionalities of DEflow were in place, I decided to start collecting user feedback. This, in addition to providing useful data to evaluate the usability of the application, also gave me the time to integrate part of the user feedback in the final project deliverable (which will be discussed in section 8.6).

The user feedback collection has been carried out among university students with previous knowledge of digital electronics concepts. This aims to mimic the basic knowledge that first year students will have after receiving an introduction of digital electronics concepts before their first laboratory session. Though this approximation is imprecise, the alternative was to collect feedback form users with no previous digital electronics knowledge. Such an approach would have probably not yielded informative results since users would be mostly confused about digital electronics topics, hence could not make any sensible use of the application.

It proved difficult to gather volunteers due to both Covid-19 UK lockdown and the overlapping with the exam term. Nonetheless, nine responses were collected. They mostly come from Imperial College first year EE students, but also from some Cambridge University Computer Science third year students (who use Quartus extensively during their second year of studies).

The feedback survey was expected to take around 35 minutes and it was split in three sections:

1. In the first section, volunteers downloaded DEflow and used it to design five small electronic circuits. At the end of each task, users were asked questions about their experience with the application. For example, users were asked roughly how long it took them to carry out the proposed task and whether the various functionalities they used were intuitive.

   The five tasks have been chosen to cover most of the core functionalities of the application, without pushing the volunteers to spend an unreasonable amount of time on them. The tasks required users to create, validate and simulate: an AND gate, a half-adder, a full-adder, a ROM and a two bits register. This set of tasks allows users to try to implement circuits with combinatorial and synchronous logic, as well as buses, memories and hierarchical components.

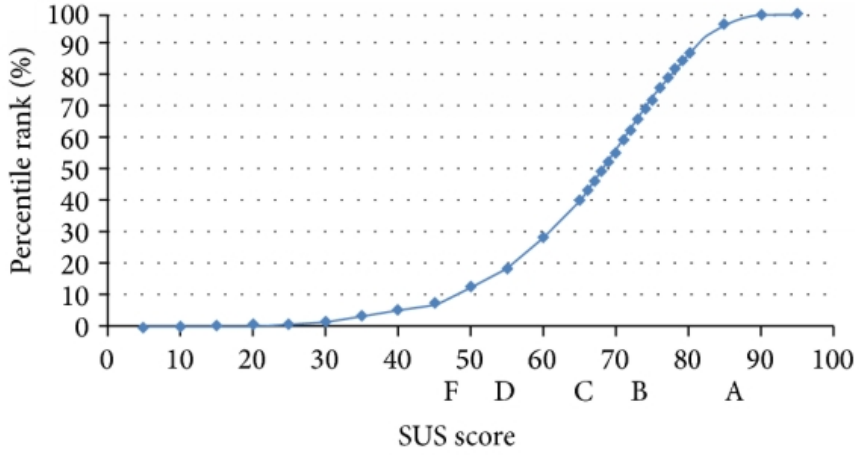

Figure 20: Percentile ranks related with SUS scores and letter grades. 50% of applications score more than 68, but only 10% score more than 80. Image credits: [62].

One of the main characteristics of this section is that users have been given minimal guidance for the tasks. For example, instructions stated to create a project and design an AND gate, but they did not provide details about the steps involved in such operations. The aim of this choice is to verify that the application can actually be used intuitively without requiring any user manual.

2. The second section of the questionnaire consisted in the System Usability Scale (SUS) [49, 68]. SUS is composed of ten statements that each user must rank on a scale from 'Strongly Disagree' to 'Strongly Agree'. Each user's response can be used to calculate a 'usability score' for the application, ranging from 0 to 100. A score above 68 is considered above the average, while a score above 80 is considered excellent. Figure 20 shows the fitting curve for SUS scores.

   For DEflow evaluation, it has been decided to remove the first of the ten SUS statements: "I think that I would like to use this system frequently". In fact, this statement makes little sense for an educational software primarily intended to be used in laboratories. DEflow SUS score has therefore been rescaled from 90 to 100 to make up for the removed points.

   SUS is nowadays considered an industry standard to evaluate the usability of digital products. It has been used in over 1300 publications and it has been observed to be valid (can reliably tell whether a system has usability issues or not). On the other hand, I decided to not solely rely on SUS as it is domain agnostic and it can not identify what usability issues the application may have (though it can point out whether such issues do exist).

3. The third part of the questionnaire indagated users' experience with Quartus via a series of 'Strongly Disagree' / 'Strongly Agree' questions. The aim of this section is to consolidate the understanding of Quartus usability concerns, as well as collecting users' opinion on whether DEflow is perceived to be more intuitive and usable than its counterpart.

Additionally, users were given the option to provide general feedback on the application. This guided

the addition of the several features which will be described in section 8.6.

The survey responses[1] will be presented and analysed in section 8.

7.2 Comparison with Alternative Software

To gauge the benefits of DEflow in the context of DECA teaching, I personally compared it to existing hardware design platforms. An initial analysis of the high level features of a series of applications was proposed in section 3.2. That overview allowed me to identify three main candidates that could be used for teaching digital electronics: Quartus, SimulIDE and BrainBox. Each of these applications has therefore been thoroughly tested on a series of tasks. This subsection introduces how the testing was carried out, while the evaluation of the results will be proposed in section 8.2.

DEflow and the three alternative software have been compared on three tasks: creating a half-adder, creating a full-adder and creating a two bits up counter. Each design has also been analysed (when possible) and simulated. These tasks allowed me to get an understanding of the capabilities of each application, as well as providing information regarding their usability. The comparison was based on the features presented in table 11. The table will be analysed in section 8.2.

Unfortunately, the evaluation of SimulIDE could not be terminated due to a crash in the application when trying to route wires in the full-adder design. The application could not be restarted as it could not detect anymore the QT libraries installed in the system. Nonetheless, enough information was gathered to compare it according to the features in table 11.

7.3 CPU Design in DEflow

While the user feedback and the comparison with alternatives mainly focused on quantifying the usability of individual features on relatively small tasks, it is crucial to verify that DEflow can be actually used for complex designs. In order to ensure this, I decided to use the application to implement a four bits RISC CPU. The CPU design is formed by:

Program Counter The Program Counter is a four bits register that contains the address of the instruction being currently executed.

Instruction Memory The Instruction Memory is an asynchronous ROM which contains five bits instruction words. The most significant bit of each instruction encodes the operation mode for the ALU, while the remaining four bits are used as an immediate operand. The ROM can be programmed before the beginning of the simulation process.

Decoder This component contains the logic necessary to decode the instructions from the Instruction Memory and to feed the ALU with the relevant control signals and data.

Arithmetic Logic Unit The Arithmetic Logic Unit (ALU) carries out the arithmetic operations. The ALU supports a control signal to decide its operation mode, and two four bits operands. There are two supported modes: addition and subtraction.

[1] The raw responses can be found at `https://docs.google.com/spreadsheets/d/1am6Sh7xbW5A7GlVn2uATqbrn3z4Z4sS4axUH-1Bz-u8/edit?usp=sharing`

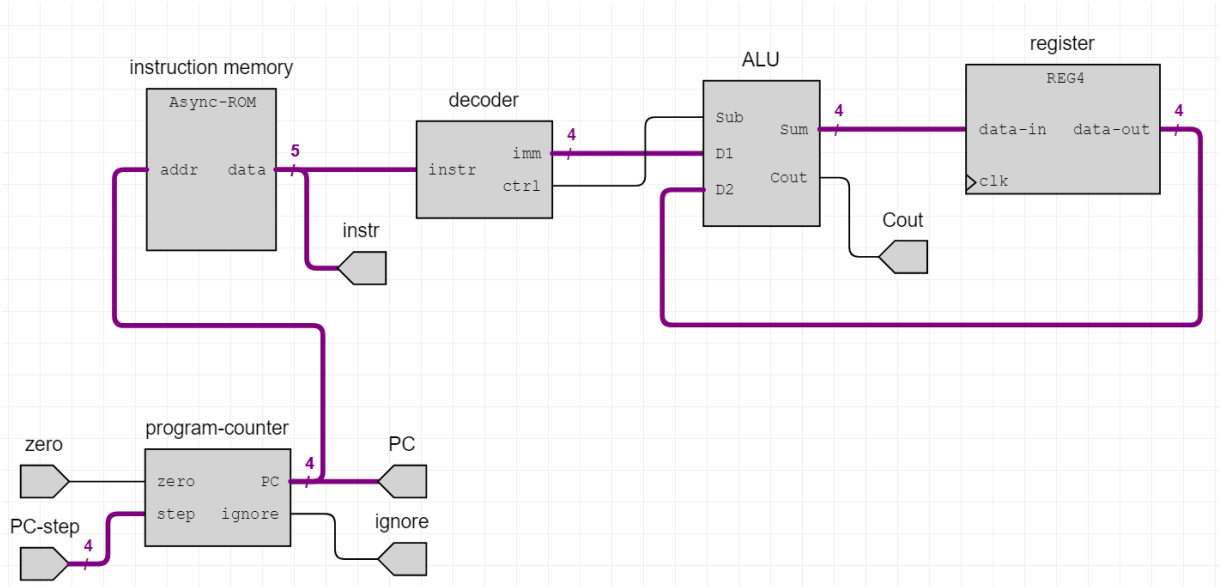

Figure 21: A RISC CPU implemented with DEflow.

