

Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2022

Project Title: **Combinational Logic Visualisation for ISSIE**
Student: **Aditya Deshpande**
CID: **01504794**
Course: **EIE4**
Project Supervisor: **Dr Thomas J. W. Clarke**
Second Marker: **Mr S. Baig**

Final Report Plagiarism Statement

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

I affirm that I have submitted, or will submit, an identical electronic copy of my final year project to the provided Blackboard module for Plagiarism checking.

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

Acknowledgments

First and foremost, I would like to thank my project supervisor, Dr Thomas Clarke, for his invaluable guidance and support throughout the duration of the Final Year Project, as well as introducing me to the F# programming language in my third year at Imperial College London – something which led me to choose this project.

I am also deeply grateful to my parents. They have always inspired me to reach for new heights, while always supporting me so that I can do so without fear of falling.

Finally, I would also like extend my thanks to all previous Issie developers for contributing to this wonderful application and making it a solid platform for me to build on top of.

Abstract

Issie (Interactive Schematic Simulator with Integrated Editor) is an education-focused digital electronics design platform used by students at Imperial College London, featuring an intuitive user-friendly UI and capable logic simulator. One of the many challenges a student may face when learning to design digital logic is conceptualising relationships between inputs and outputs in combinational logic, and how they relate to design specifications. This project extends the existing Issie application, exploring novel ways to communicate these logic relationships to the user. This is primarily achieved through the implementation of interactive automatic schematic-derived truth tables, which can be manipulated, reduced, and filtered to describe combinational logic relationships in numeric and algebraic forms. This involved the definition of an alternative formal language to Boolean algebra for representing combinational logic.

Additionally, this project has also made changes to the top-level UI of the application to make the user experience more consistent and conducive to learning. The extended functionality delivered by the project is effective and performant; this has been confirmed through a user experience survey. The novel visualisation methods have been tested on circuits designed by EE students at Imperial College London. Large digital logic circuits, such as the Arithmetic Logic Unit of an 8-bit ARM CPU, can be condensed from a schematic containing billions of possible input combinations to a short algebraic truth table describing a few dozen cases.

Contents

1	Introduction	1
1.1	Project Motivation	1
1.2	Project Definition	2
1.2.1	Core Principles of Issie	2
2	Background	4
2.1	Pedagogical Considerations	4
2.1.1	Memory Models	4
2.1.2	Educating Engineers	5
2.1.3	Cognitive Theories of Self Efficacy and Constructivism	5
2.2	Combinational Logic	6
2.2.1	Visualising Logic with Boolean Algebra	6
2.2.2	Visualising Logic with Schematic Diagrams	6
2.3	Truth Tables	6
2.3.1	Reduction using Don't Cares	7
2.3.2	Algebraic Truth Tables	8
2.4	An Overview of Issie's Technology Stack	9
2.4.1	Programming Language	9
2.4.2	Ecosystem	10
2.4.3	User Interface and Rendering	11
2.4.4	Overview	14
2.5	Issie's UI	14
2.5.1	Overview and Evaluation	14
2.5.2	Considerations when adding new features	16
2.6	Combinational Logic Simulation in Issie	16
2.7	Effects of Application Performance on Users	16
2.8	Agile Software Development	17
3	Project Requirements	19
3.1	Requirements for Combinational Logic Visualisation	19
3.2	Software/Documentation Quality Requirements	20
4	Analysis and Design	21
4.1	Approach towards Software Development	21
4.2	Technology Stack	21
4.3	Top Level UI Changes	22
4.4	Generating Numeric Truth Tables	22
4.4.1	Decision to re-use Step Simulation Code	24
4.4.2	Evolution of the Input Space Generation Algorithm	24
4.5	Generating Truth Tables for a partial selections	26
4.6	Filtering with Constraints	26
4.6.1	Constraint Validation Rules	27
4.7	Hiding Output Columns	27
4.8	Graphical Manipulation of Truth Tables	28
4.9	Don't Care Reduction	28
4.10	Algebraic Truth Tables	28

4.10.1	Designing the System for Algebra	29
4.11	Considering Logic Input with Truth Tables	30
5	Implementation	32
5.1	Overview of Data Types	32
5.1.1	Types for Representing Truth Tables	32
5.1.2	Constraint Types	33
5.1.3	Algebra Types	34
5.1.4	The <code>TableInput</code> Data Type	35
5.1.5	Table Manipulation Data Types	35
5.2	Top Level UI Changes	36
5.2.1	Simulation Sub-tabs	36
5.2.2	Moving the Waveform Simulator	36
5.2.3	Dynamic Dividerbar Resizing	37
5.3	Generating Truth Tables	37
5.3.1	Building the Simulation	37
5.3.2	Calculating the Input Space	37
5.3.3	Simulating each Combination	40
5.4	Generating Truth Tables for a partial selection of a Sheet	40
5.4.1	Canvas Correction	40
5.4.2	Caching Strategy	41
5.5	Filtering with Output Constraints	42
5.6	Don't Care Reduction	43
5.6.1	Prerequisite Concepts	44
5.6.2	Reduction Algorithm	44
5.7	Algebraic Truth Tables	46
5.7.1	Definition of the Algebraic System	46
5.7.2	Reduction Rules	49
5.7.3	Implementing Algebra	53
5.8	Sorting Truth Tables	56
5.8.1	Comparison Function for <code>CellData</code>	57
5.9	Truth Table Caching and Order of Operations	57
5.10	Rendering the Truth Table	59
5.10.1	Method 1: Using Fulma Tables	59
5.10.2	Method 2: Using CSS Grids	60
5.11	Column-based Operations	61
5.11.1	Moving Columns	61
5.11.2	Hiding Columns	61
6	Testing and Results	63
6.1	Testing Application Stability	63
6.1.1	Exception Analysis	63
6.1.2	Failure Analysis	64
6.2	Correctness Testing	64
6.3	Quantitative Performance Testing	66
6.3.1	Generating Numeric Truth Tables	67
6.3.2	Algebraic Truth Tables	67
6.3.3	Don't Care Reduction	67
6.3.4	Graphical Manipulation of Truth Tables	67
6.4	User Experience Testing	68
6.4.1	Stage 1: Mystery Sheets	68
6.4.2	Stage 2: Questionnaire	68
6.5	Testing on the 8-bit ALU	71
7	Evaluation	73
7.1	Evaluation against Project Aims	73
7.2	Evaluation against Issie's Core Principles	74
7.2.1	Robustness	74
7.2.2	Obviousness	75

7.2.3	Intuitiveness	75
7.3	Evaluation of Application Performance	76
7.4	Comparison Against Issie 3.0.0	76
7.5	Evaluation of Filtering Methods	77
7.6	Evaluation of Reduction Methods	78
7.6.1	Algebraic Truth Tables	78
7.6.2	Don't Care Reduction	78
7.7	Evaluation against Requirements	79
7.8	Summary	83
8	Conclusion and Further Work	84
8.1	Possible Further Work	85
8.1.1	Algebraic Output Constraints	85
8.1.2	More Algebraic Reduction Rules	85
8.1.3	Adding Algebra to the Step Simulator	85
8.2	User Guide	85
A	Screenshots of Issie 3.0.0	89
B	Work Breakdown Structure	92
C	8-bit ALU Schematic	94

List of Tables

2.1	Truth Table for the Boolean AND operator ($C = A.B$)	7
2.2	Truth Tables for a 2-bit Multiplexer	8
4.1	Arguments for and against implementing truth-table defined custom components	31
5.1	Data Types and Structures used for Truth Table Representation	32
5.2	Explanation of <code>TruthTable</code> Record fields	33
5.3	Explanation of Fields in the <code>TableInput</code> data structure	36
5.4	Average Truth Table Generation and View times with varying Bit Limits	38
5.5	Time taken by each method to generate a numeric truth table for an 8-bit ALU	39
5.6	Terminal Value Symbols	47
5.7	Terminal Operator Symbols	47
5.8	Non-Terminal Symbols	48
5.9	New additions to the defined behaviour for each component in <code>fastReduce</code>	55
5.10	Summary of operations which change the <code>TruthTable</code> data structure	58
5.11	Time taken to move a column in a Truth Table under different methods (ms)	60
6.1	Library functions in project code which can throw exceptions	64
2	List of all features which were manually tested	65
6.3	Time taken to generate a numeric truth table (ms)	67
6.4	Time taken to generate an algebraic truth table for the ALU schematic (ms)	67
6.5	Time taken to reduce a numeric truth table using Don't Cares (ms)	67
6.6	Time taken to conduct different graphical manipulations on a truth table (ms)	67
6.7	Mystery Sheets	69
6.8	First set of questions asked in questionnaire	70
1	Evaluation against Requirements	80

List of Figures

2.1	The Atkinson-Shiffrin Multi-Store Model	4
2.2	Algebraic Truth Table in Datasheet for a Three-input Multiplexer [17]	8
2.3	An example of incomplete Pattern Matching in F# , with a warning from the compiler	9
2.4	How browsers traditionally render HTML using a DOM [33]	12
2.5	The Elm Architecture [33]	13
2.6	Fulma Table Example	14
2.7	An Overview of Issie’s Technology Stack	14
2.8	An overview of Issie’s Step Simulation	17
4.1	View of Issie’s updated Top-Level UI	22
4.2	Contents of Truth Table tab after a truth table has been generated	23
4.3	High-Level view of Truth Table Generation	24
4.4	Adding a Constraint to the Truth Table	27
4.5	Algebra Input Selector popup	29
5.1	Circuit with multiple <i>MergeWires</i> connected to each other	35
5.2	Selection Cases	42
5.3	Representation of the Canvas at different stages of Canvas Correction	42
5.4	Method for Filtering the Truth Table with Output Constraints	43
5.5	Schematic and Reduced Truth Table	44
5.6	Splitting and Merging	49
5.7	Circuit for Bitwise And	50
5.8	Boolean Algebra Identities [54]	51
5.9	Adder Circuits	53
5.10	Annotated Schematic showing propagation of Algebra	54
5.11	Process for Arithmetic Simplification with Example	56
5.12	How operations update the Truth Table	59
6.1	Graph showing the percentage of participants which discovered each feature and found it useful	70
6.2	First 18 rows of the Algebraic Truth Table for 8-bit ALU	72
A.1	Issie’s opening screen	89
A.2	A sheet (schematic) open in Issie, with a component and wire selected	90
A.3	Issie’s Waveform Simulator selection menu	90
A.4	Issie’s Waveform Simulator	91
B.1	Work Breakdown Structure with ordered backlog	93
C.1	Schematic Diagram for an 8-bit ARM ALU	95

Table of Acronyms

Acronym	Definition
ALU	Arithmetic Logic Unit
CLR	Common Language Runtime
CSS	Cascading Style Sheets
DC	Don't Care
DOM	Document Object Model
GUI	Graphical User Interface
HTML	Hypertext Markup Language
ISSIE	Interactive Schematic Simulator with Integrated Editor
MVU	Model View Update
UI	User Interface
UWP	Universal Windows Platform
WBS	Work Breakdown Structure
WPF	Windows Presentation Foundation

Chapter 1

Introduction

1.1 Project Motivation

Digital Electronics and circuit design are core fields in the study of Electronic Engineering and are focused on the analysis and logical interpretation of digital signals, as well as the engineering of hardware that manipulates them in accordance with a desired logical function. A strong understanding of the fundamentals of digital electronics and circuit design serve as a foundation for multiple branches of study within Electronic Engineering. Therefore, it is vital that undergraduate students at Imperial College and other institutions have the best tools available to aid their study of these fundamental concepts. One of the many challenges a first-year undergraduate student may face while learning Digital Electronic design is conceptualising relationships between inputs and outputs, and how these relationships relate to design specifications. At Imperial College London, EE students have the opportunity to gain a deeper insight into combinational logic through practical laboratory sessions, during which they create and simulate combinational circuits. The tools which they use must fulfil two criteria; firstly they must provide an education-focused platform through which students can learn more about combinational logic and hardware design; secondly they must be capable design tools in their own right which allow students to design and simulate complex logic.

Issie (Interactive Schematic Simulator with Integrated Editor) [1], an intuitive hardware design application, was developed at Imperial College London to address the lack of third-party programs that matched the above criteria. Issie is designed to be easy to use (requiring no user manual) and informative; visual cues and clear error messages guide students towards correct designs while individual step and waveform simulators allow students to vary inputs and see the effect on output values. This allows them to gain a better understanding of the hardware logic they have created. However, there is room for improvement. In its current form, users of the application implement digital logic by building it component-by-component on a schematic diagram. Any syntactically correct digital circuit can be simulated using the *Step Simulator*. In the Step Simulator, users specify values for each input to the digital logic, and can read the corresponding output values. Intermediate values can also be observed using *Viewer* components. This functionality enables the user to easily verify their schematic with specific test cases, but lacks the ability to clearly summarise and verify the overall relationship between the inputs and outputs of the logic circuit. Users must therefore gain an overall understanding of the circuit through a combination of:

1. Visually analysing the schematic to understand its logical function.
2. Entering different input combinations into the Step Simulator and analysing the effect each change has on the outputs.

As the implemented digital logic grows in complexity, the relationship between the inputs and outputs often becomes more obscure, and the schematic itself grows in size and can start to feel divorced from the specification. In such situations, the aforementioned method for understanding the logic circuit becomes less effective. To stop the schematic from getting too large and crowded Issie lets users define hierarchical *Custom Components* which modularise the schematic and cut down on logic duplication. For example, an ALU may be implemented as a Custom Component

within a CPU design schematic. This feature however, does not fully combat the issue of obscure relationships between inputs and outputs for complex circuits. Firstly, custom components that are not named clearly further obscure the logic function of the circuit. Secondly, due to their hierarchical nature, custom components can be nested inside other custom components, meaning that the user may have explore multiple layers of nested components before they can analyse the top-level schematic. This is a time-consuming exercise, requiring significant effort by the user. Therefore, there is significant value to adding functionality to Issie which allows users to better understand the relationships between inputs and outputs in digital logic circuits in a shorter amount of time.

One possible solution to this problem is automatically generating truth tables from the schematic. Truth tables exhaustively show the relationship between all inputs and outputs in an organised, persistent format. Inspecting cases in a truth table is far quicker than repeatedly changing values in the Step Simulator. Furthermore, by investigating novel ways of presenting and interacting with these truth tables their value addition to the learning and circuit design experience in Issie can be boosted. For example, the ability to present relationships inferred from the schematic in the truth table, or reduce an existing truth table with user-defined constraints, would provide the user with far more information than a simple simulation.

There is also merit in investigating the reverse; generating schematics from user-entered truth tables. This could reduce time spent designing hardware components which implement simple logic but require many gates and connections, as well as serve as a stepping stone between schematic design and HDL-based design.

Thus, there is a strong case for finding and implementing alternative ways to visualise (and possibly input) combinational logic in Issie to enable users to gain a better understanding of relationships in the logic, as well as the specification of the top-level design. If added in a way which complements Issie's existing features, such enhancements are likely to increase Issie's effectiveness as an educational platform in addition to its capability as a digital logic design tool. This will benefit students at Imperial College and other educational institutions.

1.2 Project Definition

The purpose of this project is to explore novel ways in which interactivity can be added to automatic schematic-derived truth tables, and how interactively generated truth tables can be used as a fast aid to design combinational logic. This is to achieve the overall aim of this project - to improve Issie in such a way that it is easier for students to understand the use of combinational logic in digital design. The deliverable will be integrated as an extension to the Issie application, with users being able to generate truth tables from the schematic and interact with them in ways that will augment their understanding of the logic they are designing and of Digital Electronics concepts in general. This project will conduct a short evaluation of Issie, highlighting the areas where it can be improved. While the primary focus of the project is on visualising combinational logic with interactive truth tables, the project will also seek to improve the overall user experience of Issie in other ways such as tweaking/redesigning elements of the UI or changing how information is communicated to users such that it is consistent and clear. In addition to improving the user experience for Issie, this project also aims to improve the developer experience wherever possible. Since its inception, maintainability and extensibility have been key to Issie [2]; therefore the code contributed to the Issie repository should be well-documented, readable, and interface well with existing code so that it is easy for future Issie developers to maintain and extend it. Further to this, if an appropriate opportunity arises, the project should also aim to reduce technical debt within the existing codebase.

1.2.1 Core Principles of Issie

As this project aims to improve Issie, any work done on this project should align with Issie's core principles. All features implemented in Issie must be:

1. **Robust:** Software is robust when it is able to handle errors and behave correctly under exceptional circumstances, such as when supplied with erroneous inputs [3]. For simulations

and text field inputs, Issie notifies the user of the nature of the error. User input, no matter how malformed, must never crash the application or lead to undefined behaviour.

2. **Obvious:** The visual output given to the user should make it obvious what is happening without the need for unnecessary explanation. Issie prefers to *show not tell* in order to remain beginner-friendly. For example, Issie uses colour-coded popups and highlights to draw user attention where it is needed and communicate events clearly.
3. **Intuitive:** All functionality must be easy to expose to the user - there should be no need for detailed user guides as the UI for all functionality must be designed in a way such that users can intuitively learn how to use all of the application's features.

In addition to these core principles, any extensions this project makes to Issie must also take into account the targeted users and the intended primary use case - teaching undergraduate students in a university laboratory while also enabling students to carry on where they left off at home. Thus, all new features must be cross-platform compatible and be suitable for students working in a laboratory and working alone at home.

In conclusion, this project has two final deliverables. The first is an improved version of Issie, while the second deliverable consists of appropriate documentation of added features, and improvements to the documentation of existing features.

Chapter 2

Background

This chapter describes various theoretical concepts which provide context to many of the decisions made throughout the duration of the project. It also describes and evaluates the current version of Issie, analysing its strengths and weaknesses. Given the overall project goal of improving combinatorial logic visualisation in Issie, this analysis provides context for the extensions made to Issie by the project, which are described in subsequent chapters.

2.1 Pedagogical Considerations

Given the overall project aim of making it easier for students to understand the use of logic in digital design, any features added to Issie must enhance the learning experience of its users. Many decisions related to the project, such as which features to add, the UI/UX design, and the level of interactivity will all be made through the lens of pedagogy.

2.1.1 Memory Models

One of the key facets of learning is the long-term retention of key concepts and relationships. The Atkinson-Shiffrin multi-store model [4] provides a good framework for modelling the workings of human memory, and many task-specific models, theories, and techniques have been derived from it. The aim of any effective learning application should be to convey and revisit information in such a way that it succeeds in reaching the Long Term Memory store.

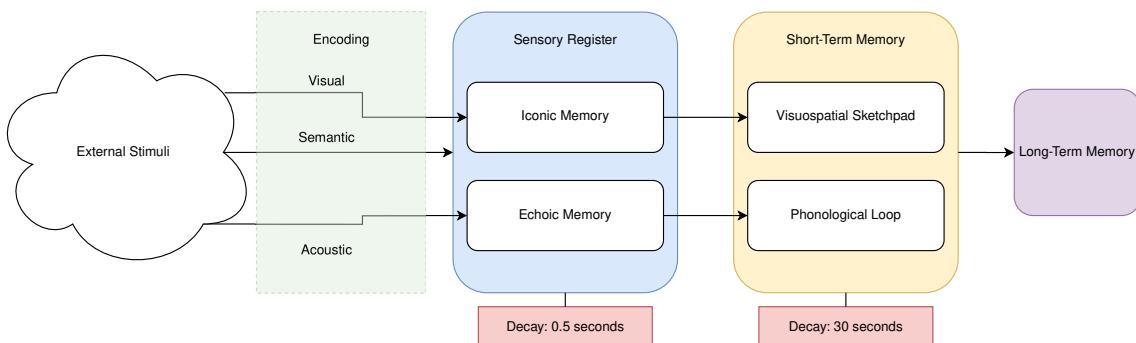


Figure 2.1: The Atkinson-Shiffrin Multi-Store Model

As shown in Figure 2.1, the Atkinson-Shiffrin Multi-Store Model states that humans perceive information through external stimuli, and that information is encoded as three different types of stimulus. **Visual** encoding encodes information in pictures and diagrams, **Semantic** encoding encodes information in words and their meanings, while **Acoustic** encoding encodes information in sounds. This encoded information is processed by the sensory register. If attention is paid to this information, it travels into short-term memory, otherwise it decays within half a second and is forgotten. For information to progress from short-term to long-term memory, a process

of repetition called maintenance rehearsal is necessary [5], in which the information is repeated within the learner's mind. For visually encoded information, this could be through repeated visualisation of diagrams or patterns in what the learner sees on a screen, while for acoustically encoded information this is achieved through the learner playing back the soundbite in their head (phonological loop). A key take-away from this memory model is that human brains process different information encodings independently, meaning that even if one encoding type is saturated with information, additional information can still be conveyed to the learner using an alternative encoding method.

In its current form, Issie encodes information about the digital circuit both visually (circuit diagram, error highlighting etc.) and semantically (error messages, simulation outputs). No information is encoded acoustically, however outside of chimes for errors there is not much scope for implementation of sounds. Furthermore, students will receive plenty of auditory information in the form of teaching from lab assistants and conversation with their peers. Any extensions added to Issie by this project should therefore focus on conveying information to users through a combination of diagrams with appropriate annotations and informative text.

2.1.2 Educating Engineers

As a tool for teaching Digital Electronics, Issie's primary user base is engineering students. Therefore, there is value in exploring trends in how engineers learn best, and tailoring Issie's design and structure to align with these trends. In 1992, Fleming and Mills proposed that there were four categories of how students learned. These are **visual**, **auditory**, **reading/writing**, and **kinesthetic** [6]. Most learners will learn using all of these methods, however will exhibit a preference towards one or two. Studies have shown that engineers have a preference towards visual and kinesthetic learning techniques [7]. The visual aspect means that engineers prefer information be conveyed to them through diagrams, patterns, and highlighted meaningful symbols. On the other hand, the kinesthetic aspect means that engineers prefer to learn through demonstrations, simulations, and their own experiences. The existence of practical lab sessions and Issie itself lends itself to a focus on kinesthetic learning; students explore concepts they have been taught about in lectures by building and simulating combinational logic. Therefore, the results of the VARK survey concur with the current teaching style in the EEE Department at Imperial College. Currently, Issie has an interactive diagram with highlighting (visual encoding), as well as simulators for testing logic and general experimentation (kinesthetic learning). Thus, it can be said that Issie in its current form is fit for its purpose: educating engineers. In turn, any extensions added to Issie by this project should continue this trend of visual and kinesthetic learning, but while also taking care to not overload the users' sensory register and short-term memory.