Register The Register has the role of storing the result of the ALU operations. It also acts as one of the two ALU operands.

The top level circuit diagram of the CPU can be found in figure 21.

This experiment will be used in the evaluation (section 8) to inform the considerations about the application usability and performance.

8 Results and Evaluation

This section will present the results of the experiments described in section 7 and provide a critical evaluation of the project outcomes, in relation to the requirements established in section 4.

First, an evaluation of the usability of the application will be presented in subsection 8.1. Subsection 8.2 will compare the features of the application against existing hardware design software. The performance of the application will be analysed in subsection 8.3. Subsections 8.4 and 8.5 will concern the quality and extensibility of the application code. Subsection 8.6 will present the improvements suggested by the users. Lastly, a reflection on what requirements have been fulfilled will be provided in section 8.7.

8.1 Usability

One of the core requirements of the application is that it ought to have a high degree of usability. Usability is a broad concept, so I decided to evaluate it using four pillars:

Intuitiveness How quickly can a first time user master a certain feature? Can this be done without referencing a user manual?

Error Messages How well error messages lead the user toward fixing their issues?

Scalability Can the application be used to create complex designs?

System Usability Scale How well does the application score in the commonly used System Usability Scale?

Each of these pillars will be thoroughly discussed in the following subsections.

8.1.1 Intuitiveness

The user feedback survey described in section 7.1 contained a series of questions that can be used to evaluate how intuitive the users perceived the application to be. In fact, at the end of each practical task, the user was required to roughly estimate the time it took them to complete it, plus some questions about how intuitive the various features they used were.

It is important to stress out that the questionnaire only provided the minimum amount of instructions necessary for the user to understand the task they are supposed to perform. For example, the instructions of the first task were:

1. Open the application.
2. Create a new project called TEST.
3. Crate a new file called and_gate.
4. Add an AND gate to the diagram and connect it to two inputs and one output.
5. Run a simulation, change the input values and verify that the AND gate actually works.

Task	Less than 5 minutes	Less than 10 minutes	Less than 15 minutes	More than 15 minutes
AND gate	100	0	0	0
half-adder	88.9	11.1	0	0
full-adder	55.6	33.3	0	11.1
ROM	66.7	33.3	0	0
2 bits register	77.8	11.1	11.1	0

Table 7: Time taken by the users to perform each questionnaire task. The numbers reported are percentages.

Task	Statement	Score
AND gate	Creating a project and a file was intuitive	4.89
	Adding components to the diagram was intuitive	4.44
	I was quickly able to grasp how to navigate the interface	4.44
half-adder	It was easy to create a design with several components	4.55
full-adder	I was quickly able to grasp how to reuse the half_adder component	4.33
	I was quickly able to understand how to simulate combinatorial logic	4.33
ROM	I was quickly able to understand how to view/edit the memory content	4.67
	The memory editing interface was intuitive	4.55
	I was quickly able to correctly use multi-bit input/output components	4.44
	I was quickly able to make use of multi-bit inputs/outputs in the simulation	4.67
2 bits register	I was quickly able to understand how to merge and split wires	4.00
	The wire labelling is clear and useful	4.44
	I was quickly able to step through a synchronous simulation	4.00
	It is clear that synchronous components are implicitly connected to the global clock	3.67

Table 8: Score associated with each statement after questionnaire tasks. The score is the average of all user responses, where 5 is 'Strongly Agree' and 1 is 'Strongly Disagree'.

This initial task introduces the user to several new concepts, such as creating a project, creating a file, adding components to the circuit diagram and simulating the circuit they created. Nonetheless, 100% of the participants reported that the entire task required them less than 5 minutes to complete (table 7).

Table 7 reports the times taken by the users to perform the five tasks in the questionnaire. Most of the time, users were able to familiarise with the features of the application and complete the task in less than 5 minutes. Creating and simulating a full-adder proved to be the longest task, probably due to the larger amount of components in the circuit. One user reported that they struggled to understand how to reuse the half-adder component they created in the previous exercise, and therefore it took them more than 15 minutes to complete the full-adder task. DEflow allows users to use previously created components by scrolling to the bottom of the components catalogue. I expect that, once the user becomes familiar with this concept, it would take them less than 5 or 10 minutes to complete the task.

After the completion of each task, the users are asked to state their level of agreement with a series of statements on a scale from 'Strongly Disagree' (score 1) to 'Strongly Agree' (score 5). Table 8 reports all of

Task	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree	Number of responses
AND gate	0	100	0	0	0	1
half-adder	100	0	0	0	0	3
full-adder	100	0	0	0	0	5
ROM	83.3	16.7	0	0	0	6
2 bits register	60	40	0	0	0	5

Table 9: User agreement with the statement "The error messages guided me well toward fixing the problems" at the end of each task.

these statements and their aggregate scores, obtained by averaging the scores of all the user responses. Since almost every score is above 4, it can be stated that users are generally pleased with their experience with the various features, which is also proven by the short execution times reported in table 7. It seems that the most confusing part of the application is the fact that synchronous components are implicitly connected to the global clock. Yet, once a user is familiar with such a concept, they should not be confused by that anymore.

Overall the application seems to be very intuitive, with most of the first time users being able to complete a variety of tasks in less than five minutes, without a detailed guidance on how to do so.

8.1.2 Error Messages

Requirement D3 (from section 4) focuses on the ability of the application to provide helpful and thorough feedback for errors. Thanks to its underlying MVU architecture, DEflow can provide immediate feedback for errors in the form of error notifications and, when relevant, by highlighting parts of the circuit diagram (see section 5.2 for details).

At the end of each task, users were asked to rate how well the error messages guided them toward the resolution of the issues (users could skip such questions if they received no error notifications). The collected responses are summarised in table 9. The number of responses for each task shows that users tended to complete easier tasks with less errors, which was expected. Users highly appreciated errors for combinatorial logic design (the first 4 tasks). It is possible that some error messages related to the creation of buses (necessary in the last task) may be slightly less clear, as the last row has two users not strongly agreeing with the statement. On the other hand, this may be due to experiment noise given the low number of data points (only 5 users answered that question in the last task).

In conclusion, users seem to uniformly agree that the error messages are clear and helpful. This is a very important requirement toward the usability of the application.

8.1.3 Scalability

As the application ought to be used for first year DECA teaching, it must allow students to create complex designs that are covered by the course syllabus.

In order to verify that DEflow has such capability, it has been used to implement the RISC CPU described in section 7.3. The implementation process took me slightly less than an hour to complete. I expect an

experienced DEflow user with a clear understanding of the design that they are going to create to match this timing. The design process involved the creation of many intermediate components such as a full adder and a decoder. Creating and handling these components was very easy thanks to the responsiveness of the application. In fact, upon changing the circuit diagram file, the previous circuit is immediately listed as a component in the catalogue. The absence of delays when analysing and simulating a circuit also played an important role in providing a smooth user experience. Finally, the immediate error feedback provided by the application allowed me to promptly identify and fix the design bugs.

Unfortunately, due to a bug in the JavaScript drawing library it is not currently possible to save the user changes to the routing of the wires. In other words, if the user moves a wire, then closes and reopen the diagram, such wire will be automatically rerouted with the default algorithm. This can be annoying when a user needs to deal with designs with a lot of wires, such as a four bits adder. The bug has been reported to the creators of the library and it will hopefully be fixed in the next release of the library [16].

In conclusion, the process of implementing a RISC CPU proved to be extremely smooth and pleasurable, demonstrating that DEflow can indeed be used for large designs.

8.1.4 System Usability Scale

As mentioned in section 7.1, one part of the questionnaire required users to evaluate the application according to the System Usability Scale (SUS). Such scale is an industry standard for evaluating digital applications. As previously mentioned, an application that scores 68 on this scale is considered average, while any score above 80 is considered excellent.

The responses of each user have been recorded and used to evaluate DEflow SUS score. On average, DEflow scored 84.57, well above the excellence threshold. This result matches the positive feedback obtained in the previous sections, further confirming that the users appreciate the application.

8.1.5 Summary

This section provided an analysis of DEflow usability based on four pillars: intuitiveness, error messages quality, scalability and System Usability Scale score. The evaluation was based on both user feedback and a CPU design I personally made. Overall, each evaluation pillar yielded a very positive result, confirming that the application is indeed usable.

8.2 Comparison with Alternatives

In order to be adopted for Imperial College EE DECA teaching, DEflow must be preferable over the alternative hardware design platforms. In this section, a comparison of DEflow against other software will be proposed. The comparison is based on the results of the final part of the user feedback survey (which are reported in table 10) and on the manual testing described in section 7.2.

#	Statement	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	I tend to get confused in Quartus menus, when I perform familiar tasks (and I often have to consult a guide/handout)	12.5	0	50	37.5	0
2	I tend to get confused in Quartus menus, when I perform UNfamiliar tasks (and I often have to consult a guide/handout)	37.5	50	12.5	0	0
3	I was not confused by DEflow menus while performing the tasks in this questionnaire	25	62.5	12.5	0	0
4	I sometimes struggle to understand error messages in Quartus	37.5	25	37.5	0	0
5	DEflow errors guided me well toward fixing problems in my design	50	25	12.5	12.5	0
6	I feel that simulating the circuit is easier in DEflow than in Quartus	37.5	37.5	0	12.5	12.5
7	I feel like editing the content of memories is easier in DEflow than in Quartus	37.5	37.5	12.5	12.5	0
8	I feel I could have learnt Digital Electronics concepts with less guidance if I used a simpler app such as DEflow	50	25	25	0	0
9	Overall, I think using a simplified digital design app such as DEflow would have improved my Digital Electronics lab learning experience	37.5	37.5	25	0	0
10	DEflow is more pleasurable to use than Quartus	50	50	0	0	0

Table 10: Results of the third part of the user feedback form (see section 7.1). Statement 3 has been negated in order to transform it in a positive one (like the other rows). The original statement did not contain the 'not'.

Requirement	Feature	SimulIDE	BrainBox	Quartus	**DEflow**
E1, E2	Combinatorial components	✓	✓	✓	✓
E1, E2	Synchronous components	✓	✓	✓	✓
E1	Single-bit wires	✓	✓	✓	✓
E2	Memories (ROM/RAM)	✗	✗	✓	✓
E3	Hierarchical components	✗	✗	✓	✓
E4	Multi-bit wires (buses)	✗	✗	✓	✓
E5	Inputs and probes	✓	✓	✓	✓
E6	Simulator	✓	✓	✓	✓
E7	Circuit analyses	✗	✗	✓	✓
E8	Save and load circuits	✓	✓	✓	✓
E9	Maintained	✓	✓	✓	✓
D1	Shallow user search [2]	✓	✓	✗	✓
D1	Instantaneous circuit analyses	–	–	✗	✓
D1	Instantaneous simulation startup	✓	✓	✗	✓
D1	Uncluttered user interface	✓	✓	✗	✓
D2	Drag and drop (or equivalent) [3]	✓	✓	✓	✗
D2	Automatic wires routing	✗	✓	✗	✓
D2	Automatic bus width inference	–	–	✗	✓
D2	Waveform generator	✗	✓	✓	✗
D3	Display errors on circuit	✗	✗	✗	✓
D4	Extensible by EE students	✓[4]	✓[5]	✗	✓
D5	Developer's documentation	✗	✓	–	✓
D6	Open-source	✓	✓	✗	✓
D7	Fully cross-platform	✗	✓	✗	✓

Table 11: Comparison of the main features of four hardware design applications. The 'Requirement' column indicates the requirement under which the feature falls (see section 4 for a detailed requirements list).

As mentioned in section 3.2, the existing hardware design platforms that seem to best fit DECA teaching requirements are SimulIDE, BrainBox and Quartus. In particular, Quartus is the software currently in use for the DECA laboratories. The experiment described in section 7.2 allowed me to compare the applications on the list of features proposed in table 11. In the table, each feature is shown together with the requirement

[2] Users do not have to navigate several menus in order to perform common actions such as inserting components or running simulations.

[3] Users can insert new components in the diagram by dragging and dropping them.

[4] Requires maintainers to familiarise with the QT toolkit.

[5] Extensions would probably have to be written in JavaScript.

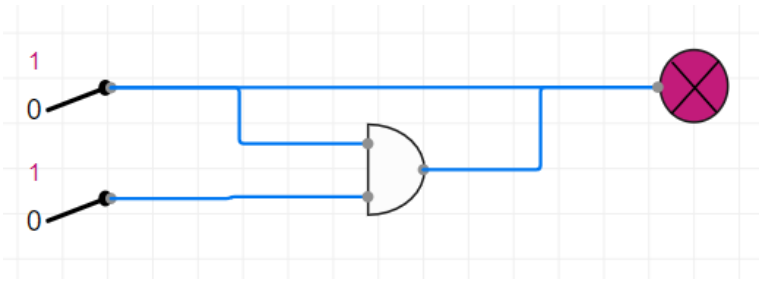

Figure 22: Undefined behaviour during a simulation of a simple circuit in BrainBox. The output is high (red colour) even if no voltage is applied to the circuit.

it falls under (from section 4).

There exist two types of requirements: Essential (letter E followed by a number) and Desirable (letter D followed by a number). The essential requirements are the features that must be part of the application in order for it to be adoptable for DECA teaching. If those features are missing, the application will not be versatile enough to allow students to reach the laboratory session objectives. On the other hand, desirable features can be absent from the application without affecting its ability to be chosen as a teaching tool, but their absence may negatively impact users' experience.

Table 11 clearly shows that only DEflow and Quartus have all of the essential features. SimulIDE and BrainBox, on the other hand, tend to only have the features necessary to design and simulate small circuits. In fact, they do not help users to handle design complexity since they lack both hierarchical components and buses. The absence of builtin ROM and RAM memories is also a fundamental shortcoming of these applications. The last missing essential feature is the ability of running analyses on the circuit design. This means that the user is allowed, for example, to create circuits with floating wires or wires driven by an arbitrary amount of signals. In both applications, the simulation behaviour of such circuits seems to be undefined. For example, figure 22 shows a BrainBox circuit where a wire is driven by two signals, and even though both signals are low, the component connected to such wire seems to receive a high signal (coloured in red).

Nonetheless, both SimulIDE and BrainBox seem to meet many of the desirable criteria. This is not surprising, as they are designed to be teaching tools. They both have an uncluttered user interface, an instantaneous simulation startup mechanism and the possibility of inserting components into the diagram via drag and drop. Additionally, both applications are open-source, so they could be extended to introduce the missing essential features. Unfortunately, SimulIDE appears to be mostly written in QT C++ and Assembly, and the application developer received complaints about how difficult it is to contribute to the project [65]. BrainBox is written in JavaScript and makes use of the JQuery library to handle UI updates [44]. As JavaScript is relatively easy to learn and commonly known, BrainBox would probably be easier to extend than SimulIDE. Nonetheless, relying on JavaScript as the main programming language for a large scale project poses some concerns, in particular related to the language type system. Such concerns have been analysed in section 3.3.3.

Both Quartus and DEflow support all the features that have been deemed as necessary for the tool to be used for DECA teaching. The big difference between the two applications lies in the set of desirable features. In fact, Quartus seems to miss many of them. For example, Quartus requires the user to navigate

several menus in order to start a simulation or create a new circuit diagram. On the other hand, DEflow focuses on reducing the number of menus as much as possible, to keep a clean interface that users should have no difficulties navigating. For example, DEflow allows the user to validate the circuit diagram and start a simulation with just two mouse clicks. The user feedback showed that 89% of the students agreed that they were "quickly able to grasp how to navigate the [DEflow] interface" with virtually no external guidance. This data has been obtained by the responses at the end of the first questionnaire task (see table 8), and confirmed by the results on statement 3 of table 10. On the other hand, 88% of the students agreed that they struggle to perform unfamiliar tasks with Quartus, and they often have to refer to a manual in order to complete them (statement 2 in table 10). Similarly, none of the users disagreed about the fact that they tend to get confused by errors in Quartus (statement 4).

Although statement 6 shows that the majority of the users found that simulating the circuit in DEflow was easier than in Quartus, 25% of them disagreed. Multiple user comments suggest that this is due to the lack of the waveform generator feature in DEflow, which is present in Quartus. For example, one user said "Quartus has a good VWF [Vector Waveform File] generator which would be nice to have in DEflow".