2.1.3 Cognitive Theories of Self Efficacy and Constructivism

There are more factors that contribute towards an effective learning experience than just the conveyance of information. The learning process must be structured in such a way that students remain motivated while learning complex concepts. A ten-year longitudinal study [8] found that there is a significant correlation between students who are motivated, therefore having high self-confidence, and academic attainment. Therefore, while developing educational tools like Issie, an emphasis must be placed on communicating information to students in such a way that they retain the belief that they can successfully learn the content. This approach aligns well with the theory of constructivism in education, which seeks to educate students by having them discover knowledge intuitively in contrast to traditional methods in which a student is considered an 'empty vessel' waiting to be filled up by a teacher [9]. The rationale behind the theory is that this self-discovery of knowledge will build a stronger conceptual understanding of what is being learnt. Constructivism aligns well with teaching styles that suit kinesthetic learners, as both approaches focus on students learning through their own experiences. Engineers tend to be kinesthetic learners, therefore, this project should aim to improve Issie in such a way that students are able to interactively and iteratively build their understanding of digital electronics and circuit design by themselves.

2.2 Combinational Logic

Combinational logic is a type of digital logic in which the output of the logic is a pure function of its present inputs [10]. This means that combinational logic is memoryless; it is not affected by any previous outputs. This is in contrast to sequential logic, for which the output is dependent on present input values and some internal state derived from previous outputs.

2.2.1 Visualising Logic with Boolean Algebra

The aforementioned pure function which maps inputs to outputs can be written as a Boolean expression. A Boolean expression, much like traditional mathematical expression, features a set of input terms combined using operators. There exist three fundamental Boolean operators [11]:

NOT, a unary operator which outputs the inverse of the input.

AND, a binary operator which outputs HIGH when both inputs are HIGH and LOW otherwise. Denoted by

OR, a binary operator which outputs HIGH when either of the inputs are HIGH, and LOW otherwise

Other Boolean operators, such as **NAND**, **NOR**, and **XOR** also exist, but can be defined using a combination of the three fundamental operators. Therefore, the first and most basic way of visualising combinational logic is simply through writing the Boolean expression which represents it. However, it can be difficult to quickly understand what some logic does using just Boolean expressions. Take for example the following Boolean equation for output Y , derived from inputs S , A , and B :

$$Y = \overline{S} \cdot A + S \cdot B \quad (2.1)$$

While the operations performed are clear, it may not be very clear on first inspection to an inexperienced student what the expression actually achieves. Furthermore, Boolean expressions grow in complexity as schematics become larger, making them even tougher to understand at a glance.

2.2.2 Visualising Logic with Schematic Diagrams

Schematic diagrams give the hardware representation of the combinational logic, and are the primary way of creating and visualising combinational logic in Issie. At their simplest, they consist purely of logic gates; these gates each correspond to a Boolean operator. However, in practice (and Issie) schematics are at a slightly higher-level, with certain combinational components (which can be built from gates) pre-created for the user. For example, Equation 2.1 represents a 2-bit Multiplexer, which is a component available in Issie's catalogue. The recall of stored knowledge due to the visual stimulus of the multiplexer component on the diagram is much more likely compared to the semantic stimulus of the Boolean expression. However, as a schematic increases in size, the number of components may become so large that holding all of the visual stimuli in short-term memory is unfeasible. Issie combats this by letting users modularise their schematic through hierarchical custom components. Alongside decreasing schematic size and repeated logic, this feature actually aims to teach students the technique of modularising their work (whether that is a schematic or code), and its advantages. These advantages [12] include efficient development, easier debugging, and logic reuse. However, as mentioned in the Project Motivation, hierarchical components are not always effective (particularly if badly named and organised), leaving a opportunity for improvement through complementary visualisation techniques.

2.3 Truth Tables

A truth table represents a given combinational logic function; featuring all input combinations on the left, and their corresponding output(s) on the right. As the truth table maps all possible inputs to their output, it is trivial to look up the behaviour of the logic in a given scenario. A very basic example is the truth table for the Boolean **AND** operator, shown in Table 2.1.

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1: Truth Table for the Boolean AND operator ($C = A \cdot B$)

Given that a truth table defines some logic using an exhaustive set of examples, it could be said that truth tables are ideal for kinesthetic learners. This exhaustive property can also be used to test for logical equivalence. Suppose the claim is made that two schematics, with one schematic featuring far fewer components than the other, are logically identical. Equivalence could be confirmed by simply checking if the truth tables for the two schematics are the same.

However, a disadvantage of using truth tables to visualise combinational logic is that they can very rapidly grow in size. For logic with n single-bit inputs, the number of rows in the associated truth table is $2^n - 1$. For multi-bit inputs this number would grow even larger. Thus, Issie schematics with a large number of inputs would result in very long truth tables, which would likely intimidate the user. The size of generated truth tables could be reduced by filtering them based on user selections, or through truth table reduction methods.

2.3.1 Reduction using Don't Cares

A "Don't Care" term in a truth table can mean different things based on its positioning. It is more commonly seen on the right-hand side of a truth table, signifying that an output for a given input combination is invalid or has no use [13]. This information is often used when attempting to simplify Boolean expressions using Karnaugh maps. On the other hand, a Don't Care term in an input row signifies that the particular input has no effect on the eventual output of the logic for that combination. These *Don't Care inputs* can be found through logic minimisation, of which there are numerous techniques ranging from the aforementioned Karnaugh Maps [14] to heuristic-based tools like Espresso [15]. The latter is very effective at reducing large circuits efficiently, and would therefore appear to fit the needs of the project well. An example of the reduction can be seen in Table 2.2, where the eight row exhaustive truth table (2.2a) is reduced to two rows (2.2c). However, Espresso and other common minimisation techniques treat their inputs and outputs as single-bit Boolean values which are either ON, OFF, or DC – corresponding to 1, 0 and Don't Care [16]. Multi-bit IOs are broken into their constituent bits. In Table 2.2c, only HIGH outputs are shown – this works when it can be assumed that all other input combinations yield 0. This approach aligns well with industrial applications, where signals on wires can only be HIGH or LOW, and there is value in reducing to the form which requires the fewest gates. On the other hand, Issie is an educational application where inputs and outputs can have more than two values, and focus is more on semantics and understanding rather than the most cost-efficient hardware design. Therefore, such minimisation would not integrate well with Issie's existing implementation of multi-bit IOs.

In order to implement DC reduction in Issie, a suitable algorithm will have to be written which supports multi-bit inputs. One possibility may be analysis of redundancies in the existing truth table. For example, in the full truth table, the first and third row both have an output value of 0, with the only difference being the change in the value of B . Given that B can only take two values, this shows that when $A = 0$ and $S = 0$, the value of B does not affect the output - therefore we "don't care" about B . Rows one and three can therefore be collapsed into one row, with the entry for the B column being replaced with an "X". This process can be repeated until all such row combinations have been collapsed. The results from this process can be seen in Table 2.2b. The length of the truth table has been reduced by 25%, and the actual semantic function of what the multiplexer does is much clearer as well. This table is however, much larger than the reduced table generated by Espresso, indicating that it may not work well with larger schematics.

2.3.2 Algebraic Truth Tables

While reduction with Don't Cares is useful, neither implementation of it is ideal. Industry-style minimisation doesn't fully align with Issie's implementation, while custom algorithms may not be able to reduce the table enough. Additionally, Don't Care reduction cannot simplify relationships which involve all inputs, such as arithmetic. A viable alternative is an Algebraic Truth Table; these are often found on component datasheets [17] and have the task of summarising the behaviour of the circuit in a concise and readable format. One such example is shown in Figure 2.2, which is an excerpt from a datasheet for a three-input multiplexer. H and L are equivalent to 1 and 0, but the terms of form I_x in the table are algebraic values representing inputs. The select signals (S_2, S_1, S_0), which actually control the circuit behaviour are still numeric.

TRUTH TABLE						
INPUTS			OUTPUTS			
SELECT			OUTPUT ENABLE \overline{OE}	Y	\bar{Y}	
S_2	S_1	S_0				
X	X	X	H	Z	Z	
L	L	L	L	I_0	\bar{I}_0	
L	L	H	L	I_1	\bar{I}_1	
L	H	L	L	I_2	\bar{I}_2	
L	H	H	L	I_3	\bar{I}_3	
H	L	L	L	I_4	\bar{I}_4	
H	L	H	L	I_5	\bar{I}_5	
H	H	L	L	I_6	\bar{I}_6	
H	H	H	L	I_7	\bar{I}_7	

H = High logic level, L = Low logic level, Z = High impedance (off),
X = Irrelevant, $I_0, I_1 \dots I_7$ = The level of the respective input

Figure 2.2: Algebraic Truth Table in Datasheet for a Three-input Multiplexer [17]

Table 2.2d shows what the corresponding algebraic truth table would look like for the running example in this section. A is propagated to the output when the selection input (S) is low, and B is propagated when S is high. This clearly describes a two-input multiplexer. Having functionality which could create similar tables for user-created schematics, with support for more complex algebraic operators would likely be useful. This is because algebraic truth tables carry far more semantic information in a much smaller visual space, meaning that a students' sensory register is less likely to be overloaded.

A	B	S	OUT												
0	0	0	0	0	X	0	0	1	X	0	1	1	X	0	1
0	0	1	0	0	1	1	0	0	0	1	1	0	0	X	1
0	1	0	0	1	0	1	1	1	1	0	1	1	1	1	0
0	1	1	1	X	0	1	0	0	0	X	0	0	0	0	X
1	0	0	1	X	1	1	1	1	1	X	0	1	1	1	1
1	0	1	0	1	X	0	1	1	1	0	X	1	1	1	1
1	1	0	1	0	0	X	0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	X	1	1	1	1	1	1	1	1	1

(a) Full Truth Table (b) DC Reduced
(Inc. Zeros) (c) DC Reduced
(Espresso) (d) Algebraic Truth Table

Table 2.2: Truth Tables for a 2-bit Multiplexer

2.4 An Overview of Issie's Technology Stack

The reasoning and process behind the decisions made for Issie's technology stack can be found in Marco Selvatici's dissertation on DEFlow, the predecessor to Issie [2]. This section describes the technology stack, and evaluates and reaffirms why Issie's technology stack is well suited.

2.4.1 Programming Language

Issie is written in F# , an open-source, cross-platform, interoperable programming language for writing succinct, robust and performant code [18]. It is functional-first; meaning that it contains many features found in functional languages and encourages a functional programming style while also allowing programmers the flexibility to use the programming styles of other paradigms. Pure functional programming languages adopt a philosophy of declarative programming and immutable data; data values cannot be updated after initial assignment, and functions take this data as input and map it to output data. These functions are deterministic, meaning that their output is solely dependent on the value of their inputs and that there is no internal state affecting the behaviour of the function. This differs from the more common imperative style of programming where code is treated as a sequential list of instructions which mutate program data/state. The deterministic nature of functions in pure functional programming makes them very easy to understand, as operations on data have no side-effects and are therefore very easy to track. Not only does this make debugging easier, but it leads to fewer overall bugs in the code. A large-scale study of programming languages and code quality on Github [19] found that "Functional languages have a smaller relationship to defects than other language classes, whereas procedural languages are either greater than average or similar to the average." The study also found that the "Functional-Static-Strong-Managed" class of languages (i.e. languages that are functional, statically and strongly typed with in-built memory management) are less likely than average to result in defect-fixing commits on Github. While not a part of the study, F# does belong to this class of languages, and is therefore a wise choice of programming language for Issie. F# features a Hindley-Milner type system, which has a provably sound type inference algorithm [20]. Type inference allows for F# to be statically typed while eliminating the need for type annotations in the code. This results in clean-looking and succinct code while still maintaining the benefits of static typing. Furthermore, as types can be inferred on the fly by IDEs such as Visual Studio, it is much easier for the programmer to track the correctness of the program.

```
type RainbowColor =
| Red
| Orange
| Yellow
| Green
| Blue
| Indigo
| Violet

let printColor color =
    match color with
    | RainbowColor.Red -> printfn "Red"
    | RainbowColor.Orange -> printfn "Orange"
    | RainbowColor.Yellow -> printfn "Yellow"
    | RainbowColor.Blue -> printfn "Blue"
    | RainbowColor.Indigo -> printfn "Indigo"
    | RainbowColor.Violet -> printfn "Violet"
```

Figure 2.3: An example of incomplete Pattern Matching in F# , with a warning from the compiler

Such hints are immensely useful for the programmer. The advantage of such checks being performed before and at compile-time is that it decreases the number of errors at run-time, which tend to be more disruptive.

F# also helps protect against runtime errors through the use of Monadic types. Sometimes, certain actions in a program may need to return *nothing*, such as an unsuccessful lookup in a Map/Dictionary or if there is an absence of some data. Usually such messages are communicated within the

One such example is found when pattern matching, as seen in Figure 2.3. The function `printColor` takes as input some colour of the rainbow (type `RainbowColor`) and prints the colour. `RainbowColor` is an F# Discriminated Union (DU) type [21], where the data stored in the value is not fixed; it can be one of several distinct options. DU types have many applications, ranging from representing valid and error cases to small object hierarchies. The type system allows for the IDE to first infer the type of the variable `color`, and then realise that the pattern match does not cover the DU case for when `color` is `Green`. Mousing over the warning line in Visual Studio gives the following message: "Incomplete pattern matches on this expression. For example, the value 'Green' may indicate a case not covered by the pattern(s)."

program using either exceptions or NULLs which have to be caught and handled. If not tracked and handled appropriately, NULLs in particular can lead to some very nasty and hard-to-debug runtime errors. F# allows programmers to minimise the use of exceptions and avoid the use of NULLs using the `Option` and `Result` types. The `Option` type can take the value `Some <type a>` or `None`, giving the programmer a safe way to indicate nothing without using NULLs. The `Result` type can reflect success (`Ok <type a>`) or failure (`Error <type b>`), giving the programmer a safe way to propagate errors through their code without raising exceptions or returning NULL.

While F# has the many features and benefits of pure-functional languages and strongly encourages the programmer to use them, it is an impure functional language and does allow the programmer to use other styles of programming. For example, mutable state in functions is allowed by the use of the `mutable` keyword upon definition. This allows flexibility; programmers can use the default functional style in most places, and can selectively choose where to introduce functionally impure constructs such as mutable state. An example of this in Issie is the *FastSimulation* used by the Step Simulator, which uses mutable arrays to represent the value of component inputs/outputs at different clock cycles.

As most of the features mentioned in this section are either unique to or far better implemented in F# compared to other common app development languages such as JavaScript or Python, and that F# belongs to the most bug-free class of languages, it can be said that Issie's choice of programming language is apt and ideal.

2.4.2 Ecosystem

While originally built as a language for the .NET framework, built to run on the Microsoft Common Language Runtime (CLR) [22], F# can also execute in Javascript environments through the use of third-party trans-compilation tools [23]. Therefore, F# can be used to build desktop applications using .NET, as well as Javascript web apps. Issie is built using the latter method; F# code is compiled to JavaScript, which is executed in a desktop environment through Electron. This section describes the chosen tools in this process, and the reasoning behind those choices.

Electron

Electron [24] is a framework for building desktop applications using JavaScript, HTML, and CSS. Electron comes bundled with the open-source Chromium browser and Node.js, a runtime environment that lets JavaScript code execute outside of a browser. Using these, Electron enables web apps to run locally on the user's machine outside of a web browser, akin to a native program. The advantage of this approach is that a developer can create a cross-platform application without any experience of programming native applications for any platform. On the other hand, pure native desktop applications often require differing codebases if multiple platforms are to be targeted [25]; this is a far more labour-intensive approach which also requires a wider skill set. Common native desktop application technologies, such as UWP and WPF, have a much steeper learning curve compared to Electron as well [26]. Given that Issie is maintained by various students over time, an ecosystem that allows for easy cross-platform development with a relatively shallow learning curve is preferable, making Electron well suited for Issie's ecosystem.

However, potential issues with Electron must also be discussed. As Electron apps bundle Chromium and Node, they are often quite large. Additionally, due to the use of the RAM-intensive Chromium browser, Electron apps tend to use more memory than other similar applications [27]. This could lead to decreased performance on low-end hardware. Electron is also dependent on a large amount of open-source code, and web apps are often very reliant on Node dependencies, which can change at any time. This reliance could also be perceived as a weakness.

However, despite some of its shortcomings, Electron is still a good fit for Issie's use case. Issie in its current form is performant, responsive, and stable; implying that the performance issues with Electron have not affected Issie significantly enough for there to be notable issues. The advantage of easy cross-platform development, as well as good integration with Fable (which allows F# to be used as the programming language), outweighs the risks posed by potential Node dependency issues, as well as a larger app footprint.

Fable

While Electron provides a convenient framework for building cross-platform applications, it requires a JavaScript codebase. JavaScript is a weakly and dynamically typed imperative language - making it a sub-optimal choice for the development of Issie. The gap between development and deployment is bridged by **Fable** [23], a compiler that brings F# to the JavaScript ecosystem. Fable compiles F# to clean JavaScript code which can run under Electron. Fable also includes bindings for React [28], a highly performant JavaScript library for building user interfaces; this allows Issie's F# code to create React elements which Fable will compile to their respective JavaScript implementations.

2.4.3 User Interface and Rendering

A well-written and structured Graphical User Interface (GUI) is an essential component of any application, but it becomes even more vital in the context of an application like Issie which prioritises an interactive and intuitive interface. In Issie, most of the user's interaction with the program is done using the mouse; clicking, dragging, and hovering. This means that the UI code must be able to handle a constant stream of pseudo-random mouse events and deal with them quickly and appropriately. It must also be maintainable and extensible to easily allow for more functionality to be added over time. This section describes the workings of Issie's UI framework, as well as the reasoning behind the decision to choose the framework.

Elm, Elmish, and MVU

The Elm Architecture [29], also known as the MVU Architecture, is a pattern for creating interactive programs which emerged from the Elm programming language. Elm [30] is a purely functional language for creating websites and web apps. Elm compiles to JavaScript - this is similar to how using F# with Fable allows functional F# code to be compiled to JavaScript for a web app. Issie uses the Elmish library [31], which brings Elm's MVU architecture to F# and integrates well with Fable, enabling Issie's smooth and robust GUI.

The MVU architecture gains its name from the three parts it splits the UI code into:

Model: A data structure which stores the state of the application. In Issie this is a very large record, containing lots of information ranging from the results of a simulation to which components are selected and highlighted.

View: A pure function which specifies how the information in the model (application state) should be displayed to the user. The developer writes a function to turn the model into HTML/CSS equivalents, and Elm (or Fable if using F# with Elmish) converts this to actual HTML/CSS and renders it.

Update: A way to update the application state triggered by events such as user input. These events are communicated within the program using *messages*, which are triggered by the user interacting with the GUI. The update function is a pure function which accepts two parameters, the incoming message and the current model. It then interprets the message and returns an updated version of the model.

An initial Model is specified in the code, which is rendered by the View function on application startup. The user can then interact with the application - each interaction will generate a message which is passed to the update function. The update function processes this message and, if required, will return an updated model which reflects the effects of the interaction. On its next invocation, the View function will render this updated application state, showing the user the result of their interaction.

The MVU Architecture has numerous strengths, many of which make it suitable for use in Issie. Its structure, based on an immutable Model with deterministic (pure) View and Update functions, means that data flows in only one direction through the whole application, while events which warrant a change in the application state are clearly marked as messages. This makes the rendering process easier to understand for the programmer [32], meaning that more time is spent writing useful code instead of debugging. This is unsurprising given that Elm, like F#, is a functional language, and therefore belongs to the "Functional-Static-Strong-Managed" class of languages mentioned in Section 2.4.1. F# also has features which integrate well into the MVU

architecture, strengthening it further. One such example is F# 's strict compile order, which helps avoid cyclic references between view subfunctions [32]. Altogether, the MVU architecture is a good fit for Issie, improving the developer experience while delivering a performant UI.

On initial inspection, the design decision of having a View function which repeatedly re-renders the Model every time there is some update appears inefficient. This would be the case if Elm (and other implementations of MVU) used the traditional process for rendering a web page/app. This traditional process for rendering HTML is shown in Figure 2.4. The HTML is parsed and a DOM (Document Object Model) is constructed; this is a tree representation of the HTML. Whenever the website/web app state changes, certain parts of the generated HTML change too. The DOM tree must be updated to reflect these changes, however this is often a slow process for large web apps with a large number of nodes in the DOM tree, resulting in the UI appearing sluggish. Elm combats this by using an intermediate **Virtual DOM**, a lightweight and optimised version of the main DOM tree on which operations are cheap. As shown in Figure 2.5, Elm exposes this virtual DOM to the programmer, who specifies how the state should be viewed in the View function. For each change of state, a new virtual DOM is constructed and compared to the previous virtual DOM. Through this, Elm can determine the exact change to the UI. Both the construction and comparison are fast due to the lightweight nature of the virtual DOM. Elm repeats this process for multiple changes to the UI, creating a batch of updates to be performed on the actual DOM tree at once. This method significantly reduces the per-update penalty for updating the DOM, allowing for efficient re-rendering of the program state. This improves the performance of the web app. Another advantage of this method is that the task of efficiently updating the DOM is abstracted away from the programmer, leaving them with only the task of specifying how their website or web app should look.