This comparison has shown that DEflow is the only application that provides the following two features: automatic buses width inference and displaying errors by highlighting the faulty components and connections in the circuit. The benefits of the former feature are thoroughly described at the beginning of the section about the inference algorithm (5.5.1). The latter greatly helps users to precisely grasp what is wrong in their design and it can be likely accredited for the extremely positive user feedback on error messages discussed in section 8.1.2.

Another noticeable difference between Quartus and DEflow, is that the former forces the user to wait a few seconds for the circuit analysis to complete, while such process is instantaneous in the latter. This is probably due to the fact that Quartus compiles the circuit into a format that can be both simulated and transformed into a physical circuit, which takes time. On the other hand, DEflow only performs the validations necessary to ensure the circuit contains no flaws and can therefore be simulated successfully. Similarly, Quartus takes a few seconds to start a simulation, while DEflow can do so without any delay.

DEflow, like BrainBox, can run on all major platforms (Windows, Linux and MacOS). This is not the case for SimulIDE and Quartus which cannot natively run on MacOS. Furthermore, since DEflow is open-source and mostly developed in F#, Imperial College students that attend the third year High Level Programming EE course will be able to maintain it with minimum extra learning effort.

In conclusion, it seems that DEflow fulfills the success criteria outlined in section 4 better than the alternatives. This suggests that DEflow is better in the context of DECA laboratories. Statements 8 and 9 of table 10 show that the users generally agree that using a simplified application like DEflow would have helped them learn digital electronics concepts with less guidance, improving their overall laboratory experience. Nonetheless, DEflow usability would probably benefit from the inclusion of some features currently present in other hardware design platforms, such as waveform generation or components insertion via drag and drop. Alternative platforms provide several additional features like Raspberry Pi support in BrainBox or analogue electronics components in SimulIDE, but these are not needed in first year DECA laboratories.

Algorithm or group of algorithms	Described in section	Time Complexity	Mean (ms)	Standard Deviation (ms)
Wires width inference	5.5.1	$\Theta(E)$	2.2	0.7
Diagram validations	5.5.2	N/A	3.6	1.1
Feed combinatorial signal	5.6	$\Theta(E_{tot})$	0.9	0.3
Feed clock tick	5.6.1	$\Theta(S_{tot} * E_{tot} + N_{tot})$	29.2	6.8
Cycle detection precalculation	5.7.3	$\Theta(E_{tot})$	2.5	0.9
Cycle detection	5.7.3	$\Theta(E)$	4.6	1.9

Table 12: Response time measurements for the various algorithms in the application. The measurements have been carried out on an Intel Core i7 CPU, clocked at 2.5 GHz.

8.3 Performance

Applications which feature a high level of user interactions are time critical. In particular, the productivity of users highly depends on the responsiveness of the application. For example, a DEflow user adding a bus to their circuit diagram should see its inferred width immediately, and the system should not get stuck performing such calculations.

I decided to explore the performance of the application by measuring the response times of the various algorithms described in section 5. This methodology provides a relatively accurate representation of the application behaviour, as these algorithms are directly exposed by the API of their respective modules, hence accounting for virtually the entire API call execution time. A performance analysis of the MVU interface updates is not presented here, as they are handled by the Elmish-React library and are generally outside the scope of possible optimisations. Moreover, the underlying React rendering engine ensures that updates are fast enough to be imperceptible by humans [3, 63].

An analysis of the asymptotic time complexity of the algorithms was already provided in section 5. This section aims to measure the real world performance of such algorithms by testing them on the CPU design described in section 7.3. The performance has been informally evaluated by manually running each algorithm 20 times and averaging the response times. These measurements have been carried out using my laptop, which is powered by an Intel Core i7 of sixth generation clocked at 2.5 GHz. Such informal performance evaluation aims to determine whether the algorithms are fast enough to be imperceptible by a human user, rather than providing a detailed performance profile. It is widely believed that response times under 100 milliseconds are perceived as instantaneous by human users [84].

Table 12 reports the results of these measurements, with their corresponding standard deviations. In the table, E is the number of wires in the circuit, N is the number of components and S is the number of synchronous components (usually S is much smaller than N). The *tot* subscript indicates that the elements nested within hierarchical components are also taken into consideration. All measurements have been collected on the top level circuit of the CPU design (in figure 21), except for the algorithm to feed a combinatorial signal into the simulation graph. In fact, this algorithm has been tested on the four bits ALU circuit, which is the largest largest combinatorial component of the entire CPU. In particular, the test has been carried out by changing the value of one the two four bits operands from `0x0` to `0xF` in order to update the signal on the maximum possible number of wires.

All the algorithms response times are well below the threshold of human perception. Therefore, when they are run as a result of events triggered in the user interface, the response of the application is perceived as instantaneous.

Feeding a clock tick into the simulation graph appears to be the slowest algorithm, which is expected as it has the highest time complexity. An in depth performance analysis of the algorithm seems to suggest that the process of recursively exploring all the hierarchical components to feed clock ticks accounts for more than half of the algorithm latency. This is represented by the N_{tot} factor of the time complexity. Although other algorithms also need to recursively explore all the hierarchical components, they can optimise this process by assuming that components of the same type behave the same way. For example, the combinatorial cycle detection algorithm can assume that if some component has no cycles then all the components of the same type also have no cycles. Therefore, there is no need to re-analyse them. However, this assumption does not hold for the 'feed clock tick' algorithm. In fact, even if two components are instances of the same hierarchical component, their internal state may be different during the simulation and therefore they have to be treated separately.

This higher algorithm latency may become an issue if the user wants to let a simulation unroll for hundreds or thousands or clock cycles. For example, simulating 100 clock ticks of the CPU would probably take around 3 seconds to complete. Approaches that could be used to speed up the simulation process are described in the next subsection.

8.3.1 Simulator Performance Improvements

In this subsection, I will discuss how the performance of the simulator can be improved. These improvements fall under two main categories: systematic improvements and algorithmic improvements.

First, I will discuss how the technology stack limits the performance, and therefore how we may systematically improve upon it. Unfortunately, though functional programming languages are very expressive and convenient, they tend to suffer in terms of performance. Because of their high-level nature, they rely on garbage collection and do not permit low-level optimizations. Furthermore, it seems that running JavaScript code compiled from F# via Fable leads to further performance loss with respect to running the same F# code directly under .NET [28].

In order to avoid such overheads, one could run the simulator logic directly under .NET using edge.js [20, 19]. In fact, this tool allows to invoke .NET programs from a Node.js application with low overheads.

Another (more radical) idea that can lead to performance improvements is rewriting the performance-critical sections of the simulator in C++ and compiling them to WebAssembly [75, 34]. WebAssembly is a binary instruction format which can be executed on a browser (hence on Electron). This approach would be beneficial for performance for two reasons. First, the browser can run WebAssembly code at near native speed, therefore removing part of the overhead that comes with running web applications [74]. Second, rewriting the code in C++ would allow the developers to implement a series of low-level optimizations not possible in F#. On the other hand, this interaction with WebAssembly may be complex to set up and could affect the maintainability of the application itself.

Alternatively, we may consider algorithmic improvements. For example, one could keep track of what synchronous components have received new signals and only update those during the next clock tick. Unfortunately, it is unclear whether this approach would actually lead to performance improvements, as the cost

of maintaining the tracking data structure may outweigh the benefits of skipping some synchronous components updates. In fact, optimisations that are already in place ensure that only the outputs that changed get propagated into the combinatorial logic, so this optimisation would only save the time of recalculating some unchanged components state.

Also, there exists an interesting optimisation that aims to cut the time taken to explore hierarchical components while searching for synchronous components. In fact, there is no need to search for clocked components within big hierarchical combinatorial components such as an ALU. The information about which hierarchical components do not contain any synchronous logic can be cheaply precalculated. Manually instructing the simulator to not look for synchronous components within the ALU of the RISC CPU cut the 'feed clock tick' latency to around 12 ms. Given its simplicity, this approach seems to be extremely promising, in particular for circuits with large purely combinatorial elements.

Finally, providing more complex builtin components, such as an N-bits adder, will probably speed up the simulation process. Such larger components will have a simulation performance similar to the other basic components such as simple gates. This will be faster than simulating the same digital logic implemented via hierarchical components. On the other hand, this approach can only target commonly used circuits such as the aforementioned N-bits adder.

8.3.2 Summary

In conclusion, the performance of the algorithms seems to be order of magnitude below the threshold of human perception. This guarantees that the application feels highly responsive to the user. Nonetheless, in case a user wants to run lengthy simulations they may experience delays in the range of seconds. Approaches to speed up such a process have been proposed in subsection 8.3.1.