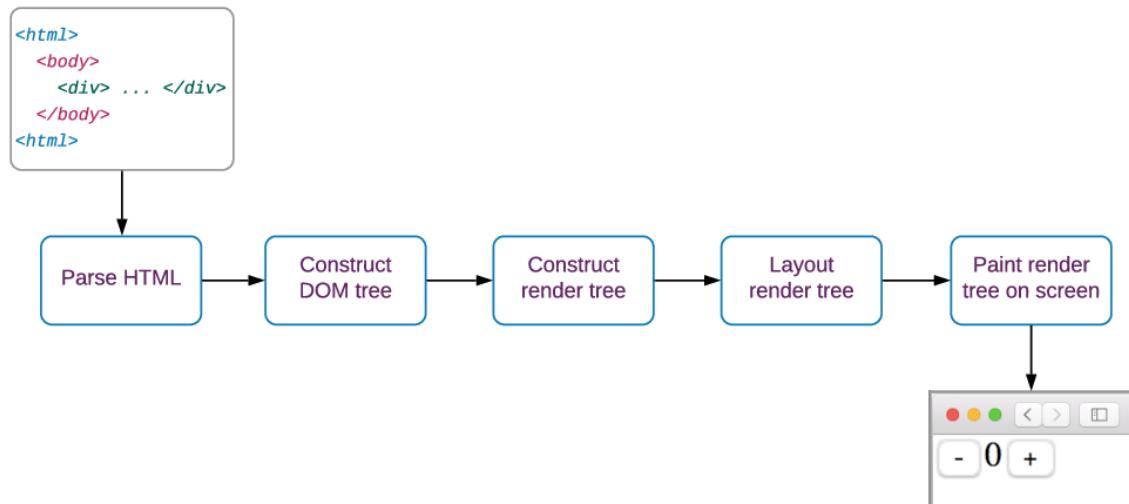


Figure 2.4: How browsers traditionally render HTML using a DOM [33]

React

React, as mentioned earlier, is a library for building user interfaces. In the Issie project, Elmish uses React for rendering the UI due to its efficient virtual DOM. Thus, the virtual DOM mentioned previously is actually the React Virtual DOM, which is a tree of React Elements. React Elements are simple objects, and are therefore cheap to create [34], meaning that operations on the React DOM are computationally inexpensive. The View function returns a React Element which shows the current state of the UI. React improves the performance of an application by only re-rendering elements when necessary. The React equivalent of the process mentioned previously where virtual DOMs are compared to find UI changes is called *Reconciliation* [34].

React also provides a further method for increasing performance: memoization [35]. Memoization is most useful when working with components whose state is partially dependent on computationally intensive calculation. Under traditional React, this value would be re-calculated on every render, even if it had not changed. This increases the amount of time each render takes. An example of this would be the result of the addition of two very large numbers ($a + b$), which is a very CPU

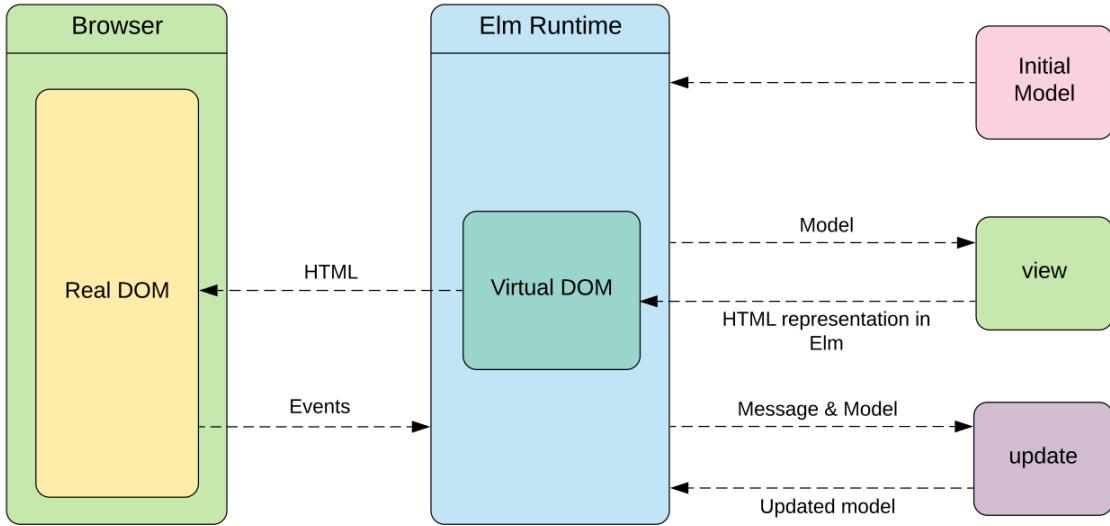


Figure 2.5: The Elm Architecture [33]

intensive task. Memoization combats this by caching the previous value of the computation and only performing the calculation whenever a or b changes. Issie already uses some memoization in its code for the Step Simulator: after a sheet is simulated for the first time the simulation is cached. If the sheet hasn't changed the next time a simulation is started, the cached simulation is used instead of building a new one, increasing performance.

Fulma

Fulma [36] is an F# library which provides a wrapper around Bulma [37], an open-source CSS framework which provides ready-to-use front-end components for building responsive web interfaces. Fulma brings these components to F# for use with Fable React. React components such as buttons, forms, and tables can easily be specified in the F# code using the functions provided by Fulma. Listing 2.1 shows the F# code for generating a table using Fulma, while Figure 2.6 shows the rendered table.

```
Table.table [ Table.IsBordered
             Table.IsNarrow
             Table.IsStriped ]
[ thead []
  [ tr []
    [ th [] [ str "Firstname" ]
      th [] [ str "Surname" ]
      th [] [ str "Birthday" ] ] ]
  tbody []
  [ tr []
    [ td [] [ str "Maxime" ]
      td [] [ str "Mangel" ]
      td [] [ str "28/02/1992" ] ]
    tr [ ClassName "is-selected" ]
      [ td [] [ str "Jane" ]
        td [] [ str "Doe" ]
        td [] [ str "21/07/1987" ] ]
    tr []
      [ td [] [ str "John" ]
        td [] [ str "Doe" ]
        td [] [ str "11/07/1978" ] ] ] ]
```

Listing 2.1: Simple F# code for generating a table with Fulma [38]

Firstname	Surname	Birthday
Maxime	Mangel	28/02/1992
Jane	Doe	21/07/1987
John	Doe	11/07/1978

Figure 2.6: Fulma Table Example

2.4.4 Overview

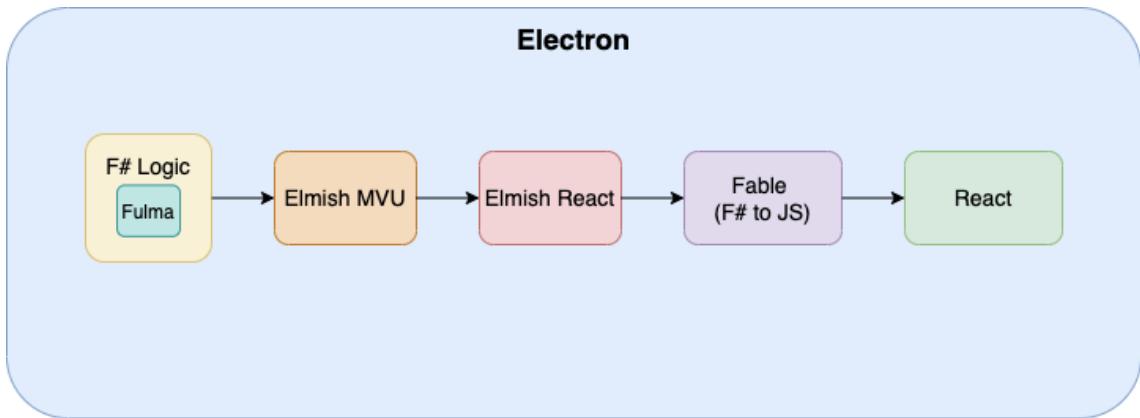


Figure 2.7: An Overview of Issie’s Technology Stack

We can bring the above sections together to get an overview of Issie’s Technology Stack, which is presented in Figure 2.7. The core logic of the Issie application is written in F#, with a structure which allows for adoption of the MVU architecture. Within the View function, using libraries like `Fable.React` and `Fulma`, HTML and React components can be specified. The `Elmish` library provides the MVU framework and methods for bringing together the Model data structure, View function, and Update function. A virtual DOM is necessary for an MVU application because it allows the application to frequently re-render its state after each update in an efficient manner. Due to its reliable and performant implementation of a virtual DOM, the React framework is used for Issie. The library `Elmish.React` handles the transition from the generic Elmish view to React, after which `Fable` compiles all of this F# code to JavaScript for React to render. All of this runs within Electron, allowing Issie to run as a desktop application despite having the technology stack of a web app.

2.5 Issie’s UI

2.5.1 Overview and Evaluation

In line with the core principles that state all features implemented in Issie should be **Obvious** and **Intuitive**, Issie’s UI aims to be consistent and straightforward. Consistency in the UI helps constantly prove a user’s assumptions about the user interface right, creating a sense of control, familiarity, and reliability [39]. It also plays a crucial role in exposing all of the features of an application to the user – this is because features which are accessed in an inconsistent manner may be missed or incorrectly understood by end users. The intuitive guidance provided by a consistent and straightforward UI means that students using Issie spend less time learning how to use it compared to other similar applications; at its launch most students were able to get started within five minutes [2]. With the limited amount of time in labs available, less time spent learning to use tools means more time spent on learning educational content.

Since its original release, more features have been added to Issie, such as the Waveform Simulator, with the UI being updated to accommodate these features. At the time of writing, the latest release version of Issie is version 3.0.0, released on 16th April 2022. Images of Issie's UI can be found in Appendix A. On launch, Issie gives the user the option to create a new project or open an existing one. A project consists of one **main** sheet, and any number of sub-sheets, which define reusable custom components. Figure A.2 is an annotated screenshot of a sheet open in Issie. The UI can be split into the following sections, which are highlighted in different colours in Figure A.2.

Traditional Menu Bar (Highlighted Red): The traditional menu bar, consists of the "Edit" and "View" menus. It is not used to expose Issie's core features - instead, it is used to let the user do basic operations, such as zooming, copying and pasting etc. Many of these functions are duplicated through keyboard shortcuts or through the buttons in the bottom left of the sheet (undo, redo, copy, paste). The fact that Issie has two separate menus is a little strange, and it may be worth moving all functionality from the traditional menu bar to the top white menu bar. However, this is not an urgent cosmetic issue.

The Canvas: The canvas is the central focus of the application - it is where the schematic is designed. Components are dragged onto the canvas, with red-dashed lines appearing to show when a component is aligned with a grid line. Components and connections on the canvas can be interacted with in many ways, such as hovering, clicking to select, and dragging to move.

Right Tabs Panel (Highlighted Green): The panel on the right-hand side of the application provides the user with ways of (i) affecting the state of the canvas and (ii) gaining insight into the schematic itself. There are three tabs: the **Catalogue**, which lets users add new components to the canvas (users can hover on components for details); **Properties**, which lets users view and modify component properties like name and width, and **Simulation**, which opens the Step Simulator, giving users the option to start a simulation. Due to the Step Simulator requiring more display real-estate, the right panel widens when the Simulator tab is selected. When the Waveform Simulator is running, an additional fourth tab appears in this section – this makes the tab layout appear cramped.

Top Bar (Highlighted Blue): In the first release of Issie, the white menu bar located at the top of the application contained controls to, (i) save the current sheet, (ii) Switch to another sheet in the open project, and (iii) Open another or create a new project. The filepath of the current project and sheet are also displayed. All of these elements were consistent - with the focus on interacting with files. A slight grievance with the "Project" and "Sheets" dropdown menus is that they are not closed by clicking elsewhere in the application, only by clicking on the menu again. Since then, an "Info" button has been added featuring information about Issie and a short user guide. While this is not necessarily consistent with the other elements in its vicinity, the top bar enables the "Info" button to have maximum visibility, something which a student requiring information may appreciate. One very inconsistent part of the UI is how the Waveform simulator is accessed. While most actions which analyse the schematic (e.g. Step Simulator) are accessed via the Right Tabs panel, the Waveform simulator is opened through a button on the top bar. This causes the previously unseen "WaveSim" tab to spawn in the right panel, and the panel boundary becomes draggable. The user can select the waveforms to view (Figure A.3) and then view the waveforms (A.4. The Waveform simulator's behaviour is inconsistent with that of the Step simulator:

- Step Simulator displayed in a wider, fixed-width panel. Waveform Simulator displayed with panel starting at regular width, but the panel can be manually resized by dragging.
- The user can start a simulation in the Simulation tab, and still interact with other tabs. This is not the case for the Waveform simulator, which locks the user in the WaveSim tab.
- Step Simulation started and ended with the same button, with a clear indication of how to end the simulation. There is no end/close button on the WaveSim tab when waveforms are being viewed. Instead, the user must press the "Edit List" button, which takes them back to the selection menu where there is a close button. This is very unintuitive.

Buttons on bottom-left (Highlighted Yellow): Undo, Redo, Copy, and Paste buttons. These are four common functions, and therefore it makes sense to have obvious and fast ways to access them. Useful to get started if the user is not familiar with keyboard shortcuts.

2.5.2 Considerations when adding new features

As discussed in the previous section, Issie's UI at time of launch was very intuitive and straightforward. However, when there are fewer features it is easier to condense the interfaces for them into a well structured and streamlined UI. The inconsistencies of the Waveform Simulator tell a cautionary tale of how new features can introduce impurities into an application's GUI. This project aims to add a whole new way of visualising logic (truth tables) to Issie, and given the saturated nature of the current UI, some redesigning may be necessary to accommodate all features.

2.6 Combinational Logic Simulation in Issie

Currently in Issie, the users can gain an understanding of combinational logic by simulating it using the Step Simulator. Users provide values for each input to the logic, and can read the output almost instantly. This indicates high performance. Given that visualisation of combinational logic through truth tables is likely to require some form of combinational simulation, there is merit in exploring and understanding the implementation of the performant Step Simulator.

Figure 2.8 provides an overview of Issie's process for building and running a Step Simulation. In this process, various checks must be performed; firstly the logic designed by the user must be verified to be syntactically correct, secondly the organisation of project files must be correct, and thirdly some Issie specific limitations (e.g. no cycles in combinational logic) must be enforced. Issie's simulation building process can be said to have three levels, with each level having an associated data structure which represents the schematic. These data structures are: the **Canvas State**, **Simulation Data/Error**, and a **Fast Simulation**. A set of checks are performed at each level, and only upon passing these checks can a schematic be transformed into the subsequent data structure. If any of these checks fail due to an issue with the user's schematic, the simulation building process returns a *SimulationError*, which tells the user what the error is and which components/connections are affected. A key takeaway from Figure 2.8 is that the process of building a simulation is separate from the process of running it. During the **FastSimulation** building process, the schematic is analysed, with components being placed into an appropriate order for combinational reduction. However, as the **FastSimulation** data structure is mutable, the values of each input can be updated without having to rebuild the whole simulation. Therefore, the time taken simulating a different input combination is quite short, as only the reduction function has to be re-run to find the new outputs. This distinction between building a simulation and running it with different input values makes the existing Step Simulator an optimal choice for use in truth table generation, which will need to simulate multiple input combinations as fast as possible.

2.7 Effects of Application Performance on Users

Interactivity is a key part of Issie, and one of the major contributors to perceived interactivity is the perceived responsiveness of an application [40], which itself is often tied to application performance. Perceived responsiveness is relevant in systems where a user's "perceived control over the interaction process reflects their ability or confidence in performing related activities". Combining these findings with the cognitive theory of self-efficacy described in Section 2.1.3, which proposes that academic performance is linked to perceived ability, indicates that perceived responsiveness is an important factor to consider when building an educational application. Robert Miller [41] describes three classes of perceived responsiveness based on a computer's response time:

- $\leq 100ms$: Perceived as instantaneous.
- $\leq 1s$: Noticeable but does not lose the user's attention.
- $\geq 10s$: Loses the user's attention.

Ideally, every interaction would be in the first class, however naturally certain tasks will take longer than others. In his paper, Miller investigates specific "Topics": examples of human-computer

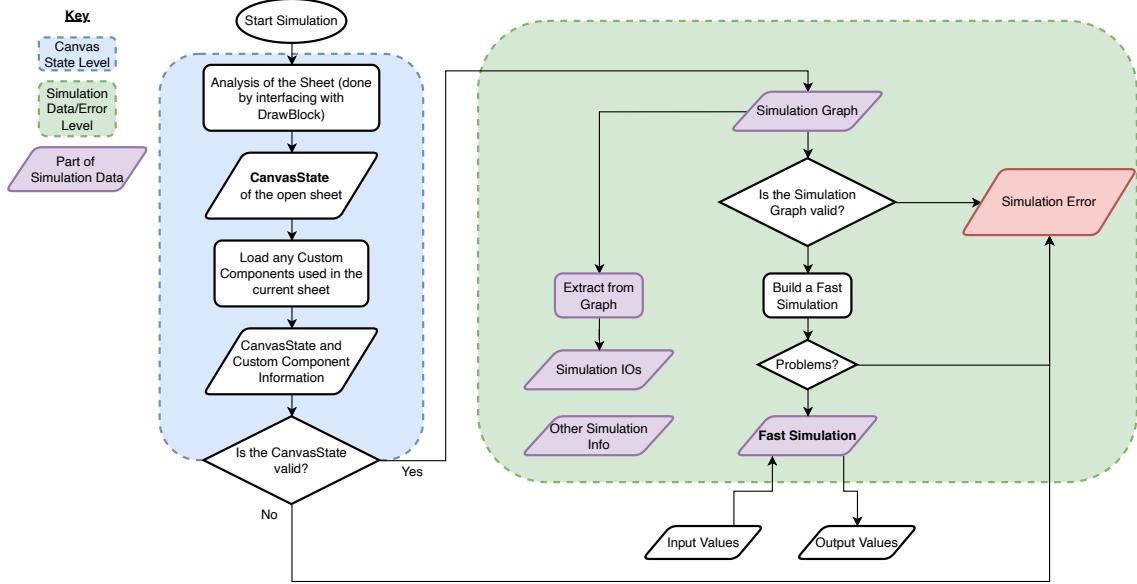


Figure 2.8: An overview of Issie's Step Simulation

interaction, and examines the acceptable response time. A "*Response to complex inquiry in tabular form*" (Topic 9) should return a complete response within 4 seconds, but a maximum delay of 2 seconds is preferred. This is particularly relevant to this project as it resembles the operation of calculating and presenting a Truth Table to the user. Therefore, it could be suggested that in order for the added Truth Table generation feature to have the maximum positive effect, generation should take less than 4 seconds in all cases, and ideally less than 2 seconds for simpler cases. By the same rationale, this time limit should also apply to complex operations on the truth table, such as reduction with Don't Cares or Algebra. As mentioned in subsection 2.1.2, engineers tend to be kinesthetic learners and therefore are likely to prefer tools which let them interactively experiment with what they are learning. Therefore, it can be proposed that increasing the responsiveness of Issie (which increases perceived interactivity), will lead to a better learning experience.

2.8 Agile Software Development

Agile software development [42] is an iterative approach to software development, prioritising an "agile" response to changing requirements and user feedback. In Scrum, a popular Agile framework, the development team's plan is split into short *sprints*: 1-4 week periods of work which deliver some "product", which is fully usable in its own right [43]. The high-level descriptions of the work are organised in a structured list called the *backlog*. The hierarchy in the backlog is as follows [44]:

- **User Story:** Is a short, basic task which should not take too long to complete. An example would be a task to move the waveform simulator to the right tab, next to the step simulator.
- **Epic:** A large body of work which consists of multiple user stories. An example of an epic would be to implement truth table generation for a user selection of components.
- **Initiative:** Represent overarching goals of the software development project, consisting of multiple epics.

Over successive sprints, the number of stories in the backlog should reduce. There are regular meetings with the product owners, during which the development team¹ receives feedback which can then be acted upon. This may often add more work to the backlog. This cycle of continuous development aims to deliver working products which match user requirements on a frequent basis. A survey [45] of software projects managed in an Agile fashion found that "The practice of agile

¹Usually there is an intermediary (Scrum Master) managing the organisation of the development team. However, given that Final Year Projects that improve Issie rarely interact with one another, this role is mostly redundant.

software engineering techniques "is a critical success factor that contributes to the successful agile software development projects in terms of Quality and Scope". This is also reflected by industry, where Agile approaches to software development have succeeded where more traditional linear Waterfall-esque approaches have failed [43].

In the context of this project, the "product owner" is Dr Thomas Clarke, who oversees the overall development of Issie, while the "development team" consists of the author and other students.

Chapter 3

Project Requirements

This chapter specifies the comprehensive set of requirements for the updated version of Issie delivered by this project, as well as any accompanying documentation. These requirements build up to a more formal specification that is representative of the project aims, and takes into account the prior evaluation of the current version of Issie. Upon completion of the project, the deliverables will be evaluated against these requirements to ascertain the extent to which the project was successful.

3.1 Requirements for Combinational Logic Visualisation

The primary purpose of this project is to develop new ways for visualising combinational logic in Issie. These requirements specify the features that these novel visualisation methods should have, as well as constraints on their operation (e.g. required performance). Requirements can either be essential (**Ex**) or desirable (**Dx**). Essential requirements must be fulfilled for the project to be considered successful.

Essential Requirements

E1.1 Analyse a schematic containing **only combinational logic** and display a standard numeric truth table for that sheet.

E1.2 Analyse part of a schematic selected by the user containing **only combinational logic** and display a standard numeric truth table for that selection.

E1.2.1 New inputs and/or outputs should be created temporarily (if necessary) to feed inputs and/or read outputs from the selected logic.

E1.2.2 It must be clear which newly generated inputs/outputs shown in the truth table correspond to inputs/outputs into the selected logic.

E1.3 Have a truth table generating algorithm which can handle:

E1.3.1 Multi-bit inputs and outputs. Any temporary inputs/outputs created while generating a truth table for a selected logic block must have correct widths.

E1.3.2 Custom Components (sub-sheets), including when they are part of selections.

E1.3.3 Displaying inputs, outputs, and viewers.

E1.4 Give users an option, when possible, to reduce the truth table based on patterns in the logic (e.g. Don't Cares).

E1.5 Give users the option to filter the truth table by fixing input or output values.

- E1.6** Truth Tables must be displayed in a clear and easy to understand format, and features involving truth tables (e.g. filtering, reducing etc.) must be presented in an intuitive way.
- E1.7** Truth Table generation and reduction must take no longer than 4 seconds, with an ideal target of under 2 seconds.
- E1.8** Graphical manipulation operations on the Truth Table, such as re-ordering rows, sorting etc. should appear instantaneous (i.e. take less than 100ms).
- E1.9** Use the above features to give users a clearer insight into the digital logic on the schematic, or to further reinforce their existing understanding.