8.4 Code Quality

One of the core focus of the project is on maintainability and extensibility, which tend to be highly correlated with the quality of the code. Objectively assessing such a property is a challenging task, but defining some qualitative metrics can help greatly. In this section, such qualitative metrics will be investigated.

Code Modularisation Code modularisation is fundamental for understandability, extensibility and testability. In fact a well modularised code allows developers to make isolated changes without the risk of breaking unrelated functionalities of the application. As thoroughly described in section 5.1, DEflow codebase is divided among core application logic, user interface logic and testing logic. Each of these macro-modules is internally split into several smaller modules. For example, the user interface logic is split among several independent views, while the core application logic is split between Width Inferer logic and Simulator logic. The latter is further split into several independent components, such as the circuit analyser, the simulator graph builder, the dependency merger and so on.

Overall, DEflow codebase heavily relies on modularisation in order to guarantee maintainability. Modularisation is greatly facilitated by the use of functional programming concepts, which have been extended to the user interface logic thanks to the adoption of Functional Reactive Programming (see section 5.2). In fact, since pure functions are stateless, it is possible to characterise a module just in terms of its inputs and its outputs. Other parts of the codebase can be agnostic of the internal

behaviours of the modules. This encapsulation is critical to the idea of Application Programming Interfaces (API).

API Clarity API are interfaces that define the interaction between the different modules in the application. In order to fully exploit the advantages of code modularisation, each module should expose their capabilities in a clear and intuitive way. In DEflow, this is done at every level of the modularisation hierarchy. For example, a large module such as the Simulator exposes its functionalities via the functions described in table 5. These API strive to be clear and intuitive by making use of descriptive names and explanatory input and output data types.

As mentioned above, the simulator itself is split into several submodules (usually, just an F# file) which expose their functionalities via API and hide their implementation by using private functions (which cannot be accessed from other files). For example the module that analyses the circuit state has a single API function called `analyseState` which takes in input a circuit state and optionally returns an error.

Types Clarity As mentioned in section 3.3.3, one of the main strengths of F# resides in its type system. In order to successfully exploit the benefits of static type checking, the developer must make a good use of type definitions. Additionally, descriptive type definitions greatly help in understanding the modules API.

DEflow types structure closely follows the modularisation of the application. In fact, each module has their own set of types. For example, there exist a set of extra types used in the simulator and a separate set of extra types used in the user interface. Additionally, there exist some types that are used across all modules, such as the type to represent a circuit diagram.

In order to maximise the help that the type checker can provide, I relied on generic types only for functions that have to deal with unspecified types. A series of utility functions, such as inserting an element at a specific index in a list, fall under this category. In the overwhelming majority of cases, DEflow types strive to precisely map the data domain being modeled thanks to the use of F# discriminated unions (DU) and records. Tuples have been generally avoided in favour of records, as the latter support descriptive named fields. Nonetheless, tuples have been used when it proved useful for pattern matching and in the cases where the nature of the tuple elements was clear from the context.

As an example of type definition in DEflow, consider the type `Component`, which represents a circuit component and is defined in `src/Common/Types.fs`. This is a record containing: a string that uniquely identifies the component; the component typology (for example, AND gate or D-flip-flop), which is a DU; a string representing the component label; a list of input ports, where each port is represented by a record type; a list of output ports; and finally two integers representing the x and y coordinates of the component in the diagram. This type precisely maps all the information needed to represent a component.

DEflow precise type definitions allow the compiler to provide very helpful suggestions and error messages. In turn, these allowed me to spot and fix almost every bug before the code was even run. For example, the initial version of the Width Inferer module (which was around 400 lines of code), was entirely bug free before the first execution.

Immutability and Functional Style One of the core aspects of all functional programming languages is data immutability. Using immutable data means that the developer does not have to worry about state maintenance. As discussed in section 3.3.3, F# is actually a hybrid programming language, therefore it also supports mutable data and Object Oriented Programming constructs.

In order to fully benefit from the use of a functional programming language, I tried to avoid mutable data as much as possible. In fact, the core application and testing logic exclusively use immutable data. This did not negatively impact algorithms performance, as explained in section 5.8. Thanks to the use of Functional Reactive Programming, the user interface logic can also almost entirely adhere to this functional style, except for two cases. The first, is a couple of variables in the F# wrapper for the draw2d JavaScript library. Such variables, in fact, have the role of maintaining the mutable state of the circuit diagram object provided by the draw2d library, and therefore they have to be mutable (see section 5.2.1). The second time mutable data was used was in the boilerplate code to set up the Electron application.

Sticking to an almost pure functional style makes the code much more predictable and easy to reason about, as data will not change when passed to functions and functions will have no side-effects. This, in turn, greatly helps maintainability.

Tests Coverage DEflow testing methodology has been thoroughly discussed in section 6. The regression testing infrastructure entirely covers the part of the codebase that can be run under the .NET framework. This includes the core application logic, which represents roughly 49% of the F# codebase (excluding the testing logic itself). Though one may think that a 100% testing coverage may be desirable, it would probably sensibly slow down development. In fact testing the user interface has several drawbacks, which have been described in section 6.2.

In summary, all the algorithmically complex parts of the application are thoroughly tested, which greatly increases the confidence of not breaking functionalities when making changes to the application. On the other hand, the user interface code is intended to be manually tested, since updating the user interface tests at every change would become a very time consuming task.

F# Dependencies DEflow is mostly developed in F# and, as any large software project, it has a series of dependencies. F# dependencies are managed via paket [53], which is the standard package manager for .NET projects. The main F# dependencies are Fable (provides utilities to interoperate with JavaScript), Elmish (to use the MVU architecture) and Fulma (style library for the user interface).

JS Dependencies DEflow is based on Electron, which, in turn, relies on Node.js in order to run JavaScript code outside the Chromium sandbox. JavaScript dependencies are managed by the powerful yarn dependency manager [80]. The main JavaScript dependencies for this project are JQuery (used by the draw2d library), draw2d (which is the graphics library chosen used to draw circuit diagrams), React (which is used as rendering library for the MVU architecture) and webpack (which is the module bundler).

Relying on a series of dependencies is necessary to build large scale projects such as DEflow. DEflow dependencies are all high quality and actively maintained. Particular attention has to be given to draw2d, as DEflow contains some custom extensions to it. Such extensions are extremely unlikely to

break for future versions of the library as they are based on one of the core elements of the library. This means that a change that would break the custom extensions would probably also break the vast majority of other library functionalities, hence it is very unlikely to happen.

Packaging and Deployment Electron allows DEflow to benefit from the advantages of the web ecosystem, such as being cross-platform by design. DEflow has a series of scripts which let the developer to package the application for the three major operating systems with just one terminal command. This, in turn, allows the developer to deploy new versions of the application with minimal effort.

In summary, the metrics indicated in this section seem to prove that DEflow codebase is highly maintainable and extensible. This directly follows from the extensive use of code modularisation, functional style and descriptive data types. Additionally, the large set of regression tests for the algorithmic logic of the application ensure that it will be possible to make changes without unknowingly breaking existing functionalities.

8.5 Extensibility Example

One of the goals of this project is to produce an easily extensible application (requirement D4 from section 4). Arguments about the application extensibility have already been proposed in subsection 8.4. This subsection will demonstrate the extensibility of DEflow by describing the process of adding the 'D-flip-flop with enable' (DFFE) component to the list of builtin components. The code for these changes can be found at `https://github.com/ms8817/DEflow/commit/d0085b0`.

This specific task has been chosen as it touches every section of the application codebase: the JavaScript extensions, the user interface logic, the core application logic and the testing logic. On the other hand, user interface improvements and algorithmic changes generally require modifications to only one or two files, so they are of lower interest here. The process of adding the DFFE is the following:

1. The first step is to add a small amount of extension code to the draw2d library. This extension defines the shape and look of the component in the diagram. In order to allow the user to add an instance of the component to the diagram, the extension must be exposed to the F# code via the draw2d F# wrapper, which involves adding a few trivial lines of code.

2. The second step involves updating the user interface. This allows users to see the DFFE listed in the components catalogue and to see its description and label when selected in the diagram. To do so, the developer has to add a few simple lines in both the catalogue view and selected component view.

3. The third step is to instruct the application on how to deal with the newly defined component. This requires small additions to the Width Inferer and Simulator logic. The Width Inferer must be informed about the widths of the DFFE inputs (two single-bit wires) and outputs (one single-bit wire). The Simulator has to know the behaviour of the component during the simulation, which varies depending on whether the simulation step is a clock tick or not.