Desirable Features

- D1.1** Generate and display algebraic truth tables.
 - D1.1.1** At minimum should at least support multiplexer and adder circuits.
 - D1.1.2** Preferably should have a rich set of algebraic operators with support for most circuits.
- D1.2** Provide an interactive truth table interface.
 - D1.2.1** Mousing over parts of the truth table could have effects on the schematic (e.g. annotations or highlighting).
 - D1.2.2** Users can rearrange order of columns/rows in the truth table.
 - D1.2.3** Users can sort the truth table in ascending and descending order.
- D1.3** Let the user access truth table related functionality without going through numerous steps while also keeping the number of buttons on the screen to a minimum to avoid cluttering the interface.

3.2 Software/Documentation Quality Requirements

In the process of adding new combinational logic visualisation methods to Issie, this project will also make significant additions/changes to the Issie codebase. This section highlights the requirements for the quality of the software contributed, along with the documentation for said software.

Essential requirements

- E2.1** Deliver performant, working, bug-free code which adheres to Issie's code guidelines and other principles such as "MVU-ness".
- E2.2** Write comments in the delivered code which adequately explain it such that it may be worked on in the future by other developers.
- E2.3** Deliver code that is easy to maintain for future developers.
- E2.4** Provide any other necessary documentation.

Desired features

- D2.1** Deliver a tweaked, or possibly partially redesigned UI which exposes all Issie features in a straightforward and intuitive way to the user, with a focus on extensibility (i.e. can the UI accommodate for future extensions to Issie?)
- D2.2** Update the Issie website with information about any newly added features.

Chapter 4

Analysis and Design

This chapter describes all the changes and additions made to Issie by this project, as well as organisational decisions. Additionally, the rationale behind these changes, including an analysis of high-level methods, will be explained.

4.1 Approach towards Software Development

Features were added to Issie using an incremental and Agile approach [46]. The incremental approach seeks to write code through a repeated cycle of three steps: (1) Analysis and Design, (2) Writing Code, and (3) Testing. A basic task/requirement is broken into several parts - with each of these parts being written as an individual function¹. Each function is tested both as a unit and when integrated into the codebase. All of these different parts build upon one another and come together to deliver the desired functionality. One caveat of the incremental approach is that the intermediate versions of the app are incomplete and therefore not suitable for any kind of release or proper demonstration. This can make it difficult to get proper feedback on the state of the application as a whole. This was acceptable within short time-frames, but not for long-term software development over the course of the project. For that case, the Agile approach was considered. Small features, represented as a short sequence of stories, were built using the aforementioned incremental approach during sprints. Upon completion of each feature, it could be considered that a new "product" (slightly improved version of Issie) was delivered. During each project meeting, feedback was obtained on the work completed, and any necessary adjustments will spawn new user stories. Agile is generally used in continuous software development projects; however, due to the constraints of a Final Year Project: deadlines, need for planning and report writing, a pure Agile approach was not deemed suitable. Instead, a hybrid approach was pursued, one which embodies Agile principles while still working within a plan-based framework. During the planning phase, the backlog of user stories was intelligently structured such that epics were ordered by importance to the project, and stories within the epics were structured such that features would be added incrementally. This ordered backlog can be found within the work breakdown structure (WBS) described in Figure B.1 in Appendix B.

4.2 Technology Stack

This project does not make any deviations from Issie's existing technology stack, described in Section 2.4. The F# application code is compiled to JavaScript using the Fable compiler, and is run as a web app in a desktop environment using Electron. The user interface is powered by React and the MVU architecture, brought to the F# ecosystem by the `Fable.React` and `Elmish` libraries. New files added to the Issie codebase by this project are split between two existing directories in the *Renderer* project. Files which implement features related to actual truth table generation and reduction are placed in the *Simulator* directory, while files which implement the UI are placed in the *UI* directory.

¹"individual function" does not refer to a single F# function, but to a top-level function which uses groups of helper and sub-functions

4.3 Top Level UI Changes

The top-level UI of the application has been changed to accommodate truth tables. As a result of this, the way the waveform simulator is accessed has been changed as well. In prior versions of Issie, the right tab section had three options: **Catalogue** for adding new components, **Properties** for changing component properties, and **Simulation** for launching the Step Simulator. When applicable, Waveform Simulator could be accessed by clicking a button in the top bar, which would temporarily create an extra tab in the right section. Given that most interactions related to modifying or gaining insight into the schematic were done through the right section tabs, the decision was made for the truth table to be displayed there. The large amount of space needed to display a truth tables and related functionality warranted a separate tab for truth tables. However, this approach posed some potential problems. During the earlier evaluation of Issie, it was found that the method of accessing the waveform simulator was inconsistent, and that the waveform simulator should ideally have its own permanent tab too. Therefore, the total number of tabs would increase from three to five, resulting in a cramped layout. The updated version of Issie delivered by this project solved this problem by grouping all circuit simulation activity under one tab. The third tab in the right section has been re-named to **Simulations** and features three further sub-tabs, each for the Step Simulator, Truth Table generator, and Waveform Simulator respectively. This can be seen in Figure 4.1, where the **Truth Table** sub-tab is open. From this tab, the user can choose to generate a truth table for the whole sheet. Additionally, if a correct configuration of components is selected, a second option to generate a truth table for the selected configuration is also shown to the user. As with the Waveform Simulator, when the Truth Table tab is open, the dividerbar can be dragged to resize the right section. Previously, there was a bug where, if the right section was scrollable, the dividerbar height did not dynamically resize. This bug was fixed by this project.

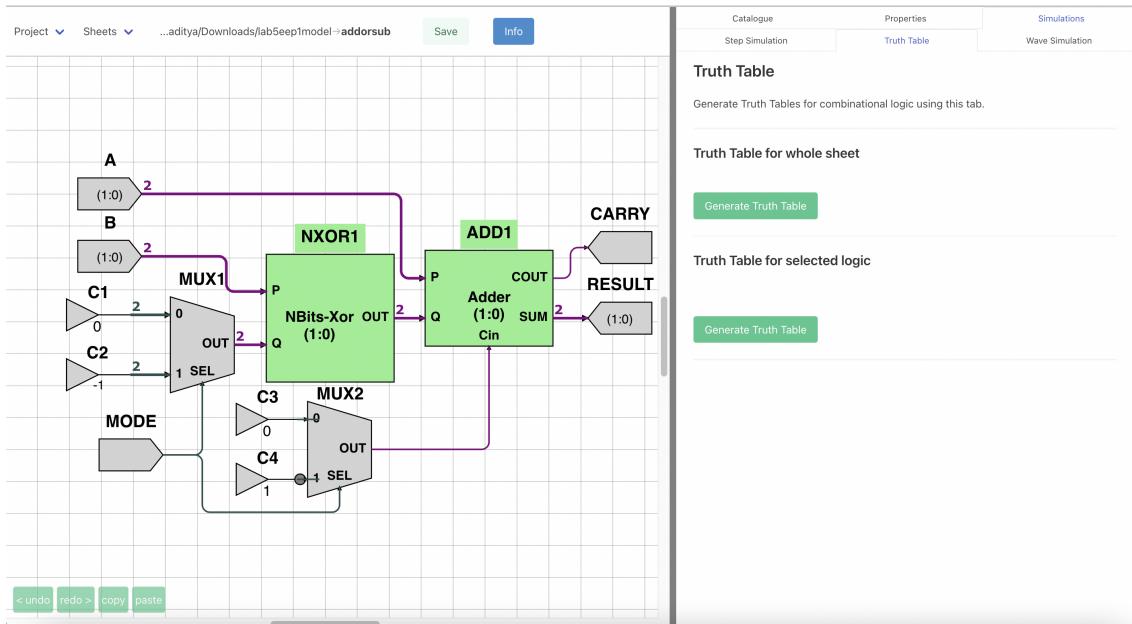


Figure 4.1: View of Issie's updated Top-Level UI

4.4 Generating Numeric Truth Tables

The user can generate a truth table by clicking on the *Generate Truth Table* button. Like the *Start Simulation* button in the Step Simulator, this button provides feedback on the correctness of the canvas to the user through colours. A green button indicates a correct canvas which can be used to generate a truth table. If there are issues with the schematic, the button will instead be yellow, and clicking it will inform the user of the nature of the error via an error message and highlighting the erroneous component. Finally, the button can also be a faded green; this indicates that while

the schematic is correct, it is unsuitable for truth table generation as it contains sequential logic. This is done to maintain consistency with the Waveform Simulator button, which becomes faded when there are no sequential components.

When the user generates a truth table, a numeric truth table is generated at first. Due to performance reasons, which will be discussed later, Issie will only generate up to the first 1024 rows in a truth table. In cases where the truth table should have been larger than this limit, it is considered *truncated*. The user is warned of this through a yellow coloured popup with the following message: "*The Truth Table has been truncated to 1024 input combinations. Not all rows may be shown. Please use more restrictive input constraints to avoid truncation*". Once a truth table is generated, the user can interact with it in numerous ways through the user interface, transforming and regenerating it as necessary. Figure 4.2 shows two views of the Truth Table tab, which contains a menu with collapsible sections. The compact view (Figure 4.2a) is the default, with all sections other than the one displaying the truth table, reduction operations, and base selector being collapsed. The other sections can be expanded to reveal additional functionality; Figure 4.2b shows the view of the Truth Table tab when all sections are expanded. The tab section is also scrollable.

Figure 4.3 shows a high-level overview of the truth table generation process. This process will be covered in more detail in the Implementation chapter, but the overall process can be summarised into three phases: building the simulation, finding the input combinations which make up the left-hand side of the the truth table (Input Space), and simulating each combination in the input space to build the truth table. Decisions had to be made on how to generate the input space, as well as how to simulate each input combination.

(a) Compact View

(b) Expanded View

Figure 4.2: Contents of Truth Table tab after a truth table has been generated

A decision also had to be made regarding the handling of multi-bit IOs. In Issie, inputs to combinational logic can be multi-bit ($\text{width} > 1$), and propagate through the logic via multi-bit buses. Two possible approaches for handling these multi-bit inputs were considered. The first option was to split a multi-bit input into its constituent bits in the truth table; this aligns more closely with the actual hardware implementation of the logic [47], where component ports only accept a single-bit signal. However, this is not user-friendly at all - for example a 32 bit bus would be split into 32 different inputs. This would massively increase the number of columns in the truth table, impacting the utility of the truth table as an aid. Furthermore, splitting inputs and outputs into separate bits would obscure certain relationships, such as those of arithmetic circuits. Instead, the decision was made to represent each multi-bit input/output as one column in the truth table, with the value shown in hexadecimal format by default. The user can choose between hexadecimal, decimal, and binary representations.

4.4.1 Decision to re-use Step Simulation Code

At its core, generating a complete numeric truth table is a brute-force process. Each possible combination of input values must be simulated to calculate the corresponding output values, and the relationship should be recorded in the table. Given that Issie already features a performant and reliable simulator (step simulator) for calculating the outputs of combinational logic, the decision was made to use as much of its implementation as possible. This approach has many advantages:

- The existing step simulator has been extensively tested by end-users, meaning that its implementation is most likely bug-free. By using it, it will reduce the chance of the new feature introducing new bugs.
- In most cases, reusing existing code is much faster than writing new code from scratch. Not only is time saved on writing new code, but the amount of time spent debugging is also reduced.
- Reusing existing code will help keep the overall size of the codebase small. Not only does this help future programmers who work on the project by reducing how much they have to understand, but it also means that any future improvements made to the simulation code are also improvements to Truth Table generation.

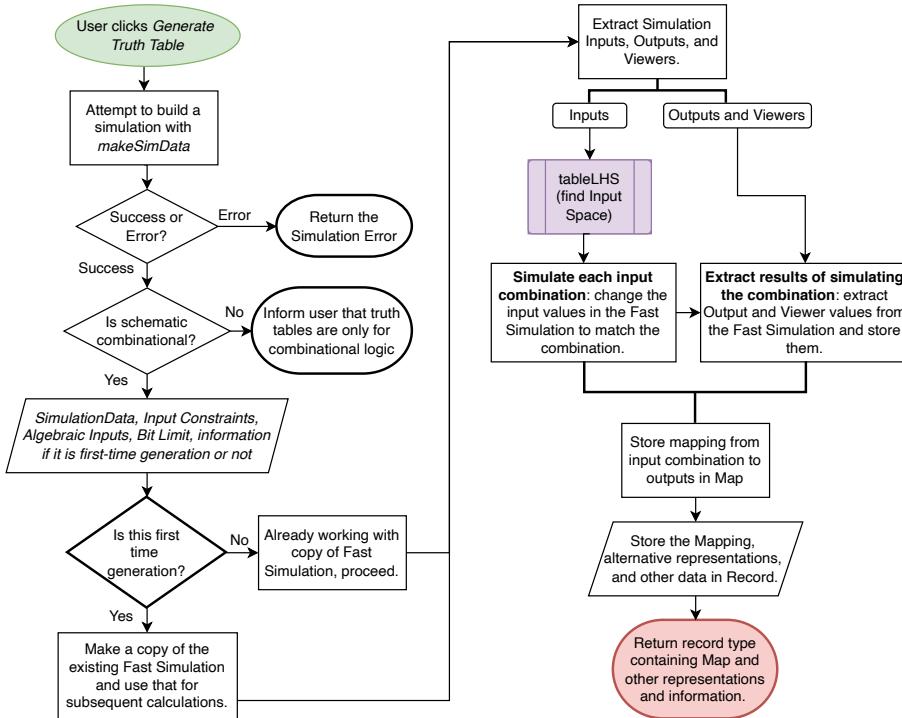


Figure 4.3: High-Level view of Truth Table Generation

4.4.2 Evolution of the Input Space Generation Algorithm

The process of calculating the input space, highlighted in purple in Figure 4.3, had two versions. The change from the first to the second version affected the overall structuring of operations on truth tables in the application. Initially, the entire input space was calculated and simulated. This would result in a very large, exhaustive truth table – one which contained every possible input and output combination. The user would have been able to reduce this truth table by filtering the table or using Don’t Care Reduction. However, several issues arose with this approach, with the most crucial being that of performance.

For a truth table with n inputs ($x_1..x_n$), each of width w , ($w_1..w_n$), the number of rows in the truth table is given by the sum of all widths raised to the power of 2, as shown in Equation 4.1. For example, a schematic with three inputs, with widths: $w_1 = 1, w_2 = 2, w_3 = 3$ would result in a

truth table with 64 rows. This sum-of-widths formula is derived from the idea that if we define a set S_i as the set of all possible input combinations for input x_i , the left-hand side of a truth table is the **Cartesian Product** of all sets from S_1 to S_n . This project defines the left-hand side of the truth table as the **input space** of the truth table, and the right-hand side as the corresponding **output space**.

$$\text{Row Count} = \prod_{i=1}^n 2^{w_i} = \prod_{i=1}^n S_i = 2^{(\sum_{i=1}^n w_i)} \quad (4.1)$$

The complexity of finding the Cartesian product of multiple sets is an exponential function of the number of sets. For example, the complexity of finding the Cartesian product of n sets, each of size s , would be $O(s^n)$. The size of each set is dependent on the width of the input; Issie does not limit input widths, meaning that the size of the sets may also be very large. For example, a single 16-bit input has 65536 unique values, and combining this with a second 16-bit input yields an input space of 2^{32} combinations (over 4 billion). Simulating this large number of combinations and storing the result takes upwards of 20 seconds, violating essential requirement **E1.7**. Additionally, a sufficiently large input space could cause the system to run out of memory and crash, which is unacceptable. Along with the practical issues with the generation of the complete truth table, it can also be argued that a very large complete truth table is not useful to the user. According to the Atkinson-Shiffrin memory model described in Section 2.1.1, external stimuli only stay in the sensory register for 0.5 seconds prior to decaying. A large number of rows is likely to overload the sensory register, meaning that it is likely that the information will not pass to short term memory unless the user slows down and processes each row one-by-one. Even if this were the case, a truth table with a large number of rows would take over 30 seconds for the user to read and process. Given that short term memory decays in 30 seconds, this means that the user will likely have forgotten the first row by the time they finish reading the last one. Therefore, the method of generating and simulating the entire input space to create an exhaustive truth table was deemed unfit.

As a result, the decision was made to **truncate the truth table**; generate and simulate only part of the input space. The current limit is 1024 rows. Considering that users would only be able to focus on a small subset of rows at a time, and would likely reduce the size of the table anyways, this approach would sacrifice very little for a significant gain. However, any subsequent operations, such as filtering and reduction, would take place on the truncated table, which does not contain all of the necessary data. The nature of these issues, as well as the steps taken to mitigate them are:

Issue 1: Input constraints will be applied to the truncated table, so user may not see a full representation of the relationships for the case they enter. For example, consider the input X which has a width of 8 bits (so values range from 0 to 255), but due to truncation, only rows with values of X up to 31 are present. If the user applies the input constraint $X = 32$ to the table, an empty table will be returned as no rows which fulfil this condition exist in the truncated table.

Solution 1: Apply input constraints during truth table generation. This is achieved by using the constraints to determine a tighter input space, and then simulating that. Giving users a way to choose which inputs contribute more to the input space (they can fix certain inputs and let others vary within bounds) allows them to interactively generate truth tables that deliver the most information to them.

Issue 2: Output constraints will be applied to the truncated table, which is missing many rows. Due to this, the result of applying the output constraints will include all rows which match the condition.

Solution 2: Unfortunately there is not much that can be done to combat this issue alone other than warn the user that they are looking at incomplete results. However, if the user is able to use input constraints to sufficiently reduce the input space first, output constraints could help filter the table further.

Issue 3: Don't Care Reduction will occur on the truncated table, which does not fully represent the logical function performed by the circuit. This could result in incorrect relationships being inferred by the reduction algorithm.

Solution 3 Much like with output constraints, there is no perfect solution as relationships for reduction are inferred from the truth table. Don't Care reduction is therefore limited to smaller schematics which do not produce truncated truth tables. The user is instead guided towards reducing truth tables for larger schematics with **Algebraic Reduction**.

Implementing Solution 1 required a new method of generating the input space. The new method needed to know which sub-sets of the input space it could and could not generate before it actually generated it using the Cartesian product method. Further details of the workings and implementation of this method can be found in the Implementation chapter.

4.5 Generating Truth Tables for a partial selections

The motivation behind Requirement **E1.2** is that a large schematic with lots of components will often contain smaller blocks of logic within it. These blocks may be defined Custom Components, or simply be a collection of gates in one corner of the canvas. Either way, there is value in the user being able to isolate these blocks and learn about the combinational logic implemented by them. Such functionality would also allow users to take a divide-and-conquer approach to debugging logical errors - individual blocks could be inspected to ascertain if they had been implemented correctly.

A challenge with generating a truth table from part of a canvas is that Issie has no existing method for simulating part of a canvas. When working with a whole sheet, the inputs and outputs are well-defined; sheets where any ports aren't connected to inputs/outputs throw *Simulation Errors*. In contrast, a partial selection from a sheet will rarely contain all inputs and outputs. Two methods were considered for simulating the selected logic to generate a truth table, with the latter being chosen.

1. **Extracting the Fast Simulation and feeding values into specific wires.** This method would have involved creating a Fast Simulation for the whole sheet as usual, but then manually changing values in component arrays and seeing how those changes propagated through to the output connections of the selected logic. While this method seemed fit initially, several issues were found after some analysis. The Fast Simulation would be built for the whole sheet, meaning that an error elsewhere on the canvas would stop the selected logic from being simulated. Custom Components would also be harder to manage as the Fast Simulation datatype flattens the design, meaning that all nested logic in Custom Components would be expanded out. The new logic would also be quite different from the truth table generation logic for whole sheets - this is not ideal for future code maintenance purposes.
2. **Intelligently building and correcting a new Canvas.** Following the highlighting of the issues with the first approach, an alternative approach was put forward. Rather than attempting to work with the complicated Fast Simulation data structure, it instead aims to use as much of the existing code as possible by treating the selected logic as a separate instance of a Canvas State and trying to simulate it using the same method as simulating a whole canvas. The main difference between simulating a whole sheet and simulating selected logic is the lack of guaranteed input and output components. This is overcome by finding which ports/connections are inputs/outputs for the selected logic, then intelligently adding 'phantom' input/output components to the canvas in a process called Canvas Correction. Once a corrected canvas corresponding to the selected logic is created, the logic used for generating and viewing a Truth Table for a whole sheet can be reused.

The method implemented by this project first sanity-checks the part of the sheet the user has selected. Following this, the canvas correction algorithm adds new Input and Output components to the partial selection to transform it into a valid Issie schematic. The existing truth table generation process can then continue using this valid schematic.

4.6 Filtering with Constraints

Once a truth table has been generated, it can be filtered using constraints; these can be equality or inequality constraints. Equality constraints on an input or output (IO) are of the form $IO = value$,

where the truth table is filtered such that only rows where the input or output (IO) is equal to the given value are shown. Inequality constraints are of the form $LowerBound \leq IO \leq UpperBound$; the filtered truth table will only contain rows where the IO is between the lower and upper bounds (inclusive). Due to reasons that will be discussed further in this chapter, applying input constraints will re-generate the table, while output constraints will only filter the existing table. Constraints are added through a popup window as shown in Figure 4.4a. Issie validates all constraints in real time; the user is prevented from entering an invalid constraint and is told the why it is invalid. This can be seen at the bottom of Figure 4.4a, where the message informs the user of the issue with the constraint they are trying to add. This approach has two major advantages. Firstly, from a system design perspective, the logic which applies the constraints can trust that all the constraints are well-formed and do not conflict with one another, maintaining robustness. Therefore error handling for constraints need not be implemented in these sections. Secondly, from a user experience perspective, this approach is more obvious and intuitive, aligning it with Issie's core principles. The issue with the user input (invalidity of the constraint) is addressed at the moment it happens, making it clear to the user exactly what has gone wrong. This is far more clear than propagating the error to the user later on in the process. Once added, the constraint will appear as a small tag in the Filter section – the constraint can be deleted by clicking the cross on the tag; alternatively all input or output constraints can be cleared by clicking the *Clear All* button for each respective group.

4.6.1 Constraint Validation Rules

1. All entered values must be valid numbers – numbers can be entered in decimal, hex (0x), or binary (0b) form.
2. All entered values must not exceed the width of the IO, with checks also implemented for negative numbers (e.g. cannot enter 9 or -6 for a 3-bit IO).
3. Constraints must be unique; the entered constraint must not already exist.
4. Constraints must not overlap. For example, if a constraint $X = 5$ exists, the constraint $4 \leq X \leq 7$ cannot be added (and vice versa).
5. For inequality constraints, the upper bound must be greater than the lower bound. This check is always performed using the unsigned representation of the number entered.