4. Finally, a set of tests can be added to test if the new component behaves as expected.

This set of changes (including tests) took me around 25 minutes to complete, and no bugs were introduced. The whole process is heavily guided by the presence of many other existing components, and it should be relatively easy to perform even by a new maintainer. Furthermore, steps 2 and 3 are guided by the F# compiler which detects that the pattern matching case for the new component has not been covered yet, and precisely hints where changes should be made. In the case of the DFFE component, the set of changes adds up to around 90 lines of code in 10 files (excluding tests).

8.6 Application Improvements

As mentioned in section 7.1, users were given the option to leave suggestions for improvements if they wished to do so. This subsection presents a collection of such suggested improvements. This set of improvements has been augmented with my own personal considerations which I collected while comparing DEflow with alternative applications (see 7.2) and while designing the RISC CPU (see 7.3).

The possible improvements are ordered by benefit for the user (highest to lowest) in table 13, with an estimate of the amount of time required for their implementation. Note that the value of the benefit does not reflect the relative importance of the various features (in essence, 'benefit 2' is not twice as beneficial as 'benefit 1') but it does provide a scale to order the features by importance. In order to improve the application as much as I could in the time from the completion of the evaluation to the project deadline, I focused on the high-benefit low-difficulty features (toward the top of the table). Most features are related to the user interface and do not require changes to the algorithmic part of the codebase.

8.7 Reflection on Requirements

A detailed evaluation of the application from several standpoints have been provided in the previous subsections. This subsection will instead summarise the main capabilities of DEflow, to determine whether and to what extent the initial requirements have been met. Each requirement from the requirements capture (section 4) will be referred to using their identifier. Identifiers starting with letter E indicate essential requirements, while those beginning with letter D refer to desirable features.

E1 DEflow includes an interactive circuit diagram interface that allows users to design both combinatorial and synchronous circuits.

E2 DEflow allows users to select and add components to the circuit via a components catalogue. The catalogue contains numerous digital electronics components including gates, flip-flops, registers, ROMs, RAMs, multiplexers and demultiplexers.

E3 Users can use DEflow to package their combinatorial or synchronous circuits into a single component that gets listed in the catalogue. Such a component can then be reused in the following designs.

E4 DEflow supports buses.

E5 DEflow allows users to inject signals in their designs and probe specific wires by using the input and output components, respectively.

Feature description	**Done**	**(Estimated) time required**	**Benefit**
N-bits register	✓	Low	9
D-flip-flop with synchronous enable	✓	Low	
Waveform generator in simulation	✗	High	8
Improve current simulation UI	✓	Medium	7
Multiplexer and Demultiplexer allow buses	✓	Low	
Common keyboard shortcuts	✓	Low	6
Insert components into diagram via Drag and Drop	✗	Medium	
Larger number of properties can be changed from the 'components properties' menu	✓	Low	
Use small endian representation for signal in buses	✓	Low	5
Constant input signals (not settable in simulation)	✗	Low	
Ignore output signal (not shown in simulation)	✗	Low	
End simulation when changing file	✗	Low	4
Show clock tick number in simulation	✗	Low	
Improve simulator performance	✗	Medium	3
N-bits adder (for performance)	✗	Low	
Explicit 'global clock generator' component	✗	Medium	
Library of prebuilt hierarchical components	✗	Medium	
Show list of recently opened projects	✗	Low	2
Autofocus text boxes in created popups	✗	Low	1

Table 13: Possible improvements to the application, ranked by benefit. 'Low' indicates less than 2 hours, 'Medium' less than 5 hours, 'High' more than 5 hours.

E6 DEflow supports an interface that allows users to feed combinatorial inputs and clock ticks into a simulation and visualise the produced outputs. Additionally, users can inspect the state of stateful components such as flip-flops, registers and RAMs at any time during the simulation.

E7 DEflow contains an analysis tool that validates the circuit diagram. Several validations are carried out, such as combinatorial cycle detection and wires width consistency (see section 5.5.2 for details).

E8 DEflow allows users to save and load circuit diagrams. Additionally, the application supports a concept of project, which allows users to easily handle and reuse their designs in the form of hierarchical components.

E9 DEflow will be maintained and extended by an UROP student in the summer 2020. Afterwards, it is likely that groups of third year students will take over its maintenance and extension as part of their High Level Programming course project.

D1 User testing results presented in section 8.1 seems to suggest that DEflow interface is very intuitive and clear. Additionally the analysis performed in section 8.3 indicates that the application is responsive in every scenario except for when the user requires to run lengthy simulations. In such a scenario, they may have to wait a few seconds before being able to visualise the results.

D2 DEflow offers a series of extra features to facilitate the hardware design process, such as auto-routing of wires and automatic bus width inference. Endless possibilities for extensions exist, and the most promising ones will be discussed in the 'Further Work' section (9.1).

D3 DEflow greatly focuses on providing understandable and thorough error messages. Furthermore, it enables the visualisation of such errors directly on the circuit diagram. Users unanimously agreed that such errors were clear and guided them well toward the resolution of their problems (see section 8.1.2).

D4 Since DEflow is mostly implemented in F# and this language is taught in Imperial College third year High Level Programming course, it should be reasonably easy to extend and maintain by Imperial College EE students.

D5 This document, together with the information provided in the project README, guarantee that a developer has enough information to confidently start making changes to DEflow.

D6 DEflow is open-source and free to use for all. Institutions and individuals outside Imperial will be given the opportunity to use and modify the application, but not to use it for commercial objectives. In fact, the application will be distributed with a GPLv3 license [36].

D7 DEflow is deployed as a packaged Electron application. This means that students with devices running either Windows, Linux or MacOS will be able to install the application on their systems. The installation process involves downloading a zipped file and running the pre-built application executable that can be found inside. None of the users that contributed to the feedback collection reported any difficulty in setting up and running DEflow.

In conclusion, DEflow meets all the essential requirements and can therefore be used as a teaching tool in first year Imperial College EE DECA laboratories. Desirable features D3, D4, D5, D6 and D7 have also been

fully implemented. Desirable feature D1 can be considered mostly met, as the application has been found to be intuitive, clear and responsive. Nonetheless, upon running long simulations the user may experience some delays before being able to visualise the results. Approaches to speed up the circuit simulation have been suggested in section 8.3.1. Given the very open ended nature of desirable feature D2, it can hardly be considered fully met. Several helpful circuit design functionalities such as wires auto-routing and automatic bus width inference have been implemented, but many more exist and will be the main subject of section 9.1.

9 Conclusion

In conclusion, this project resulted in a cross-platform hardware design application that (in the context of introductory digital electronics teaching) has been proven to be more usable and intuitive than its alternatives, such as Quartus. This has been ensured by collecting feedback from users which were familiar with Quartus, by performing a manual features comparison against several hardware design applications, and by designing a RISC CPU (see sections 7 and 8).

One of the most challenging parts of the project was setting up a software infrastructure that is easy to maintain and extend in the future. As thoroughly described in section 5, this infrastructure relies on the power of both Functional Reactive Programming and the F# programming language. Setting up this infrastructure required me to familiarise with numerous technologies such as Electron, Node.js, React, MVU architecture and packet managers for both JavaScript and F#. In the end, the choice of technologies for the project proved to be extremely successful. This was demonstrated by the fact that I was able to develop a fully functional hardware design application that is better than existing large scale industrial alternatives in the context of DECA teaching.

The design, implementation and optimisation of several graph algorithms also proved to be a challenging task (see section 5). In fact, though I had experience with implementing sophisticated algorithms before, I never did it in a functional programming language. Implementing performance critical algorithms which are several hundreds lines of code long with immutable data proved to be a very interesting learning experience, which consolidated my functional programming skills. The most algorithmically complex problem was the combinatorial cycle detection with hierarchical components (see end of section 5.7.3). This requires precalculating combinatorial paths via a post-order dependency tree traversal where each tree node is a graph on which an enhanced depth first search is run.

All of the essential requirements described in section 4 have been fully met. Furthermore, as discussed in section 8.7, most of the desirable features for the application have also been implemented. Given more time, the work could be taken further by the addition of the features described in subsection 9.1.

9.1 Further Work

As discussed in section 8.4, the application can be easily extended to support new features. This section aims to determine what extra features could be valuable to add to the application in the future.

Table 13 in section 8.6 presented a list of features that DEflow would benefit from having. Some of these have been implemented before the project deadline, but given the short project timespan it proved to be difficult to add them all. Nonetheless, in order to facilitate the job of a future maintainer, I created some stubs in the code to show how the most complex of those features, waveform generation, can be implemented[6]. These stubs should allow a maintainer which has read this document to have enough guidance to get started extending DEflow functionalities. Furthermore, the modularised nature of the codebase ensures that they can tackle that task without risking to break any other existing functionality.