Add Input Constraint

The Truth Table is re-generated every time Input Constraints change.

Select Input

MODE B A

Constraint Type

Equality Constraint Inequality Constraint

Edit Constraint

≤ B (2 bits) ≤

Lower Bound and Upper Bound cannot have the same value.

Cancel Add

FILTER

Filter Rows in the Truth Table using Input or Output constraints

Input Constraints

0x0 ≤ B ≤ 0x2 X

Add Clear All

Output Constraints

Add Clear All

(a) Pop-up for Adding an Input Constraint (b) Filter section after adding a constraint

Figure 4.4: Adding a Constraint to the Truth Table

4.7 Hiding Output Columns

The second menu section allows the user to hide (or un-hide) output or viewer columns from the truth table. This means that the user can choose to focus on relationship between the inputs and

a specific output. This may be useful when analysing arithmetic circuits – the user may wish to hide the carry-out column to only focus on the result. As can be seen in Figure 4.2b, each output/viewer has a toggle component which has two options: visible and hidden. These are used to set the visibility of columns in the truth table. The action of hiding at output column takes less than 100ms – this is classed as instantaneous. Achieving this required specific decisions to be made regarding how the truth table was rendered; these will be discussed in Section ??.

4.8 Graphical Manipulation of Truth Tables

Re-ordering Columns On either side of each IO label in the heading row, there are left and right arrows. These allow the user to arrange the truth table as they please by changing the order of columns in the truth table.

Sorting In each IO heading cell, there are a pair of up/down arrows for sorting the truth table by the values of that IO. Selecting the up arrow sorts in ascending order, while selecting the down arrow sorts in descending order. Once selected, an arrow remains highlighted (until another is selected), to inform the user of how the truth table is being sorted. The rationale behind implementing truth table sorting is that it allows users to organise the information they view, increasing the control they have over their learning experience.

Auto-resize with dividerbar The draggable dividerbar can be used to resize the right section. The truth table will dynamically resize to match the set right section width, and contents of cells will wrap to the next line and automatically adjust such that no content is cut off.

All of the operations above happen instantaneously (<100ms); achieving such performance with column-based operations relied on the same decisions as those which made hiding columns appear instantaneous (see Section ??).

4.9 Don't Care Reduction

If the generated numeric truth table is **not truncated**, then the user can reduce it through Don't Care reduction. When the user presses the *Reduce* button, the truth table will be analysed, and any rows where certain inputs are redundant will be collapsed, with the value for that input being replaced by an **X**. The user can always return to the full table by clicking the *Back to Full Table* button. Like their full numeric counterparts, reduced truth tables can also be filtered and sorted. If the truth table is truncated, Don't Care reduction is unavailable; this is communicated to the user by a greyed-out *Reduce* button. Hovering over the disabled button will show the user a popup explaining why the option is unavailable.

As discussed in Section 2.3.1, two possible methods of implementing DC Reduction were considered; either porting an existing heuristic-based minimisation tool to Issie, or writing a reduction algorithm from scratch. Ultimately, the latter was chosen. This is because existing minimisation tools are tailored towards hardware design in industry, where the priority is simplifying the design so it requires fewer components. There is a risk that complicated simplification may in fact obscure the logic further; this is the opposite of this project's intention. Further to this, current minimisation tools like Espresso [15] do not support multi-bit inputs and outputs, which would make integration with Issie's existing framework difficult. Given the decision to not split up multi-bit IOs in Issie, Espresso was deemed unfit. The custom reduction algorithm takes an un-truncated numeric truth table and attempts to reduce it by recursively finding redundancies through row comparisons.

4.10 Algebraic Truth Tables

In addition to numeric truth tables, this project also adds algebraic truth tables to Issie. Instead of numerical values, outputs are instead represented as an algebraic function of their inputs. These algebraic functions are loosely based on Boolean algebra, but do include other operators such as addition and subtraction in order to provide a more useful summary of relationships. Users can introduce algebra into an existing numeric truth table by clicking the *Algebra* button above the

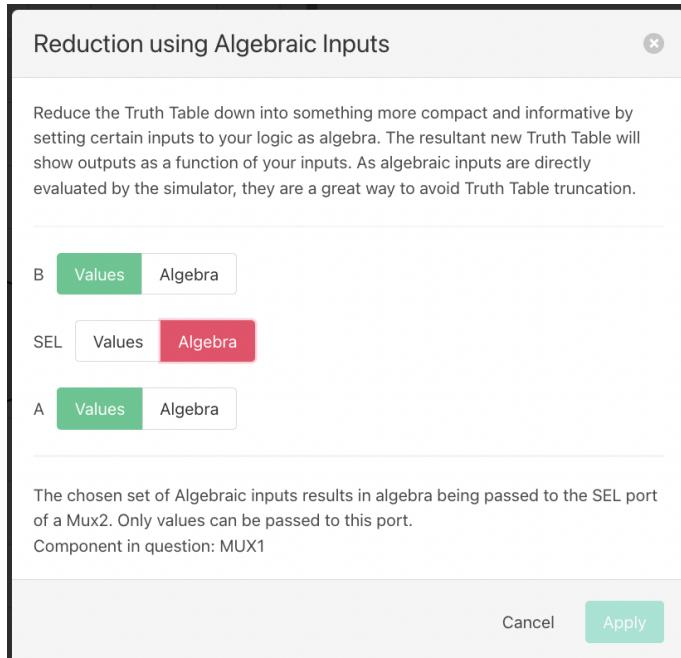


Figure 4.5: Algebra Input Selector popup

truth table. This will spawn a popup, where the user can choose which inputs to set as algebraic, and which inputs should remain numerical. There exist cases when certain inputs cannot be set as algebra, such as when an input is connected to the select (SEL) port of a multiplexer. The user is prevented from setting those inputs as algebraic values, and a helpful error message informs the user why. An example of this can be seen in Figure 4.5

4.10.1 Designing the System for Algebra

The design of the system for the generation of Algebraic Truth Tables was one of the most complex activities undertaken by the project, with numerous decisions being taken regarding which features to implement and the overall approach to take. The requirements stated that support for circuits containing combinations of multiplexers and adders was the bare minimum (**D1.1.1**), but ideally Issie should support algebra for most combinational circuits.

Decision 1: Algebraic Reduction vs Algebraic Generation

It was originally envisaged that algebraic truth tables would be obtained through reduction. An exhaustive numeric truth table would be generated, and algebraic relationships would be inferred from it. This could possibly have been achieved through converting the truth table into SOP form, and then reducing the resultant Boolean equation using algebraic identities or existing logic minimisation tools like Espresso. However, following some analysis of this approach, it was deemed unfit. The requirement for an exhaustive truth table limits algebraic reduction to small circuits, while the nature of multi-bit inputs in Issie makes existing minimisation techniques difficult to integrate into the application. Therefore, it was deemed that reducing a numeric truth table to create an algebraic truth table was not a viable strategy. This meant that algebraic truth tables would have to be generated directly from the schematic.

Decision 2: Schematic Analysis vs Schematic Simulation

Following the conclusion that algebraic truth tables would have to be directly generated from the schematic, two approaches for doing so were considered. The first was to analyse the schematic and attempt to match it to a specific case (e.g. a multiplexer circuit). The idea was that each case would describe a pre-defined algebraic relationship, which would then be returned to the user. It was thought that the approach could be extended to more complex circuits by recursively searching for cases contained within the circuit. The second approach was far more general; introduce a method

for simulating a sheet in Issie with algebraic inputs and outputs. This would be implemented as an extension to the Fast Simulator.

The advantage of the first approach was that Requirement **D1.1.1** would be fulfilled quite easily. Following the experience of writing the Canvas Correction algorithm, it was known that checking for specific cases in the canvas state was possible, and therefore recognition of multiplexer and adder circuits could be implemented quickly. However, extending it further than that could be more challenging; the canvas state is closely related to how the user has built the schematic, and there are many different ways of implementing the same logical function. Therefore, adding support for complex relationships would likely require many cases to be checked. Furthermore, repeatedly checking the canvas state for increasingly complicated cases would increase the generation time. On the other hand, adding algebra to the Fast Simulator would be a lengthy process, increasing the amount of time required to complete **D1.1.1**. However, as the Fast Simulator is already compatible with all correct Issie schematics, it would be easier to extend algebraic simulation, as required by **D1.1.2**. Merely modifying the Fast Simulator would also introduce fewer lines of code into the Issie codebase, making further maintenance easier. Additionally, adding algebraic simulation to the Fast Simulator would mean that the Step Simulator, in theory, would also be able to support algebra.

With all points considered, algebraic simulation was chosen over analysing the schematic. The Fast Simulator has been augmented so that it can receive both numeric and algebraic values as inputs to a simulation. It will then return outputs as functions of the algebraic inputs.

4.11 Considering Logic Input with Truth Tables

In addition to visualising combinational logic, the possibility of using truth tables as a method for combinational logic input was also considered. Issie in its current form only allows for combinational logic input via the canvas; components and connections are manually arranged by the user into a valid schematic. The goal of the digital electronics and computer architecture curriculum at Imperial College is to first build up students' understanding of digital circuit design using schematics, and then eventually transition to the use of Hardware Description Languages (HDLs) in subsequent modules. Component-level schematics tend to focus on the propagation of digital signals through multiple components, while HDLs describe circuits at a behavioural level, focusing on the relationships between inputs and outputs. Defining logic using truth tables could bridge the learning gap between these two concepts; prompting users to shift their design approach towards the more abstracted view of functionally mapping inputs to outputs, while also providing a familiar environment (truth tables instead of Verilog) for defining those functions.

The following system was envisaged: users would be able to define a custom hierarchical component by defining the truth table for that component. This process would begin by the user providing the names and widths of each input and output on the custom component. Using the provided inputs, the input space (left hand side) for the component's truth table would be calculated and displayed to the user with a blank right-hand side. For small input spaces, the user could manually enter the output corresponding to each input row. For larger input spaces, the user would be prompted to set certain inputs as algebra, and populate the outputs with algebraic expressions which were a function of the inputs. The arguments for and against implementing this system were considered, and are presented in Table 4.1. While there was a concern that giving users a way to directly define components may detract from them learning schematic design, it was decided that from an educational perspective the system would improve Issie. However, certain issues with the implementation of algebraic truth tables for logic input were identified. A GUI would have to be developed for users to enter algebraic expressions; these would then have to be lexed and parsed. This activity was estimated to take a significant amount of time. Additionally, the algebraic relationships would have to be converted to Verilog. It was deemed that the educational benefit brought by implementing logic creation through user-entered truth tables was outweighed by the time investment necessary to implement the system and that more tangible returns could be achieved by focusing that time elsewhere.

Pros	Cons
Can help bridge the conceptual gap between component-level design and more abstracted HDL design patterns.	While gate-level design and HDLs are widely used in industry, Issie's algebraic truth table system is not standard. Therefore, using it may not transfer well into later education or industry.
Would decrease the time spent designing specific components, as specific gate level operations need not be considered. This improves the user experience.	Decreasing the number of times students build gate-level designs could itself detract from their learning experience.
Flexible, users can choose to define all cases or use algebraic expressions to define the component. For example, inputs which control certain parameters of operation can be left as numeric values, while operands to functions can be defined as algebraic expressions.	A GUI for entering truth tables will have to be created. Additionally, to support algebra a GUI, as well as a lexer and syntax checker will have to be created – this is will take a significant amount of time. If algebra is not implemented, truth table definitions may only be used to define less complex components, decreasing the effectiveness of the system.
	Issie schematics currently can be exported as Verilog. Therefore, a function for converting a truth table component to Verilog would have to be written. This is straightforward for pure numeric truth tables, but challenging for algebraic truth tables.

Table 4.1: Arguments for and against implementing truth-table defined custom components

Chapter 5

Implementation

This section describes the manner of implementation of all features described in the previous section, discussing specific methods, algorithms and language constructs. The additions made by this project are implemented almost entirely in F#, and the complete codebase can be found at <https://github.com/adidesh20/issie>. The specific in-code changes made by this project can be found by viewing the commit history of the repository. The repository additionally serves as a logbook of completed work; commit messages are time-stamped and provide context to code changes. While this section discusses many interesting aspects of the project implementation, including challenges, compromises, and intelligent design, granular details of the pure coding activities of the project are not given. The code was written and structured with readability, re-usability and efficiency in mind – some examples of these concepts in action are highlighted, however the adherence of the code to these concepts can be verified by inspecting codebase itself.

5.1 Overview of Data Types

Various new F# types have been introduced to Issie by this project. This section describes these types and provides context to them, explaining their function and the rationale behind their creation.

5.1.1 Types for Representing Truth Tables

Data Type/Structure	Information
Cell Data	Discriminated Union type representing what data each truth table cell can hold. Cells can hold Bits (represented by Issie's <code>WireData</code> type), Algebraic expressions (represented by strings), or a Don't Care.
Cell IO	Discriminated Union type representing which input or output of the logic the data belongs to. <code>CellIO</code> s can either be the existing <code>SimulationIO</code> type used to describe Inputs and Outputs, or Viewers.
Truth Table Cell	Record type which represents the contents of a cell in a truth table. Is made up of a <code>CellIO</code> and some <code>CellData</code> .
Truth Table Row	Represents a row in a truth table, which is a list of Truth Table Cells.
Truth Table	Record type containing various <code>Map</code> data structures which map a row of inputs to a row of outputs. Different map data structures are used for caching different versions of the truth table. The record also contains fields and methods which contain or relay other metadata about the truth table.

Table 5.1: Data Types and Structures used for Truth Table Representation

Truth tables in Issie are mostly stored as a mapping from input rows (each containing a single input combination) to output rows. This is implemented using the F# Map type; therefore a truth table map has the type `Map<TruthTableRow, TruthTableRow>`. The F# Map type implements immutable maps based on binary trees, which have a lookup time complexity of $O(\log(n))$ [48]. This is an improvement on traditional lists and arrays, which have a lookup time complexity of $O(n)$. Maps were chosen to store truth tables as it would be faster to look up an output row using a given input row. The `TruthTable` data type is a record containing fields which store cached truth table representations, as well as other data. The fields of this record are described in Table 5.2. Caching different versions of the truth table trades a slightly increased memory usage for time saving, as operations on the truth table do not need to be re-applied every cycle of the view function.

Field and Type	Explanation
<code>TableMap: Map<TruthTableRow, TruthTableRow></code>	The result of truth table generation. Contains the initially generated truth table with input constraints and algebraic inputs applied.
<code>DCMap: Map<TruthTableRow, TruthTableRow></code>	The result of performing Don't Care Reduction on <code>TableMap</code> . Is an Option type, so is none if the table has not been DC Reduced.
<code>FilteredMap: Map<TruthTableRow, TruthTableRow></code>	Cached result of filtering <code>TableMap</code> or <code>DCMap</code> with output constraints.
<code>SortedListRep: TruthTableRow list</code>	The result of sorting the truth table stored in <code>FilteredMap</code> . The truth table is stored as an ordered list because Maps in F# cannot retain a custom order.
<code>IsTruncated: bool</code>	True if the truth table had to be truncated in the generation process.
<code>MaxRowsWithConstraints: int</code>	The number of rows the truth table should have after applying input constraints, but prior to truncation. Used to calculate how many rows have been lost to truncation.
<code>TableSimData: SimulationData</code>	Simulation used for the truth table, cached for use during table re-generation.
<code>IOOrder: CellIO list</code>	List of all CellIOs in the truth table, in the order they originally were when the truth table was generated.
(member) <code>Inputs: CellIO list</code>	Member function which returns a list of all inputs in the truth table.

Table 5.2: Explanation of `TruthTable` Record fields

5.1.2 Constraint Types

This project adds two types of numerical constraints; equality constraints and inequality constraints, the type definitions for which can be seen in Listings 5.1 and 5.2. Equality constraints on an input or output are of the form $IO = value$, where the truth table is filtered such that only rows where the input or output (IO) is equal to the given value are shown. Inequality constraints are of the form $LowerBound \leq IO \leq UpperBound$; the filtered truth table will only contain rows where the IO is between the lower and upper bounds (inclusive). There is also a `ConstraintType` DU which differentiates between Equality (`Equ`) and Inequality (`Ineq`) constraints.

Listing 5.1: Definition for Equality Constraint

```
type EqualityConstraint = {
    IO: CellIO
    Value: int
}
```

Listing 5.2: Definition for Inequality Constraint

```
type InequalityConstraint = {
    LowerBound: int
    IO: CellIO
    UpperBound: int
    Range: int
}
```

A set of constraints is therefore simply a list of all equality constraints combined with a list of all inequality constraints. This is the definition of the `ConstraintSet` type.

5.1.3 Algebra Types

Algebraic Operators

Table 5.7 in Section ?? describes the algebraic operators the user may encounter in Issie. This sub-section describes the F# implementation of these operators. Algebraic operators in Issie are split into three Discriminated Union types: binary operators (`type BinaryOp` which have two operands, unary operators (`type UnaryOp`) which have one operand, and comparison operators (`type ComparisonOp`). Currently, there is only one comparison operator: `Equals`, which compares an algebraic expression to an unsigned integer value. Listings 5.3 and 5.4 show all the different cases of binary and unary operators. There is one key omission from F# binary operators; the Append Operator. Initially, this was implemented as a binary operator, joining the two operands. However, in circuits with multiple connected *MergeWires* components, a chain of nested append operators would form. Figure 5.1 shows such a circuit. Expression 5.1 is the result of performing algebraic simulation when appends were implemented as binary operators. It features three pairs of parentheses which clutter the expression. In contrast, Expression 5.2 communicates the same information without brackets. This cleaner expression was obtained by representing append operations as lists of expressions during algebraic simulation. This list itself is treated as an algebraic expression – the nature of algebraic expressions in Issie is explained in Section 5.1.3. A list representation is also easier to analyse, meaning that specific patterns in appends can be inferred. For example, successive appends of consecutive bits of the same IO ($A[7] :: A[6 : 3] :: A[2]$) can be folded into one Bit Range operation ($A[7 : 2]$).

Listing 5.3: Definition for Binary Operators

```
type BinaryOp =
| AddOp // A + B (
    mathematical addition)
| SubOp // A - B (
    mathematical
    subtraction)
| BitAndOp // A & B (
    bitwise AND)
| BitOrOp // A | B (
    bitwise OR)
| BitXorOp // A XOR B (
    bitwise XOR)
```

Listing 5.4: Definition for Unary Operators

```
type UnaryOp =
| NegOp // -A (
    mathematical negation,
    bitwise two's
    complement)
| NotOp // bit inversion (
    bitwise XOR with -1)
| BitRangeOp of Lower:int
    * Upper:int // A[upper:
    lower] (subset of bits
    of A)
| CarryOfOp
```

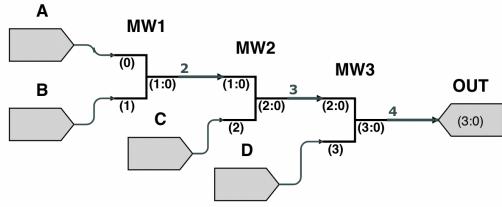


Figure 5.1: Circuit with multiple *MergeWires* connected to each other

$$OUT = (D :: (C :: (B :: A))) \quad (5.1)$$

Append as a Binary Operator

$$OUT = D :: C :: B :: A \quad (5.2)$$

Append as a list of appended expressions

Algebraic Expressions

Using a DU type, a short grammar was written to represent algebraic expressions in Issie. The grammar writing process in particular validated the choice of F# as a programming language for the project; the DU type allows for the grammar to be defined succinctly (only 7 lines of code), while pattern matching on DUs enables evaluation of an expression through a single recursive function. Large class hierarchies are therefore not required. Algebraic expressions in Issie are defined by the type `FastAlgExp` – the prefix *Fast* is used as the algebraic expressions are used within the Fast Simulation.

According to the grammar, an algebraic expression (`FastAlgExp`) in Issie can be:

1. `SingleTerm` of `SimulationIO`: Represents a single algebraic term, which is an input. This case has the type `SimulationIO` as every input in Issie is represented by that data structure.
2. `DataLiteral` of `FastData`: Represents a numeric value in the simulation.
3. `UnaryExp` of `Op`: `UnaryOp * Exp`: `FastAlgExp`: Represents a unary expression. The unary operator (`Op`) takes the expression (`Exp`) as its operand. An example would be $-A$.
4. `BinaryExp` of `Exp1`: `FastAlgExp * Op`: `BinaryOp * Exp2`: `FastAlgExp`: Represents a binary expression, in which a binary operator (`Op`), such as '+', operates on two expressions. `Exp1` is the left operand, `Exp2` is the right operand.
5. `ComparisonExp` of `Exp`: `FastAlgExp * Op`: `ComparisonOp * uint32`: Represents a comparison between an algebraic expression (`Exp`) and a numeric value.
6. `AppendExp` of `FastAlgExp list`: Represents an expression that is made up of a group of existing expressions appended together. The list is ordered such that the most significant bit is at the head of the list. F# lists are implemented as singly-linked-lists, therefore adding a new item to the head of a list is an $O(1)$ operation. Therefore, having the MSB at the head of the list means that appending something further onto the `AppendExp` is efficient.

5.1.4 The `TableInput` Data Type

The `TableInput` data type was used in the implementation of an earlier algorithm for generating the input space of the truth table. The data type, and the method that requires are no longer used in Issie. However, as the method is described in the report, details of this data type are included for context.

As discussed in Section 4.4.2, the method for generating a limited input space needs to know exactly which specific input combinations to generate out of potentially billions of possible combinations. To achieve this, a new data structure for representing inputs was required; one which stores more than the `SimulationIO` type. The `TableInput` is a record type, and its fields are described by Table 5.3. The term *Row Count* in the context of the input space generation refers to the size of the set (S_i) of unique values an input (x_i) can contribute to a table.

5.1.5 Table Manipulation Data Types

The following data types are used in messages sent from the view function to the update function to indicate that some UI interaction has occurred which requires updating certain parts of the model.

Field and Type	Explanation
IO: <code>SimulationIO</code>	Inputs in Issie are represented by the <code>SimulationIO</code> type, which contains the Component ID, Component Label, and the width of the input.
IsAlgebra: <code>bool</code>	True if the input is algebraic, as opposed to numeric
MaxRowCount: <code>int</code>	The total size of the Set S_i , which is equal to 2^{w_i} , where w_i is the width of the input.
ConstrainedRowCount: <code>int</code>	The size of the subset of S_i which contains all input values which conform with the input constraints. For example, if there were the following constraints on the input: $0 \leq x \leq 6 \ \& \ x = 9$, the Constrained Row Count would be 8. Constrained Row Count is always less than or equal to Max Row Count.
AllowedRowCount: <code>int</code>	The number of input values the given input is actually allowed to contribute to the truth table. Ideally, this is equal to the Constrained Row Count, but in situations where the truth table has to be truncated, the Allowed Row Count may be limited to keep the total number of rows in the truth table under the overall limit.

Table 5.3: Explanation of Fields in the `TableInput` data structure

Truth Table Sorting direction

Listing 5.5: Definition for Sort Type

```
type SortType = | Ascending | Descending
```

As shown in Listing 5.5, sorting can be done in ascending or descending order.

Changing the Order of Columns

Listing 5.6: Definition for Movement Direction

```
type MoveDirection = | MLeft | MRight
```

As shown in Listing 5.6, columns can be moved left or right.

5.2 Top Level UI Changes

5.2.1 Simulation Sub-tabs

All activities that involved some form of circuit simulation were organised under the **Simulations** tab (formerly called Simulation) using sub-tabs. The right section tabs are implemented with the Fulma [36] Tabs component. The Sub-tabs are implemented by nesting a second Fulma Tabs component within the body of the Simulations tab, and making ensuring that a sub-tab is visible only when it and the parent tab is open.

5.2.2 Moving the Waveform Simulator

Due to its inconsistent placement and strange tab-spawning behaviour, the decision was made to move the waveform simulator into its own, permanent sub-tab. The move was quite straightforward; the *Waveforms* button was moved to the sub-tab, a greeting was also displayed. However, one small change had to be made to how the waveform simulator operated. When a waveform simulation is running, other tabs in the app are inaccessible by design. Previously, the waveform simulation tab would only open when the user was either starting or viewing a waveform simulation. Therefore, the existence of the tab was a valid metric for ascertaining whether the app should make other functions inaccessible. Following the changes, however, the existence of the waveform

simulation tab is independent of whether a waveform simulation is running. Therefore, a new pair of messages `LockTabsToWaveSim` and `UnlockTabsFromWaveSim` can be used to control when the user is locked into the Waveform Simulator.

5.2.3 Dynamic Dividerbar Resizing

The dividerbar is situated between the canvas and the right section and marks the boundary between the two. During waveform simulation, the dividerbar becomes draggable to let the user view more content by resizing the right section. This functionality was also extended to truth tables. However, there was a bug in the implementation of the dividerbar. The CSS Style for the dividerbar element sets its height to 100% of the right section. If the height of the content in the right section exceeds the initial height of the right section, the latter *overflows* and becomes scrollable. The 100% height styling on the dividerbar did not take the overflow into account, meaning that the dividerbar would slowly disappear off screen as the user scrolled down. This issue was fixed by getting the right section `div` element from the DOM Tree, and reading its `scrollHeight`, which takes overflow into account.

5.3 Generating Truth Tables

Figure 4.3 in Section 4.4 described the high-level method for generating truth tables in Issie. Truth table generation in Issie can be broken down into three broad stages: 1. building the simulation, 2. calculating the input space, and 3. simulating each combination in the input space.

5.3.1 Building the Simulation

As explained in Section 2.6, the process of building a simulation is distinct from the process of simulating an input combination. The purpose of this stage is to transform the user-entered sheet (represented by the `CanvasState` data structure) into `SimulationData`. In the interest of consistency for both the user and future developers, as well as ease of maintenance, the truth table generation code either uses the Step Simulator code, or conforms with its design language. From the *Truth Table* tab, users can generate truth tables for the whole sheet, or for a partial selection of the sheet. When the tab is open, Issie will first check that the logic is combinational, and then try to build a simulation in the background. The simulation building process for the whole sheet is identical to that of the Step Simulator: the existing function `makeSimData` from the module `SimulationView` is used. A simulation is built for partial selections of sheets using a newly written function: `makeSimDataSelected`. The implementation of this function will be explained in Section 5.4. The simulations generated for both the whole sheet and partial selection are cached – this is done to avoid unnecessary rebuilding of the simulation every view cycle. If the simulation building process returns valid `SimulationData`, a truth table can be generated. Therefore, the user is shown a green *Generate Truth Table* button. If a `SimulationError` is returned, the button will be yellow instead and clicking it will display the error message.

5.3.2 Calculating the Input Space

As discussed in Section 4.4, the complete input space of a truth table is made up of all possible input combinations; the complete left-hand side of an exhaustive truth table. For schematics with large inputs and/or a large number of inputs, generating the complete input space is impractical. Instead, Issie only generates a subset of the input space; truth tables are limited to 1024 rows. This limit is defined as `TTBitLimit = 10` in the `Model`, as only 10 bits of input information may be simulated ($2^{10} = 1024$). If the input space (and therefore the truth table) exceeds this length, it is truncated. Input constraints are applied prior to truncation to ensure that the rows the user wants to view are guaranteed to be present in the returned truth table. The limit of 1024 combinations for the input space was obtained by experimentally varying the set `TTBitLimit` in the model. Requirement **E1.7** states that truth table generation should ideally take under 2 seconds, and at a maximum not more than 4. Other graphical operations in Issie must appear instantaneous; the time cut-off for this is 100ms. Therefore, the generation and view times for truth tables were measured for differing bit limits. The results obtained after testing four different

values are presented in Table 5.4. All measurements were made on a Macbook Air with an M1 CPU, generating a truth table for an 8-bit ALU which had a total input space of 2^{23} .

One observation for both categories is that every time the bit limit increases by one, the reported times double. This indicates that the times are directly proportional to the number of rows in the truth table, and that table size is the main variable when considering performance in this scenario. A bit limit of 12 or higher was instantly rejected, as generation time was greater than 2 seconds (not optimal), and view time was almost 200 ms (unacceptable). The high view time meant that there was noticeable lag during graphical operations on the truth table, as well as when the dividerbar was dragged. While a bit limit of 11 is acceptable, with average generation time of 1.4 seconds, Robert Miller [41] stated that response times under 1 second do not lose the user's attention. Therefore, if the choice was made to use a bit limit of 11, there would be a risk of losing the users attention. Additionally, Issie is run on a variety of devices, some which may be less powerful than the device used during the experiment. Considering that the view function time of 87ms is close to the 100ms mark, the decision was made to leave some performance headroom and choose a bit limit of 10, therefore limiting the size of truth tables to 1024 rows.

Bit Limit	9	10	11	12
Avg. Generation Time (ms)	390	702	1432	2819
Avg. View Time (ms)	23	45	87	197

Table 5.4: Average Truth Table Generation and View times with varying Bit Limits

First Truncation Method

The first approach taken to truncate the input space involved calculating exactly which input combinations could be generated, and subsequently only generating those. For each input x_i , three sets are considered: S_i representing all of the possible values the x_i , C_i representing all possible values of x_i after applying input constraints, and A_i representing how the distinct values the input could be allowed to have if the size of the Cartesian product was to be less than or equal to 1024. The sets S_i and C_i may be very large, therefore they are never generated. Instead, their sizes are inferred. The size of M_i can be found by raising the width of x_i to the power of 2, while the size of C_i can be found by calculating the sum of the ranges of the constraints on x_i . In the `TableInput` data structure, the `MaxRowCount` field corresponds to the size of M_i , and the `ConstrainedRowCount` field corresponds to the size of C_i . The size of A_i can then be calculated by folding a list containing the sizes of $C_1 \dots C_n$. The F# implementation of this is shown in Listing 5.7; the `AllowedRowcount` field in each `TableInput` is being populated using the constrained row counts.

Listing 5.7: Calculating Allowed Row Counts

```
(1, sortedInputs)
||> List.mapFold (fun rowCount ti -
    let newRowCount = rowCount * ti.ConstrainedRowCount
    let capacity = limit / rowCount
    // Case where constrained values of this input can be entirely
    // included
    if capacity >= ti.ConstrainedRowCount then
        {ti with AllowedRowCount = ti.ConstrainedRowCount},
        newRowCount
    // Case where constrained values of this input must be
    // truncated
    else if capacity > 1 then
        {ti with AllowedRowCount = capacity}, newRowCount
    else
        {ti with AllowedRowCount = 1}, newRowCount
)
```

Second Truncation Method

Following a code review meeting with the Project Owner, it was mentioned that while effective, the implementation of the first truncation method was not very clear to the reader, and therefore future developers may find it hard to maintain it. The suggestion was made that an alternative approach be investigated, preferably using F# Sequences. Sequences in F#, written in code as `seq`, are a logical series of elements all of one type. Sequences are particularly useful when working with a large, ordered collection of data whose elements may not all be used [49]. Sequences implement a form of *Lazy Evaluation*, a technique in which the evaluation of expressions is delayed until their values are required [50]. If C_i has a theoretical size of 4 billion and is stored as a `list`, all 4 billion of those elements will have to be generated and stored when the list is defined, causing the program to freeze. In contrast, as sequences are lazily evaluated, a `seq` of 4 billion elements can safely be defined and operated on in a limited manner. Care must be taken to avoid operations which cause the sequence to be evaluated while it is still very large.

The first step in finding the input space is to separate the numeric and algebraic inputs. If the truth table is being generated for the first time (i.e. not re-generation), all inputs will be numeric. For each numeric input x_i , is transformed into its constrained set C_i , which is stored as a sequence to avoid early evaluation. If an input has no constraints applied to it, as it is during first time generation, C_i is equivalent to all possible values ($0..2^w - 1$). When constraints are applied, C_i can be found by first defining a sequence for the values allowed by each constraint, and then appending them all together. In the next step, the Cartesian product of sets C_1 to C_n is calculated. Finding the Cartesian product of sequences is a cheap operation because all of the combinations are not actually evaluated. This is in stark contrast to the same operation on lists, which is very expensive. The resultant sequence represents the whole constrained input space, however it may be too large to evaluate and must therefore be truncated to 1024 rows. This is done by the `Seq.truncate` function, which evaluates and returns the first 1024 elements in the sequence: the truncated numeric input space. Following this, any algebraic inputs are appended to each row to yield the complete left-hand side of the truth table.

Comparison of the two methods

The performance of each method was tested by generating a numeric truth table for an 8-bit ALU. The ALU has 5 inputs: A (8 bits), B (8 bits), F (3 bits), X (3 bits), and CIN (1 bit). The total number of bits is therefore 23, much greater than the bit limit of 10. This means that the complete input space comprises $2^{23} = 8,388,608$ combinations. Six readings were taken for each method, with the results shown in 5.5. On average, Method 2 is around 3ms faster than Method 1. An unpaired two-tailed t-test using a 95% confidence interval was performed on the collected data, and found that this difference was not statistically significant. Furthermore, to the human eye 3ms is an intangible time difference [41]. Therefore, from a user perspective, the choice of method is irrelevant.

Method 1 Time (ms)	Method 2 Time (ms)
701	697
702	702
712	700
702	697
702	702
703	704
Avg = 703.67	Avg = 700.3

Table 5.5: Time taken by each method to generate a numeric truth table for an 8-bit ALU

However, from a developer perspective, there is a tangible difference between the methods. Under Method 1, the `tableLHS` function and the functions it called were a total of 143 lines (including comments). Additionally, developers would have to understand the `TableInput` data structure and the two-stage process of finding the row counts and then generating the input combinations. In contrast, Method 2 comprises only 86 lines and only uses data structures from the standard F# Collections library. Requirement **E3.3** states that the project should aim to deliver code that is

easier for future Issie developers to maintain. As a result, Method 2 was chosen over Method 1 for generating the input space of the truth table.

5.3.3 Simulating each Combination

The input space is represented as a list of `TruthTableRow`s – each row represents an input combination. Each input combination is simulated using the function `changeInputBatch`. This function takes as input a list of changes to be made and mutates the input values in the `FastSimulation`, then re-runs the combinational simulation. This causes the output fields in the `FastSimulation` to change – these are then extracted and stored in another `TruthTableRow`. The given input and output rows are stored in a `Map` as a pair. The result of the whole process is a F# `Map` which contains mappings from inputs to outputs; i.e. a truth table.

5.4 Generating Truth Tables for a partial selection of a Sheet

Issie uses the same underlying three-step process for generating truth tables for partial selections: building a simulation, calculating the input space, and simulating each input combination. The only part of the process that is different for partial selections is the simulation building process – the function `makeSimDataSelected` does this. The selected `CanvasState` consists of a list of selected connections and components. However, in most cases a simulation cannot be built from the selected `CanvasState`. This is because it is likely lacking a full set of input/output components, as well as connections to specific ports. Issie combats this by intelligently correcting the canvas. Following Canvas Correction, any returned `CanvasState` is fully compatible with the existing functions for simulating logic and generating truth tables. Therefore, a Truth Table can be generated for selected logic through the same methods and functions used for generating Truth Tables for the whole sheet.

5.4.1 Canvas Correction

Canvas correction needs to be robust and handle as many selection cases as possible, so that the user does not get frustrated with needless errors when trying to build a truth table for selected logic. The correction algorithm was quite challenging to write, as all possible selection styles had to be considered and handled, with correction being implemented for the majority of them. Figure 5.2 shows four common examples of user selections, and the summary of Canvas Correction found below uses these cases to showcase how the algorithm handles different input. Additionally, Figure 5.3 visually describes what each step in the algorithm does. It should be noted that the images in Figure 5.3 are for the reader's understanding only – these individual step views are not shown to the user.

Step 1 Remove Duplicate Connections: In a given partial selection, a component output port may have multiple connections connecting it to other components. In the case where the component in question and the connections are selected, but the other components are not, there are multiple *dangling* connections connected to the output port. These connections would all eventually be connected to a newly created output component, but these outputs would all have the same value as they would be connected to the same port. Therefore, connections which do not have both ports in the selection are removed from the selected `CanvasState`.

Step 2 Add Extra Connections: Sub-figures 5.2b to 5.2d in Figure 5.2 all show situations where one or more inputs/outputs for the selected logic are ports on components, rather than connections. Taking the case shown in Figure 5.2b in particular, the inputs to the selected logic are: both input ports on G1, and the bottom input port on G2, while the outputs from the selected logic are both output ports on G1 and G2 respectively. Prior to correction, the selection canvas does not have any connections going into those ports. This step finds any ports on selected components which do not have a connection in the selection and connects them to "dummy" input or output ports depending on their `PortType`. This transforms the canvas to a state similar to that seen in Case (a), where all inputs into the selected logic have connections.

Step 3 Add Extra IOs: This step adds the "phantom" input and output components to the selection canvas. The locations where these components need to be inserted are found by checking which connections in the Canvas State do not have both ports present in the selection. Any such connections are either connected to some other component in the sheet which is not selected, or are newly added connections which are connected to dummy ports. Either way, these are the connections that need to be connected to "phantom" IOs.

Step 3.1 IO Width Inference: When creating new input or output components, the correct width must be specified. This is calculated by running Issie's `WidthInferrer` on the whole sheet to find the expected width of the input or output. However, in cases such as the one shown in Figure 5.2d, where there are no connections to/from some ports on G3, `WidthInferrer` will fail to infer widths. In this case, the port width is inferred from the host component; all logic gates ports in Issie have a width of 1. Component-based width inference is implemented for:

- * All Logic Gates
- * Inputs, Outputs, and Constants
- * Bus Select and Bus Compare
- * N-bits Adder and N-bits Xor
- * IOLabels, if they are connected to a component who's port width has been inferred

Step 3.2 IO Label Inference: Usually users provide labels for IOs, which are used in the Truth Table. However, when IOs are automatically created, names for them must be automatically generated too. This is done by looking at which ports in the selection they are connected to. If the port is labelled (e.g. on a multiplexer or a Custom Component) the expression for an automatically generated IO Label is: `[Connected Component Label]_[Port Label]`. Alternatively it is `[Connected Component Label]_[IN/OUT]_[Port Number]`.

Step 4 Returning a Canvas or an Error: If any errors were found in the previous steps, most likely due to a malformed selection, they are returned. If not, then the returned Canvas State is completely compatible with the existing simulation and truth table generation functions. Possible reasons for returning an error are:

- No Components, only Connections are selected
- Selected logic contains a wire connected to no components
- There is a legitimate error in the selected logic
- Width for an input or output into the selected logic could not be inferred by `WidthInferrer` or from host components

5.4.2 Caching Strategy

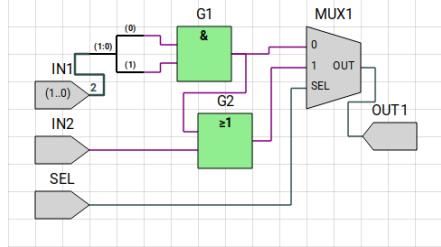
The function `makeSimDataSelected` is called in every iteration of the View function, as its return value (either a successfully built simulation or `SimulationError`) is used to determine the colour, text and action of the generation button for partial selections. Repeatedly correcting the selected canvas on every call is wasteful if the selection has not changed, the result of the process is cached and re-used so long as the selected canvas state has not changed. The type definition for this cache can be found in Listing 5.8. When `makeSimDataSelected` is called, the selected Canvas State is reduced and compared to the `UncorrectedCanvas` field in the cache. If they are equal, this indicates the selection has not changed, so the corrected canvas and stored simulation result from the last call can be returned. If the selection has changed, then the selected canvas is corrected, and the cache is updated with the latest results.

Listing 5.8: Type Definition of Cache for selected logic

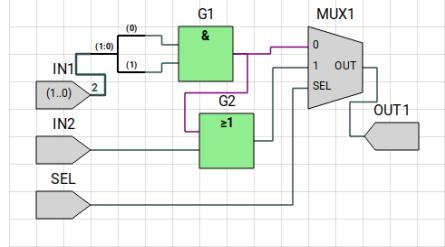
```
type SelectionCache = {
    UncorrectedCanvas: CanvasState
    CorrectedCanvas: CanvasState
}
```

```
    StoredResult: Result<SimulationData, SimulationError>
```

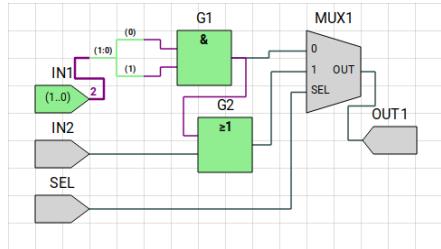
```
}
```



(a) Case where all inputs into selected logic are connections.



(b) Case where some inputs into selected logic are connections, and some are ports on components.



(c) Case where selected logic includes an input component.

(d) Case where a selected component does not have connections to all ports

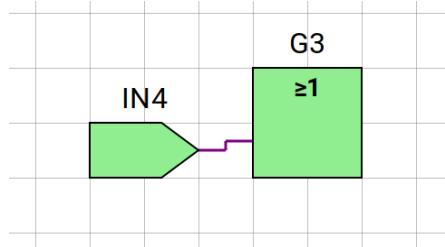


Figure 5.2: Selection Cases

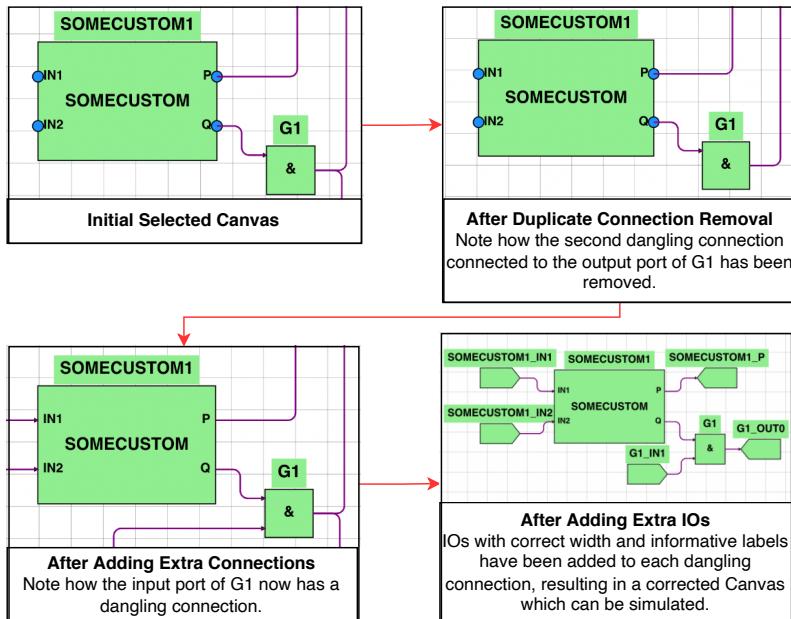


Figure 5.3: Representation of the Canvas at different stages of Canvas Correction

5.5 Filtering with Output Constraints

While input constraints are applied when generating the input space for performance reasons, output constraints are applied to the generated truth table. This is done for performance reasons; it is quicker to iterate through an existing truth table and filter specific rows compared to re-generating the whole table and only including rows which match the constraints. All output constraints (equality and inequality) are placed in a list, this list is passed to `List.fold`, along with the truth table (`Map<TruthTableRow, TruthTableRow>`), which is the initial state. Each call

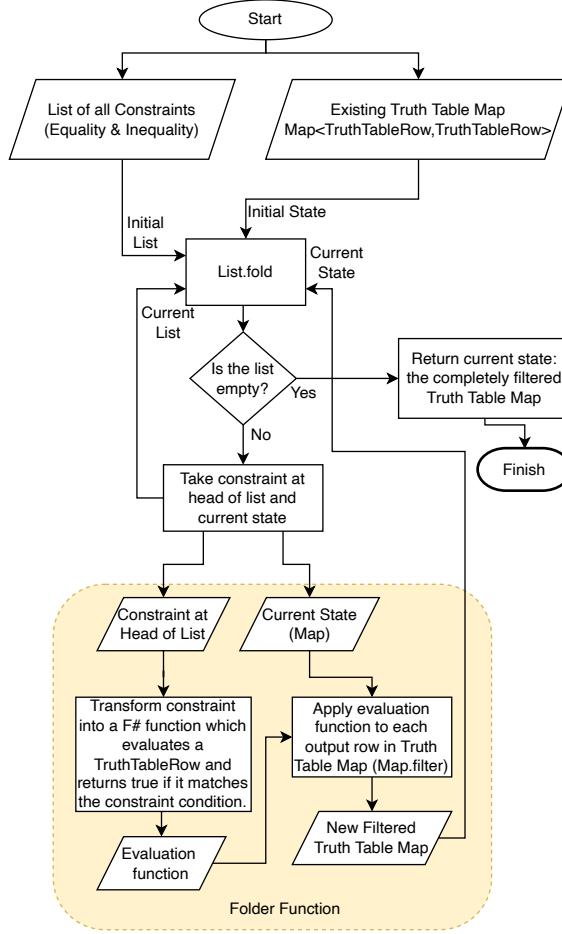


Figure 5.4: Method for Filtering the Truth Table with Output Constraints

of the folder function applies the next output constraint to the Map, which shrinks with every iteration. Figure 5.4 shows a visual description of this process.