More advanced functionalities that the application could benefit from having are:

- Define the behaviour of custom combinatorial components using a Verilog subset. This feature requires

[6]The stubs can be found in this branch: `https://github.com/ms8817/DEflow/tree/waveform-stubs`

the application to support a basic text editor. Such an editor can be implemented from scratch, by using a small React based text editor library such as react-simple-code-editor [61], or by wrapping a powerful non-functional code editor library. Though using a small React component is probably the best solution, an example of how to wrap the powerful CodeMirror [11] text editor can be found in an older version of the application[7].

- Synthesise Verilog code from the circuit diagram, which can be then used to program an FPGA.
- Import/export the content of DEflow memories from/to MIFs (Memory Initialisation Files), which are used in other software such as Quartus.
- Export simulation data in a standard file format such as VCD (Value Change Dump) or LTX (inter-Laced eXtensible Trace). These files can be then used to perform a post-mortem simulation visualisation on other tools such as GTKWave [37].

9.2 Final Thoughts

The creation of a hardware design platform was both a highly challenging and rewarding experience. I am extremely satisfied with the outcomes of the project, which gave me the opportunity to display the engineering skills I gained throughout the course of my degree.

[7] It has been removed in commit: `https://github.com/ms8817/DEflow/commit/9d264953`

Appendices

A Guides

This section provides information on how to get started with the application both as users and developers. Users may want to refer to subsection A.1, while developers may refer to subsection A.2.

A.1 User's Guide

The application can be downloaded from `https://github.com/ms8817/DEflow/releases`. The page also contains all the necessary information to run it.

A.2 Developer's Guide

DEflow is a large system with many functionalities, but there exist several improvements and additions that can be made. This section will summarise the most useful resources for a DEflow maintainer.

The DEflow codebase can be found on GitHub at `https://github.com/ms8817/DEflow`. Making a good use of version control is crucial to maintain a clean and structured repository, therefore it is important to be comfortable with such concepts. There exists many guides online to learn these technologies [2], and articles about best practices to follow [71].

Section 5.1 provides a high level overview of the codebase and presents the rationale for the separation of the application code into three main parts: user interface logic, core application logic and testing logic. Each part makes use of a slightly different set of technologies, which is worthwhile familiarising with:

User Interface Logic As explained in section 5.2, the user interface logic is based on the Model-View-Update (MVU) architecture. MVU is an implementation of the Functional Reactive Programming paradigm. Section 3.3.4 gives a comprehensive overview of the concept of Functional Reactive Programming, while section 3.4 provides an example of a basic F# MVU application created with the Elmish library. The Elmish official site [25] has also an informative and short introduction to the library, which is worth reading. Furthermore, as Elmish relies on React for rendering, it may be useful to familiarise with how React works [59]. This will allow developers to have a deeper understanding of the final behaviour of the application user interface, and ultimately achieve better performance.

Additionally, the maintainer should be familiar with how the stateful draw2d library has been wrapped and used in DEflow, which is explained in section 5.2.1. Looking at the already implemented extensions should be enough to get familiar with the library and understanding how to further extend it. In case more details are required, the maintainer may refer to the excellent draw2d documentation [17].

An in-depth knowledge of JavaScript should not be required, as a little amount of code is likely to be written in that language. Furthermore such code will be largely similar to the existing extensions code.

An in-depth knowledge of CSS and HTML is not required either, as the interface is generated via F#. The Fulma documentation [33] will likely be useful as reference.

Core Application Logic The core application logic is implemented in F#, using solely pure functions and immutable data. If algorithmic changes have to be made, it is important to read the related subsection of section 5. For example, for changes to the widths inferrer algorithm, refer to subsection 5.5.1.

Testing Logic Though new tests will be added, the testing infrastructure is very unlikely to require changes. It is important to understand the decisions that have been made for DEflow testing, which are explained in section 6. In order to create a test, it is necessary to design a circuit diagram and to define a series of inputs and expected outputs for it. The circuit diagram can be designed using DEflow itself. Then, the data structure representing the circuit diagram can be logged to the Chromium console and transformed into an F# data structure by one of the scripts provided in the repository. There are over a hundred tests and they all share the same structure, so there should be plenty of guidance for this task.

The repository README contains a detailed guide on how to setup the development environment and install all the necessary dependencies.

The maintainer can refer to section 8.5 for an example of how to add a new component to the application. This example is of particular interest as it touches all the parts of the application codebase.

Additional features that DEflow would benefit from having are presented in section 9.1. The best starting point would probably be adding a waveform visualiser for the simulation, as I already prepared a series of stubs that can guide the implementation of such a feature (see branch 'waveform-stubs' in the repository).

References

[1] Altera Quartus on Mac OSX [article] [last accessed: 15-06-2020]. https://ezcontents.org/altera-quartus-mac-osx.

[2] An Intro to Git and GitHub for Beginners [article] [last accessed: 15-06-2020]. https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners.

[3] Angular vs. React vs. Vue: A performance comparison [article] [last accessed: 15-06-2020]. https://blog.logrocket.com/angular-vs-react-vs-vue-a-performance-comparison/.

[4] Atom: a hackable text editor for the 21st Century [official site] [last accessed: 15-06-2020]. https://atom.io/.

[5] Benchmarks of Cache-Friendly Data Structures in C++ [article] [last accessed: 15-06-2020]. https://tylerayoung.com/2019/01/29/benchmarks-of-cache-friendly-data-structures-in-c/.

[6] BrainBox [GitHub repository] [last accessed: 15-06-2020]. https://github.com/freegroup/brainbox.

[7] Bulma [official site] [last accessed: 15-06-2020]. https://bulma.io/.

[8] Chromium [official site] [last accessed: 15-06-2020]. https://www.chromium.org/Home.

[9] CircuitMaker [official site] [last accessed: 15-06-2020]. https://circuitmaker.com/.

[10] ClojureScript [official site] [last accessed: 15-06-2020]. https://clojurescript.org/.

[11] CodeMirror [official site] [last accessed: 15-06-2020]. https://codemirror.net/index.html.

[12] Compiler from OCaml to Javascript [GitHub repository] [last accessed: 15-06-2020]. https://github.com/ocsigen/js_of_ocaml.

[13] Dart [official site] [last accessed: 15-06-2020]. https://dart.dev/.

[14] Detect Cycle in a Directed Graph [article] [last accessed: 15-06-2020]. https://www.geeksforgeeks.org/detect-cycle-in-a-graph/.

[15] draw2d issue 109 [GitHub issue] [last accessed: 15-06-2020]. https://github.com/freegroup/draw2d/issues/109#issuecomment-636392589.

[16] draw2d issue 115 [GitHub issue] [last accessed: 15-06-2020]. https://github.com/freegroup/draw2d/issues/115.

[17] draw2d.js documentation [documentation] [last accessed: 15-06-2020]. http://www.draw2d.org/draw2d_touch/jsdoc_6/.

[18] draw2d.js [official site] [last accessed: 15-06-2020]. http://www.draw2d.org/draw2d/.

[19] Edge [GitHub repository] [last accessed: 15-06-2020]. https://github.com/tjanczuk/edge.

[20] EDGE.JS: RUN NODE.JS AND .NET IN-PROCESS [official site] [last accessed: 15-06-2020]. http://tjanczuk.github.io/edge/#/.

[21] ELEC40003 (ELEC40003) Digital Electronics Computer Architecture [official site] [last accessed: 15-06-2020]. http://intranet.ee.ic.ac.uk/electricalengineering/eecourses_t4/course_content.asp?c=ELEC40003&s=E1#start.

[22] Electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS [official site] [last accessed: 15-06-2020]. https://www.electronjs.org/.

[23] Electron memory usage compared to other cross-platform frameworks [article] [last accessed: 15-06-2020]. http://roryok.com/blog/2017/08/electron-memory-usage-compared-to-other-cross-platform-frameworks/.

[24] Elm: A delightful language for reliable webapps [official site] [last accessed: 15-06-2020]. https://elm-lang.org/.

[25] Elmish [official site] [last accessed: 15-06-2020]. https://elmish.github.io/elmish/.

[26] Expecto: An advanced testing library for F# [GitHub repository] [last accessed: 15-06-2020]. https://github.com/haf/expecto.

[27] Fable: JavaScript you can be proud of! [official site] [last accessed: 15-06-2020]. https://fable.io/.

[28] Fable JS performance optimization ideas [GitHub issue] [last accessed: 15-06-2020]. https://github.com/fable-compiler/Fable/issues/739.

[29] Fable.Jester [GitHub repository] [last accessed: 15-06-2020]. https://github.com/Shmew/Fable.Jester.