5.6 Don't Care Reduction

Truth table reduction using Don't Care terms is implemented using a recursive algorithm which attempts to keep reducing the truth table until there are no redundancies in the table. A recursive function is used as a single row may contain multiple Don't Care terms, meaning that redundancies may still exist in the truth table after a single round of reduction. As discussed in Sections 2.3.1 and 4.9, Don't Care reduction in Issie differs from similar logic minimisation techniques as columns in truth tables can have multi-bit values instead of purely zeros and ones. Issie focuses on teaching digital design principles, therefore on displaying relationships in the logic is more important than achieving the most minimal implementation. All functions related to the reduction of truth tables can be found in the file `TruthTableReduce.fs`. Due to the complex nature of the recursive function, it would be impractical to reduce very large tables. Therefore, only **non truncated** tables can be reduced using Don't Cares. Larger schematics ought to be reduced algebraically.

An example of Don't Care Reduction in action is shown in Figure 5.5. Figure 5.5a shows a schematic in which the Issie component *Mux4* takes 4 2-bit inputs into its data lines, and a 2-bit input into its select line. The full numeric truth table for this schematic has 1024 rows. Applying the algorithm to the table yields a table (5.5b) with only 16 rows: a reduction of over 98%. Rows in this table have multiple DC terms.

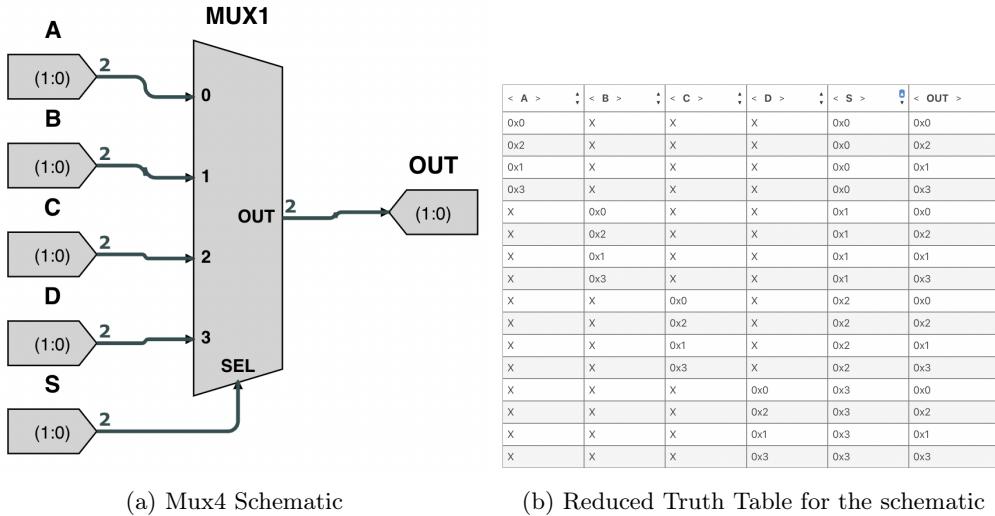


Figure 5.5: Schematic and Reduced Truth Table

5.6.1 Prerequisite Concepts

Prior to explaining the reduction algorithm, certain key concepts regarding truth tables are covered. A DC row is defined as a row that contains Don't Care terms.

Row Equality Truth Table Rows are simply lists of cells, and these lists are compared element-wise to check for row equality. A cell is equal to another cell if they have the same IO and Data values. However, if either of the cells have a Don't Care term, the cells are considered to be equivalent as a Don't Care term can be equal to any value. Two rows can be compared using the `rowEquals` function.

Looking up Rows in the Truth Table The truth table is implemented as a Map data structure, and therefore has efficient lookups. However, the existing `Map.tryFind` function uses default F# equality checking when performing lookups. In order to support DC rows, the equality function `rowEquals` should be used. The function `tableTryFind` works like `Map.tryFind`, but supports querying the table with Don't Care terms in rows. As a Don't Care term represents multiple values, one row can map to multiple rows. Therefore, `tableTryFind` returns a list of all output rows that match the input row.

Validity of a Row with Don't Cares Within a truth table, a row containing Don't Care terms is considered valid if the relationship it describes is correct – i.e. is the variable actually redundant in the given input combination. This is tested by finding all output rows that correspond to the input DC row using `tableTryFind`. If all of the returned output rows are equal, then the DC row is valid. If not, then it is invalid.

5.6.2 Reduction Algorithm

The algorithm takes an instance of the Truth Table data structure as input, and returns a new instance of the data structure, this time with the `DCMap` field populated with a Map. As the algorithm is called recursively on the input data structure, the algorithm first checks if a `DCMap` exists. If it does not, then this is the first call and therefore the numeric `TableMap` is used. If the field is populated, then this is a subsequent reduction of the `DCMap`, so it is used. Following this, the algorithm can be split into three stages:

Stage 1: Find all valid Don't Care Rows

Listing 5.9: Finding all DC Rows

```
let allDCRows =
```

```

table.Inputs
|> List.collect (fun input ->
    inputDCRows input inputConstraints table bitLimit)

```

The function `inputDCRows` finds all DC rows for a given input. This function is called for each input, and the returned rows are all collected. `inputDCRows` first generates all possible DC rows for the input by replacing any numerical values of the given input in the table with a Don't Care term. These possibilities are then validated against the truth table using the function `isValidDCRow`, with invalid rows being discarded from the list. This leaves only valid DC rows.

Stage 2: Find which rows from the Truth Table should remain

Listing 5.10: Finding which regular rows should remain in the Truth Table

```

let remainingRegularRows =
    (Map.toList tMap, allDCRows)
    ||> List.fold reduceWithDCRow

```

Once all valid DC rows are found, these rows are used to reduce the existing table. The process of reducing the table using DC rows is similar to the process of filtering the table using output constraints. Every row in the truth table is compared to every valid DC row using `rowEquals`. If a row in the existing truth table is equal to a DC Row, it means that it is redundant in the table and should be removed. This process returns a list of the rows from the existing truth table that were not made redundant by the DC rows and should therefore remain in the truth table.

Stage 3: Assembling the Reduced Table and Recursive call

By combining the DC rows and remaining regular rows, the reduced truth table for this round of reduction can be obtained. However, one round of reduction can only add one Don't Care term to each row. However, certain truth tables may require multiple DC terms in each row to be fully reduced – Figure5.5b is an advantage of this. Therefore, the reduction algorithm is called recursively on the reduced truth table. This chain of recursive calls continues until the reduced tables generated by two successive rounds are found to be equal, as this indicates that the table can be reduced no further. This fully reduced table is returned to the user for viewing.

5.7 Algebraic Truth Tables

Once a truth table has been generated, the user can reduce the number of rows in the table by changing inputs from numerical values to algebraic expressions. Section 5.1.3 discussed in detail the nature of algebraic operators and expressions in Issie, while Table 5.7 in the Analysis and Design chapter explained what every operator the user may encounter means. Algebraic truth tables are obtained through *Algebraic Simulation*; specific inputs selected by the user are fed to the underlying Fast Simulation as `SingleTerm` algebraic expressions, and these expressions propagate through the simulation and are manipulated by operators. The outputs of the simulation are therefore also algebraic expressions, which summarise the output in terms of the input.

5.7.1 Definition of the Algebraic System

A new system of algebra was created by this project for use in algebraic truth tables in Issie. The aim of the algebra is to clearly communicate the semantic meaning of the combinational logic function described by the schematic. These functions are traditionally defined using Boolean algebra, but this system was deemed unfit for use in algebraic truth tables. Simplified Boolean algebra can describe which operations are being performed on the inputs at a hardware level, but lacks a way to succinctly describe more advanced operations which consist of multiple regular Boolean operations. Arithmetic operations are an example of this. While Issie supports multi-bit inputs and wire buses in logic, traditional Boolean algebra assumes all input variables have a width of 1. Therefore it does not include operators for manipulating buses such as slicing or merging. Due to these reasons, the decision was made to define a new formal language (algebra) which would be a superset of Boolean algebra, adding support for arithmetic and multi-bit bus operations. Formal languages are a set of sequences, or strings over some finite vocabulary, defined using a grammar. A formal grammar is defined as a quadruple $\langle \Sigma, NT, S, R \rangle$, where Σ represents terminal symbols, NT represents non-terminal symbols, S represents a start symbol, and R is a set of production rules which recursively transform non-terminals into terminals [51]. In 1956, Noam Chomsky [52] defined four types of grammars, which are arranged in a hierarchy. These can be summarised as:

Unrestricted Grammars (Type 0): Set of all language grammars that computably enumerable; i.e. can be recognised by a turing machine.

Context-Sensitive Grammars (Type 1): Language grammars where the left-hand side of a production rule can contain multiple non-terminal and terminal symbols.

Context-Free Grammars (Type 2): Language grammars where the left-hand side of every production rule contains only one non-terminal.

Regular Grammars (Type 3): Grammars which can be processed by state machines, where each non-terminal can be considered as a 'state', and production rules are essentially state transition rules.

Each type in the hierarchy is a subset of the previous type; so for example all Type 3 grammars can be described with a Type 2 grammar, which can be described by a Type 1 and so forth. The algebra was initially defined as a context-free grammar, with each component in the schematic generating a single expression at each of its output ports from expressions at its input ports by applying production rules. This yielded an algebra which accurately described each transformation applied to the inputs. However, this algebra lacked any form of intelligent simplification: ways in which a generated expression could be reduced down to its most concise or informative representation. This was addressed by augmenting the set of production rules with **reduction rules**, which in turn led to the defined grammar becoming context-sensitive.

Note: The F# implementation of the algebra differs slightly in some places from the more formal definition described below to allow for easier comprehension and simplification. The F# type definitions are described in Section 5.1.3.

Terminal Symbols

A terminal symbol is the simplest constituent of an algebraic expression, which cannot be subdivided or simplified further. In the defined algebra, terminal symbols can be split into two

categories. The first category represents the base value type which operations can be performed on. There are two terminal value types in the algebra; these are described in Table 5.6. The second category consists of the operators which manipulate expressions. These are described in Table 5.7. A key difference between this formal definition and the underlying implementation is the Append Operator. From the perspective of the user who will read the algebra/language, a binary operator which joins two expressions does exist. However, to aid easier reduction, in the F# implementation appended expressions are stored in a list of expressions, which is itself defined as an expression.

Symbol	Name in Code	Explanation
Variable	SingleTerm	A symbol representing an algebraic input into the simulation. Any valid IO Label string can be a variable.
Number	DataLiteral	A known numerical value in the simulation. Used to represent numerical inputs into the simulation.

Table 5.6: Terminal Value Symbols

Symbol	Name in Code	Explanation
Numeric Addition: +	AddOp	Binary operator which represents mathematical addition of its two operands.
Numeric Subtraction: -	SubOp	Binary operator which represents mathematical subtraction of its right operand from its left operand.
Bitwise And: &	BitAndOp	Binary operator which for a Boolean And between each bit of its operands (bitwise).
Bitwise Or:	BitOrOp	Binary operator which for a Boolean Or between each bit of its operands (bitwise)
Bitwise Xor: \oplus	BitXorOp	Binary operator which for a Boolean Xor between each bit of its operands (bitwise)
Numeric Negation: -	NegOp	Unary negation operator, indicates that the single operand will have its sign inverted
Bitwise Not: ~	NotOp	Unary bitwise Not operator, indicates that all bits of the single operand will be inverted
Carry Of: <i>carry()</i>	CarryOfOp	Unary operator which signifies that the result is the single-bit carry-out from an arithmetic expression. This expression is the operand, which is situated in the parentheses.
Bit Range: $[u : l]$	BitRangeOp	Unary operator with two parameters: an upper and lower bound. Indicates that a specified range of bits (inclusive) will be selected from the operand.
Equals: ==	Equals	Logical operator which checks for equivalence between some algebraic expression and a numeric value.
Append: ::	Not in Code	Binary operator which joins the bits of the two input operands into one result whose width is the sum of the input widths. The left operand bits become the MSBs, right operand bits become LSBs.

Table 5.7: Terminal Operator Symbols

Non-Terminal Symbols and Start Symbol

Non-terminal symbols are constructs that appear within algebraic expressions which, through the application of defined rules, can eventually be simplified/replaced with terminal symbols. In the F# implementation, arithmetic and boolean operators are not defined, as they are subsets of all binary operators. Grammars will generally build up into a tree-like structure, with the terminal symbols as its leaves and non-terminals as intermediate nodes in the tree. The root of this tree is known as the **start symbol**. In a formal grammar, it should be possible to arrive at a sequence of terminal symbols by repeatedly and/or recursively applying the defined production rules to the start symbol. In the case of the defined algebra, the start symbol is the *expression*.

Symbol	Name In Code	Explanation
Binary Operator	BinaryOp	Defines all binary operators (operators which take two operands) and can be substituted by any.
Unary Operator	UnaryOp	Defines all unary operators (operators which take one operand) and can be substituted by any.
Arithmetic Operator	Not in Code	Defines all operators involved in mathematics.
Boolean Operator	Not in Code	Defines all operators from Boolean algebra.
Binary Expression	BinaryExp	Defines an expression in which a binary operator acts on its operands.
Unary Expression	UnaryExp	Defines an expression in which a unary operator acts on its operands.
Comparison Expression	ComparisonExp	Defines an expression in which a comparison operator compares a symbol with a value
Expression	FastAlgExp	Highest level non-terminal symbol (start symbol), defines all expressions in the simulation.

Table 5.8: Non-Terminal Symbols

Production Rules

Generally, production rules in grammars have the form $\alpha \rightarrow \beta$, understood as α may be replaced by β , where α and β are sequences comprising symbols from the terminal and non-terminal set of symbols [51]. In context-free grammars, α represents only one non-terminal symbol. Production rules can be described in *Backus-Naur Form* (BNF); this is a format for presenting grammars often used when defining programming languages. In BNF, non-terminal symbols, which are called *metalinguistic variables* are enclosed in brackets to differentiate them from terminal symbols. The symbol ::= indicates *metalinguistic equivalence*: whatever is to the left of the symbol can be substituted by whatever is to the right. The symbol | implies a choice of equivalences (akin to a logical OR) – i.e. something can be substituted by option 1 or option 2. Finally, concatenation of symbols (linking) is achieved by placing them next to one another [53].

The production rules for the algebra, defined in BNF are:

```

⟨unary operator⟩      ::= -
                      |
                      ~
                      |
                      carry()
                      |
                      [u:1]

⟨arithmetic operator⟩ ::= +
                      |
                      -

⟨boolean operator⟩   ::= &
                      |
                      |
                      ⊕

⟨binary operator⟩    ::= ⟨arithmetic operator⟩
                      |
                      ⟨boolean operator⟩
                      |
                      ::

⟨comparison operator⟩ ::= ==

⟨binary expression⟩  ::= ⟨expression⟩ ⟨binary operator⟩ ⟨expression⟩

⟨unary expression⟩   ::= ⟨unary operator⟩ ⟨expression⟩

⟨comparison expression⟩ ::= ⟨expression⟩ ⟨comparison operator⟩ number

⟨expression⟩         ::= variable
                      |
                      number
                      |
                      ⟨binary expression⟩
                      |
                      ⟨unary expression⟩
                      |
                      ⟨comparison expression⟩

```

5.7.2 Reduction Rules

Production rules whose left-hand sides consist of multiple symbols are a part of context-sensitive grammars. The introduction of these rules allows for specific cases to be reduced into equivalent representations which are more informative and concise. Some of these reductions are purely syntactical; one arrangement of symbols can be simplified into another. However, other reduction rules are contingent on the underlying values of specific symbols, or equivalence between the values of symbols. In these situations, the conditions under which the reduction rule can be applied are described underneath the rule.

Catching Double Negation and Inversion

Double negation and inversion cancel out – applying the negation or inversion operator twice to an expression simply returns the initial expression

$\cdot \text{--} \langle \text{expression} \rangle ::= \langle \text{expression} \rangle$

$\cdot \sim \sim \langle \text{expression} \rangle ::= \langle \text{expression} \rangle$

Arithmetic Interpretation of Xor and Not

The Xor (bitwise check for exclusivity) and Not (bitwise inversion) operations have alternative interpretations in arithmetic. Xor can be interpreted as addition, meaning that the transformation $A \oplus B \rightarrow A + B$ should be applied. Performing a bitwise Not to invert the bits of a number is the first step in 2's complement negation, with the second step being to add 1. Therefore, when a bit inversion of an expression is used in arithmetic operations, it should be interpreted as the negative of the expression, minus one. This yields the transformation: $A + \sim B \rightarrow A + (-B - 1)$. The transformed value can then be simplified into $A - B - 1$.

$\cdot \langle \text{expression} \rangle \oplus \langle \text{expression} \rangle ::= \langle \text{expression} \rangle \langle \text{expression} \rangle$

$\cdot \langle \text{expression} \rangle + (\sim \langle \text{expression} \rangle) ::= \langle \text{expression} \rangle + (-\langle \text{expression} \rangle - 1)$

Merging Bit Ranges on Append

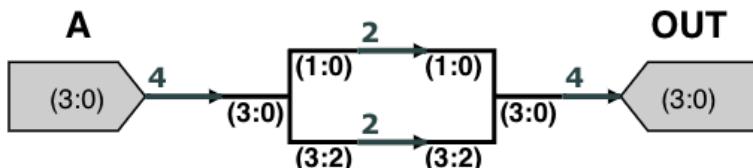


Figure 5.6: Splitting and Merging

Issie supports splitting multi-bit inputs into separate buses, as well as merging buses together in resultant wider bus. This is achieved through the *MergeWires* and *SplitWire* components. When an expression is split, the unary Bit Range operator is applied to the expression – this operator itself features two parameters: the upper and lower bound of the range. When expressions are joined (appended) together, the append operator performs the join. Figure 5.6 shows a circuit which splits a 4-bit input, and then immediately joins the two halves back together. Under naive algebraic simulation the output would yield the following expression:

$$A[3 : 2] :: A[1 : 0] \quad (5.3)$$

However, this is simply equivalent to the initial input expression A . This pattern, where unary expressions consisting of adjacent bit ranges applied to the same underlying expression are appended, is recognised and reduced appropriately. This rule can be defined in general as:

$\langle \text{expression} \rangle [\text{u1}:11] :: (\langle \text{expression} \rangle [\text{u2}:12]) ::= \langle \text{expression} \rangle [\text{u1}:12]$

'if both instances of $\langle \text{expression} \rangle$ are equal and if $11 = \text{u2} + 1$

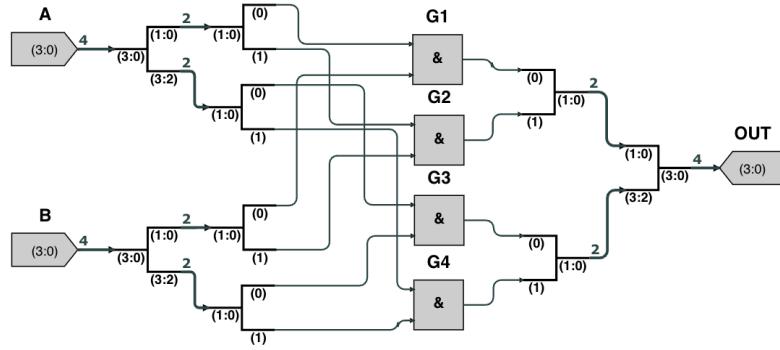


Figure 5.7: Circuit for Bitwise And

An example of how well the rule generalises is also provided below, with the following case being caught and reduced as shown. For context, the variables A and B represent inputs which are each 8 bits wide.

$$(A + B)[7 : 6] :: (A + B)[5 : 4] :: (A + B)[3 : 1] \rightarrow (A + B)[7 : 1] \quad (5.4)$$

Recognising Bitwise Operations

While Issie allows multi-bit inputs, as of the time of writing it does not feature any library components for performing bitwise Boolean operations. Therefore, users must manually split both multi-bit buses into their constituent bits using *SplitWire* components, perform the Boolean operation on each pair of bits, and assemble the results in order again using *MergeWires* components. Figure 5.7 shows the schematic that must be built to calculate the bitwise And between two 4-bit inputs A and B . The left-hand side of expression 5.5 is what would initially be produced under simulation: the constituent bits of each input being operated on and the results being appended together. The right-hand side is the result of applying the reduction rule: correct recognition of the bitwise operation on expressions A and B .

$$(A[3]\&B[3]) :: (A[2]\&B[2]) :: (A[1]\&B[1]) :: (A[0]\&B[0]) \rightarrow A\&B \quad (5.5)$$

In order for this simplification to take place, successive nested binary append operators must be flattened into a list for analysis to take place. This step is unnecessary in the F# implementation as appended expressions are already stored as a list. In the current formal system, a list of appended expressions can be defined. There is no need for an empty list terminal, as it is impossible to append nothing.

$$\begin{aligned} \langle \text{expression list} \rangle &::= \langle \text{expression} \rangle \\ &\mid \langle \text{expression} \rangle :: \langle \text{expression list} \rangle \end{aligned}$$

Using the above definition of an expression list, the following binary expression: $\langle \text{expression} \rangle :: \langle \text{expression} \rangle$, where either constituent expression may be of the same form (i.e. nested appends), can be expanded into a flat ordered list of expressions that are to be appended to one another. This list can be analysed to detect bitwise operations on expressions. The reduction rule for this case is described below.

$$((\langle \text{expression} \rangle [u_1:11]) \langle \text{boolean operator} \rangle (\langle \text{expression} \rangle [u_1:11])) :: ((\langle \text{expression} \rangle [u_2:12]) \langle \text{boolean operator} \rangle (\langle \text{expression} \rangle [u_2:12])) :: ((\langle \text{expression} \rangle [u_3:13]) \langle \text{boolean operator} \rangle (\langle \text{expression} \rangle [u_3:13])) ::= \langle \text{expression} \rangle \langle \text{boolean operator} \rangle \langle \text{expression} \rangle$$

if all instances of $\langle \text{expression} \rangle$ are equal and if u_i equals l_i

and if $u_{0..n-1}$ are successive, with the expression having width n

Boolean Simplification Rules

In Boolean algebra, there exist certain identities which allow for the simplification of Boolean expressions. These are described in Figure 5.8. These simplification rules have been implemented in the algebraic system as reduction rules; the letters A, B , and C represent any valid algebraic expression, while numbers 0 and 1 represent LOW and HIGH values rather than the actual values of 0 and 1. This distinction is important as numerical values in Issie can have widths greater than one. For example, Rules 3 and 4 for 8-bit algebraic variables would be: $A \& 0x00 = 0x00$ and $A \& 0xFF = 0xFF$ respectively. Rule 9 (double inversion) has already been described. It should be noted that the symbol "+" in Figure 5.8 means OR rather than addition, and the dot means AND. For rules involving commutative operations, the alternate case is also covered in the implementation. For example, $A \& 0$ and $0A$ will both evaluate to 0 under Rule 3.