[30] Fabric.js [official site] [last accessed: 15-06-2020]. http://fabricjs.com/.

[31] Femto [official documentation] [last accessed: 15-06-2020]. https://fable.io/blog/Introducing-Femto.html.

[32] FPGAs 5 - Clocks and Global lines [article] [last accessed: 15-06-2020]. https://www.fpga4fun.com/FPGAinfo5.html.

[33] Fulma [official site] [last accessed: 15-06-2020]. https://fulma.github.io/Fulma/.

[34] Getting Started With WebAssembly in Node.js [article] [last accessed: 15-06-2020]. http://thecodebarbarian.com/getting-started-with-webassembly-in-node.js.html.

[35] Ghcjs: Haskell to JavaScript compiler, based on GHC [GitHub repository] [last accessed: 15-06-2020]. https://github.com/ghcjs/ghcjs.

[36] GNU General Public License [official site] [last accessed: 15-06-2020]. https://www.gnu.org/licenses/gpl-3.0.en.html.

[37] GTKWave [official site] [last accessed: 15-06-2020]. http://gtkwave.sourceforge.net.

[38] Implement text editor DOM updates manually instead of via React [GitHub pull request] [last accessed: 15-06-2020]. https://github.com/atom/atom/pull/5624.

[39] Integrate React with other libraries [documentation] [last accessed: 15-06-2020]. https://reactjs.org/docs/integrating-with-other-libraries.html.

[40] InteractiveSVG.js [official site] [last accessed: 15-06-2020]. http://www.petercollingridge.co.uk/tools/interactivesvgjs/.

[41] JavaFX [official site] [last accessed: 15-06-2020]. https://openjfx.io/.

[42] JavaScript: the weird parts [article] [last accessed: 15-06-2020]. https://charlieharvey.org.uk/page/javascript_the_weird_parts.

[43] Jest [GitHub repository] [last accessed: 15-06-2020]. https://github.com/facebook/jest.

[44] jQuery [official site] [last accessed: 15-06-2020]. https://jquery.com/.

[45] jsPlumb [official site] [last accessed: 15-06-2020]. https://jsplumbtoolkit.com/.

[46] KiCad [official site] [last accessed: 15-06-2020]. https://kicad-pcb.org/.

[47] lemonpaul/simulide [GitHub repository] [last accessed: 15-06-2020]. https://github.com/lemonpaul/simulide.

[48] List of C++ 2d graphics libraries [documentation] [last accessed: 15-06-2020]. https://en.cppreference.com/w/cpp/links/libs#Graphics.

[49] MEASURING USABILITY WITH THE SYSTEM USABILITY SCALE [article] [last accessed: 15-06-2020]. https://measuringu.com/sus/.

[50] Moving Atom to React [article] [last accessed: 15-06-2020]. https://blog.atom.io/2014/07/02/moving-atom-to-react.html.

[51] Node.js by numbers [article] [last accessed: 15-06-2020]. https://nodesource.com/node-by-numbers.

[52] Node.js [official site] [last accessed: 15-06-2020]. https://nodejs.org/en/about/.

[53] paket [official site] [last accessed: 15-06-2020]. https://fsprojects.github.io/Paket/.

[54] Proton Native: Create desktop applications through a React syntax, on all platforms [official site] [last accessed: 15-06-2020]. https://proton-native.js.org/#/.

[55] QT [official site] [last accessed: 15-06-2020]. https://www.qt.io/.

[56] Quartus Prime [official site] [last accessed: 15-06-2020]. https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/overview.html.

[57] Quick-Start for Intel Quartus Prime Pro Edition Software [documentation] [last accessed: 15-06-2020]. https://www.intel.com/content/www/us/en/programmable/documentation/myt1400842672009.html#mwh1391807001678.

[58] Raphael [official site] [last accessed: 15-06-2020]. http://raphaeljs.com/.

[59] React - Hello World [official documentation] [last accessed: 15-06-2020]. https://reactjs.org/docs/hello-world.html.

[60] React: A JavaScript library for building user interfaces [official site] [last accessed: 15-06-2020]. https://reactjs.org/.

[61] react-simple-code-editor [GitHub repository] [last accessed: 15-06-2020]. https://github.com/satya164/react-simple-code-editor.

[62] ReaserchGate: Association between percentile ranks, SUS scores, and letter grades [image search] [last accessed: 15-06-2020]. https://www.researchgate.net/figure/Association-between-percentile-ranks-SUS-scores-and-letter-grades-10_fig5_296472190.

[63] Rendering elements - React [official documentation] [last accessed: 15-06-2020]. https://reactjs.org/docs/rendering-elements.html#react-only-updates-whats-necessary.

[64] Running Quartus on an Apple Mac [forum] [last accessed: 15-06-2020]. https://forums.intel.com/s/question/0D50P00003yyGKKSA2/running-quartus-on-an-apple-mac?language=en_US.

[65] SimulIDE blog [forum] [last accessed: 15-06-2020]. https://www.patreon.com/posts/simulide-0-3-12-35657927.

[66] SimulIDE [official site] [last accessed: 15-06-2020]. https://www.simulide.com/.

[67] Slack Desktop [official site] [last accessed: 15-06-2020]. https://slack.com/intl/en-gb/downloads/.

[68] System Usability Scale [article] [last accessed: 15-06-2020]. https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html.

[69] TypeScript: JavaScript that scales [official site] [last accessed: 15-06-2020]. https://www.typescriptlang.org/.

[70] V8 [official site] [last accessed: 15-06-2020]. https://v8.dev/.

[71] Version Control Best Practices [article] [last accessed: 15-06-2020]. https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/best-practices.

[72] Visual Learning Theory [article] [last accessed: 15-06-2020]. http://www.aweoregon.org/research_theory.html.

[73] Visual Studio Code [official site] [last accessed: 15-06-2020]. https://code.visualstudio.com/.

[74] WebAssembly Is Fast: A Real-World Benchmark of WebAssembly vs. ES6 [article] [last accessed: 15-06-2020]. https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193.

[75] WebAssembly [official site] [last accessed: 15-06-2020]. https://webassembly.org/.

[76] Webpack [official site] [last accessed: 15-06-2020]. https://webpack.js.org/.

[77] WhatsApp Desktop [official site] [last accessed: 15-06-2020]. https://www.whatsapp.com/download.

[78] Why does the Intel® Quartus® Prime Pro and Standard Edition version 18.1 fail to run on Ubuntu 18.04 LTS? [article] [last accessed: 15-06-2020]. https://www.intel.com/content/altera-www/global/en_us/index/support/support-resources/knowledge-base/tools/2019/why-does-my-quartus-prime-18-1-fail-to-run-on-ubuntu-18-04-lts-.html.

[79] Wrapping web-components with React [article] [last accessed: 15-06-2020]. https://www.sitepen.com/blog/wrapping-web-components-with-react/.

[80] yarn [official site] [last accessed: 15-06-2020]. https://yarnpkg.com/.

[81] Ieee standard graphic symbols for logic functions (including and incorporating ieee std 91a-1991, supplement to ieee standard graphic symbols for logic functions). *IEEE Std 91a-1991 IEEE Std 91-1984*, pages 1–160, July 1984.

[82] Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.

[83] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190. IEEE, 2011.

[84] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, page 181–186, New York, NY, USA, 1991. Association for Computing Machinery.

[85] P. Cheung. Lecture notes in digital electronics: Lecture 3 - verilog hdl, October 2019.

[86] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 7–18, New York, NY, USA, 2003. Association for Computing Machinery.

[87] E. Czaplicki. Elm : Concurrent frp for functional guis. 2012.

[88] L. M. M. Damas. Type assignment in programming languages (phd thesis). university of edinburgh. 1985.

[89] C. Dony, J. Malenfant, and D. Bardou. Classifying prototype-based programming languages. *Prototype-based Programming: Concepts, Languages and Applications*, 86:71, 1998.

[90] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[91] S. Even and G. Even. Graph algorithms. pages 46–48, 2011.

[92] G. Giorgidze and H. Nilsson. Switched-on yampa: Declarative programming of modular synthesizers. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages*, PADL'08, page 282–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[93] A. D.-L. K. Bhargavan and S. Maffeis. Language-based defenses against untrusted browser origins. 2013.

[94] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[95] I. Pembeci, H. Nilsson, and G. Hager. System presentation - functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02)*, Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02), pages 168–179, dec 2002. Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02) ; Conference date: 06-10-2002 Through 08-10-2002.

[96] J. Raiyn. The role of visual learning in improving students' high-order thinking skills. *Journal of Education and Practice*, 7:115–121, 08 2016.