1. $A + 0 = A$	7. $A \cdot A = A$
2. $A + 1 = 1$	8. $A \cdot \bar{A} = 0$
3. $A \cdot 0 = 0$	9. $\bar{\bar{A}} = A$
4. $A \cdot 1 = A$	10. $A + AB = A$
5. $A + A = A$	11. $A + \bar{A}B = A + B$
6. $A + \bar{A} = 1$	12. $(A + B)(A + C) = A + BC$

Figure 5.8: Boolean Algebra Identities [54]

In addition to the above rules, expressions which involve performing XOR with either LOW or HIGH values are also reduced. The cases for reduction are shown below; A represents any valid algebraic expression.

$$A \oplus 0 = A$$

$$A \oplus 1 = \sim A$$

Arithmetic Expression Simplification

While arithmetic expressions are not strictly defined in the algebra, with the more generic binary expressions being used, a possible definition of an arithmetic expression is shown below. This is the same definition as the one used for a binary expression, with the exception of a tighter set of operators being used.

$\langle \text{arithmetic expression} \rangle ::= \langle \text{expression} \rangle \langle \text{arithmetic operator} \rangle \langle \text{expression} \rangle$

As arithmetic expressions are themselves a type of expression, there is the possibility that other arithmetic expressions may be nested within an arithmetic expression. An example of this is shown on the left-hand side of expression 5.6, where numerous arithmetic equations have been nested. The issue with this nesting is that the arithmetic is not in its simplest form, which is shown on the right-hand side. B and $-B$ should cancel out, and the two numbers in the expression should be combined into one.

$$((A + B) - (((B + 1) + C) - 2)) \rightarrow ((A - C) + 1) \quad (5.6)$$

This process of arithmetic simplification has multiple steps. First, the structure of nested arithmetic expressions is recursively flattened into a list of arithmetic terms. These arithmetic terms in this flattened list are of the form $\langle \text{expression} \rangle$ if they are positive and $\langle -\text{expression} \rangle$ if they are negative. Algorithm 1 explains how nested arithmetic can be flattened. The flattening algorithm is recursively called on the left and right operands of arithmetic expressions to produce a list.

If the operator is subtraction, the values of the right list are negated element-wise. The two resultant lists are joined. Additionally, if a bit-inversion ($\sim A$) is recognised, it is converted into the arithmetically equivalent expression $-1 - A$; this expression is then flattened further in case A itself is an arithmetic expression. If the expression matches none of these cases, then it is at its flattest, and is therefore returned in a list by itself.

Applying the described flattening process to 5.6, results in the list of expressions on the left-hand side of expression 5.7. Matching pairs of positive and negative expressions and combining numerical values yields the list on the right-hand side. The expressions in this simplified list can then be re-assembled into binary arithmetic expressions.

$$[A, B, -B, -1, -C, +2] \rightarrow [A, -C, 1] \quad (5.7)$$

Algorithm 1 Algorithm for Flattening Nested Arithmetic

```

procedure FLATTEN(expression)
    if expression of form (expression1 + expression2) then
        left ← FLATTEN(expression1)
        right ← FLATTEN(expression2)
        return APPEND(left, right)
    else if expression of form (expression1 − expression2) then
        left ← FLATTEN(expression1)
        rtemp ← FLATTEN(expression2)
        right ← NEGATELIST(rtemp)
        return APPEND(left, right)
    else if expression of form  $\sim innerexp$  then
        return
        return FLATTEN( $-1 - innerexp$ )
    else
        return [expression]
    end if
end procedure

procedure NEGATELIST(listOfExp)
    for expression in listOfExp do
        if expression of form ( $-innerexp$ ) then
            listOfExp[i] ← innerexp
        else
            listOfExp[i] ← expression
        end if
    end for
    return listOfExp
end procedure

```

Full Adder Detection

In complex Issie designs, which often involve working with multi-bit buses of varying widths, most arithmetic operations in logic will be implemented using the *N-bits Adder* component. This component has three input ports: A, B, and Cin; and two output ports: SUM and COUT. The SUM port outputs the sum of the three inputs, while the COUT port outputs the carry-out of the addition. This arithmetic can be described by the Addition operator and the CarryOf operator. However, users can also create their own full adder component; this is a 1-bit adder. As this is implemented purely with logic gates, the addition and carry operators cannot be generated during the production phase. They must instead be inferred in the reduction phase. The schematic for a full adder is shown in Figure 5.9b; full adders are built by combining two half adders (Figure 5.9a). The sum of a full adder is simply $A \oplus B \oplus Cin$; XOR is therefore reduced to addition. Inferring the Carry is slightly more challenging. The carry-out from a half adder is simply $A \& B$ – this is too general of an expression to reduce to *carry(A+B)*. Doing so would result in multiple false-positives

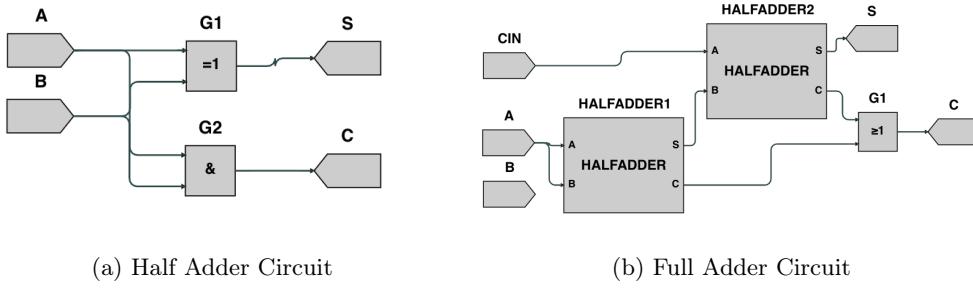


Figure 5.9: Adder Circuits

for carry detection. However, the carry-out from a full adder is slightly more complex, therefore a reduction rule for it can be written. Two reduction rules are required for recognising full adders, one for the sum and one for the carry:

$$A \oplus B \rightarrow A + B \quad (5.8)$$

$$(C \& (A + B)) | (A \& B) \rightarrow \text{carry}(A + B + C) \quad (5.9)$$

5.7.3 Implementing Algebra

In order to enable algebraic simulation, the Fast Simulation code was extended by the project to support algebraic inputs and reduction. Prior to the addition of algebra, the primary data type used to represent values in the Fast Simulation was `FastData`, a record type containing a value and its width. `FastData` was wrapped in the type `FData`; the code in the Fast Simulation used the `FData` type in the implementation. This was done to make the logic easier to extend, as if the logic ever had to handle two types of data `FData` could simply be transformed in to a Discriminated Union of the usable types. This is an example of *Extensibility*, one of Issie's core principles, in action. The approach of turning `FData` into a DU type was adopted by the project – the change in the type definition can be seen in Listing 5.11. `FData`, the data type used in all simulation logic, can either be numeric data (as it was before), or an algebraic expression. Following this change in definition, an extensive refactoring process had to take place to fix over 100 compiler errors caused by functions expecting to be given data resembling `FastData` as input, but actually given a discriminated union. Only after this exercise was completed, could the true implementation of algebraic simulation begin.

Listing 5.11: Change in definition of `FData`

```
// Old Definition
type FData = FastData

// New Definition
type FData = | Data of FastData | Alg of FastAlgExp
```

When the Fast Simulation is built, all components in the schematic are placed in a specific order which allows them to be reduced. This reduction process involves propagating values from the input ports of a component to its output ports, which are propagated to the input ports of the next component in line. This is repeated until the end of the list is reached. The function responsible for propagating these values is `fastReduce`. It consists of a large Pattern Match expression, with defined behaviour for how each component transforms its input values into output values. Previously, the values at the input ports could only be `FastData`, however after the change in definition they could also be algebraic expressions. Therefore, the defined behaviour for every component specified in `fastReduce` had to be extended to accommodate algebra.

Algebraic simulation consists of two stages. The first is the **production** stage, where a complex algebraic expression is produced by the `fastReduce` function. Each algebraic input into the overall logic is represented as a single variable (`SingleTerm`). As mentioned previously, `fastReduce`

defines a mapping from the expressions at the input ports of a component to expressions/data at its output ports. These output expressions are then propagated to the input ports of the next connected component. Figure 5.10 is an annotated schematic which shows how algebraic expressions propagate through components, growing in complexity as they pass through each component. At the completion of the production stage, every output of the simulation has an algebraic expression associated with it. While behaviour has been defined for all combinational components, algebra is not supported at select (SEL) ports of multiplexers, demultiplexers, and decoders. This is by design; as the algebraic expressions resulting from algebra at a SEL port would be long, unwieldy, and would not provide much insight into what the component actually does. Furthermore, inputs into SEL ports are often control signals for circuit behaviour, and therefore different cases ought to be separated out in the truth table. In these cases, an `AlgebraNotImplemented` exception, which contains a `SimulationError` explaining why algebraic simulation cannot take place is raised. Once caught, this exception is handled by showing the simulation error to the user. When the user designates an input as algebraic in the popup window, one case with the given input set as algebraic is simulated. If that simulation raises an exception, the user is prevented from applying that set of algebraic inputs. Table 5.9 summarises the mappings for algebra defined `fastReduce` which produce expressions at output ports.

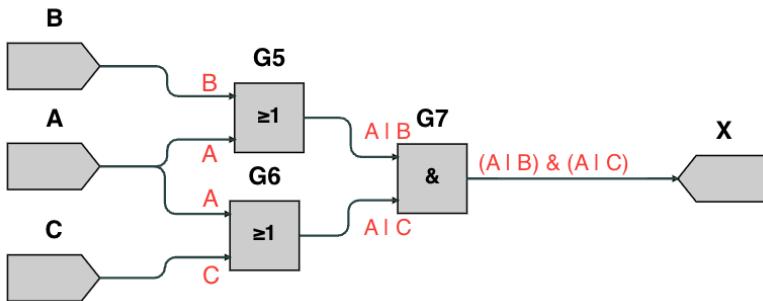


Figure 5.10: Annotated Schematic showing propagation of Algebra

The second stage is the **reduction** stage. In this stage, the expressions associated with the simulation outputs are recursively reduced using the reduction rules defined in Section 5.7.2. This is achieved by a recursive pattern match on the expression to be simplified – this pattern match is implemented in the function `evalExp`. In the example presented in Figure 5.10, the result of the production stage at the one output node of the simulation was $(A|B)\&(A|C)$. The reduction stage matches this expression against Boolean Simplification Rule 9 in Figure 5.8, and reduces it to $A|(B\&C)$. The specific case in the pattern match in `evalExp` which applies this reduction rule is shown in Listing 5.12.

Listing 5.12: Case in pattern match which implements the reduction rule for Boolean Simplification Rule 9

```
// (A OR B) AND (A OR C) = A OR (B AND C)
| BinaryExp(e1, BitOrOp, e2), BinaryExp(e3, BitOrOp, e4) ->
  if e1 = e3 then
    BinaryExp(e1, BitOrOp, BinaryExp(e2, BitAndOp, e4))
  else if e1 = e4 then
    BinaryExp(e1, BitOrOp, BinaryExp(e2, BitAndOp, e3))
  else if e2 = e3 then
    BinaryExp(e2, BitOrOp, BinaryExp(e1, BitAndOp, e4))
  else if e2 = e4 then
    BinaryExp(e2, BitOrOp, BinaryExp(e1, BitAndOp, e3))
  else
    BinaryExp(left, BitAndOp, right)
```

Component	Case	Defined Behaviour
Not	-	Apply the unary Not operator to the algebraic expression.
BusSelection	-	Apply the unary BitRange operator to the expression based on width and LSB supplied.
BusCompare	-	Apply the comparison operator to the expression, comparing it to the supplied value.
And, Or, Xor	-	Apply the relevant binary operator to both expressions
Nand, Nor, Xnor	-	Apply the binary operator for the uninverted versions of the gates, and then apply the unary Not operator to the result.
Mux2, Mux4 Mux8	Data inputs are algebra or numeric, Select input is numeric	If the select input is 0, propagate the expression/number connected to port 0. If select input is 1, propagate the expression/number connected to port 1 and so on...
	Select input is algebra	Algebra not allowed at select port of multiplexer. Raise AlgebraNotImplemented exception.
Demux2, Demux4, Demux8	Data input is algebra or numeric, Select input is numeric	Propagate the expression/value to the correct output corresponding to the Select value.
	Select input is algebra	Algebra not allowed at select port of demultiplexer. Raise AlgebraNotImplemented exception.
Decode4	Data input is algebra or numeric, Select input is numeric	Propagate the expression/value to the correct output corresponding to the Select value.
	Select input is algebra	Algebra not allowed at select port of decoder. Raise AlgebraNotImplemented exception.
NbitsAdder	-	Apply the binary Add operator to two inputs, and then apply a second Add operator to the first expression and the third input.
NbitsXor	One input is algebra, the other is -1.	This is bit inversion, apply the unary Not operator to the input.
	-	Xor on multi-bit inputs is usually interpreted as addition, so apply the Add operator to the two operands.
SplitWire	Input is an algebraic expression with a BitRange operator	Change the values in the bit range operator according to which bits are going to output ports 0 and 1.
	Input is an algebraic expression with a Not operator	Apply the BitRange operator to the expression inside the Not operator, as bit ranges bind closer.
	-	Apply the unary BitRange operator to the expression, with different ranges for output ports 0 and 1.
MergeWires	Both inputs are append expressions (a list of appended expressions)	Join the two lists in the correct order, and then check if any BitRanges can be joined.
	One of the inputs is an append expression	Add the new element to the head or tail of the list depending on the order, and then check if any BitRanges can be merged.
	-	Put the two inputs in a list together in the correct order, and check if any BitRanges can be merged.
Any other components	-	These are sequential components which should never receive, nor accept algebra, raise an AlgebraNotImplemented exception.

Table 5.9: New additions to the defined behaviour for each component in `fastReduce`

Implementation of Arithmetic Simplification

Arithmetic simplification is one of the most important reduction rules in the algebraic system. Not only does it remove redundant expressions and numeric values from the output expression, but it also converts bit inversions into subtractions. The first step of arithmetic simplification is the flattening of nested arithmetic into a list of expressions. The algorithm for this was described in

Algorithm 1, and is implemented in the F# function `flattenNestedArithmetic`. The next step is to remove redundancies in the list; this includes cancelling out positive and negative instances of expressions and collecting numeric terms. This is achieved by folding the list of expressions using two states. The first state is an integer; this is used to collect numeric values. If a positive number is encountered in the list, the number is added to the running state. If a negative number is encountered, its value is subtracted from the running state. The second state is an expression counter, implemented as a `Map<FastAlgExp, int>`. When an expression is encountered in the list, its value in the Map is incremented. If the negative version of the expression is encountered, its value in the Map is decremented. At the conclusion of the `List.fold`, the first state represents the final numeric value, while the second state contains the coefficient for each expression in the list. This information is then reassembled into a simplified arithmetic expression. Figure 5.11 summarises the arithmetic simplification process using an example, showing the value of specific data structures at different points in the process.

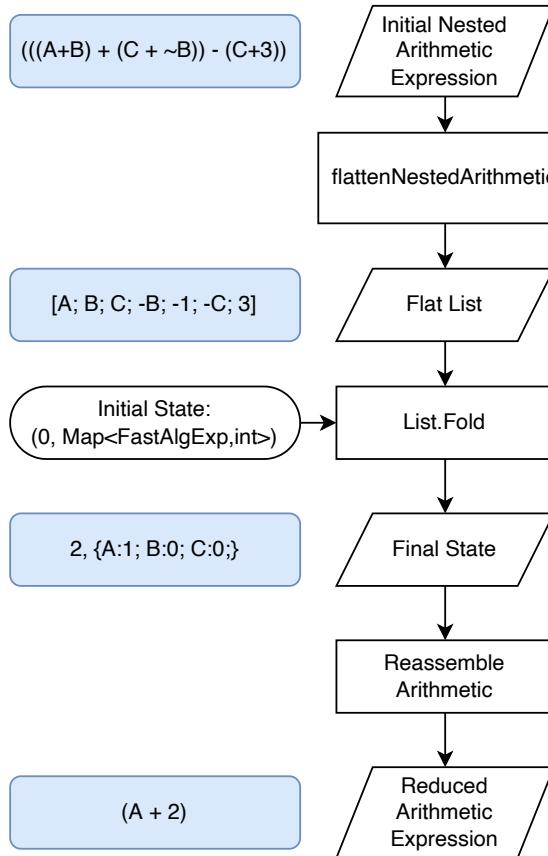


Figure 5.11: Process for Arithmetic Simplification with Example

5.8 Sorting Truth Tables

In Issie, the order in which rows appear in the truth table can be set by the user by clicking the up/down arrows in the heading of each column of the table. The initial order of the rows in the truth table is seemingly arbitrary; this is because truth tables are generated as F# Maps, and the order of the keys in the Map is determined by F#’s generic comparison [48]. As Maps cannot retain a custom set order, a different collection type had to be used for sorting the truth table. Therefore, the Map is converted row-wise into a list of `TruthTableRow`s, where each individual `TruthTableRow` is formed by appending each input row to its corresponding output row. This list representation is the last stage in the truth table caching strategy, which will be discussed further in Section 5.9.

When the user clicks a specific arrow in the truth table to sort it, two messages are sent to the `Update` function. The first message, `SetTTSortType`, contains two pieces of information: which

IO should the table be sorted by, and whether this is in ascending or descending order. This information is updated in the Model (under field `TTSortType`) . The second message marks the truth table as being out of date, with the reason that it must be sorted. The reason for why the setting of the sort type is separated from actual sorting of the truth table is explained in detail in 5.9. The actual sorting process begins by looking up the current Sort Type in the model – if there is some then the truth table is sorted according to it. F# provides library functions for sorting lists in $O(n \log(n))$ time using QuickSort; one such function is `List.sortWith`, which allows the developer to define a custom comparison function for two elements in the list. A custom comparison function for comparing two `TruthTableRows` is required, as the generic comparison offered by F# does not compare rows correctly. When the truth table is being sorted by some IO X , the position of a given row in the sorted truth table is dependent on the value of X in that row. Therefore, it was determined that the process of comparing two rows involved first extracting the appropriate `CellData` values for the IO, and comparing those.

5.8.1 Comparison Function for CellData

The following comparison rules were determined for data in truth table cells:

1. Don't Care (DC) Terms are larger than Algebraic Expressions, which are larger than numeric values.
2. Algebraic terms are compared based on their alphabetical order, so "A" < "B".
3. Numerical terms are compared based on their values, as expected for numbers: $0 < 1 < 2$ etc.

Functions for comparing two values in F# have the following signature:

`'T -> 'T -> int`

This means that the comparison function takes the two objects to compare as arguments, and returns an integer which signifies which argument was larger than the other. A positive return values indicates that the first argument is greater than the second, a negative return value suggests that the second argument is greater than the fist, and a return value of 0 indicates that the values are equal. The three comparison rules were implemented as a comparison function with the aforementioned signature; this can be seen in Listing 5.13. The alphabetical and numerical comparisons are handled with F# 's built-in `compare` function.

Listing 5.13: Function to compare two `CellData` values

```
let compareCellData (cd1: CellData) (cd2: CellData) =
    match cd1, cd2 with
    | DC, DC -> 0
    | DC, _ -> 1
    | _, DC -> -1
    | Algebra _, Bits _ -> 1
    | Bits _, Algebra _ -> -1
    | Algebra a1, Algebra a2 ->
        compare a1 a2
    | Bits wd1, Bits wd2 ->
        (convertWireDataToInt wd1, convertWireDataToInt wd2)
        ||> compare
```

5.9 Truth Table Caching and Order of Operations

So far, five user operations that change the truth table data structure have been discussed. These are summarised in Table 5.10. From inspecting this table, it can be seen that only changing the input constraints or changing algebraic inputs requires the truth table to be re-generated. The rest of the operations simply transform the existing truth table. For this reason, the whole process of serving the user a truth table is split into various parts, each of which execute in a specific order, only do as much work as necessary, and cache their results. This whole process is described in

Figure 5.12. There is a strict order of operations; first the truth table is regenerated, then it is filtered, and then it is sorted. Column hiding occurs after sorting, but this does not change the truth table data structure itself, and is therefore discussed in Section 5.11.2. If the truth table is to be reduced with Don’t Cares, then the filtering stage filters the DCMap instead of the TableMap, and this propagates through the subsequent stages. The result of each operation is stored in the TruthTable data structure, which is itself stored in the model. Once a given operation has completed and stored its result, it calls the next step using a *Command*. In the MVU architecture, Commands are used to send a message to the next call of the update function. In the context of the system in Figure 5.12, using commands means that the next operation in the order is executed during the next call of the Update function. This ensures that the next operation uses the newly stored result from the current operation.

Operation	Summary
Changing Input Constraints	The truth table is re-generated, taking the new input constraints into account.
Changing Output Constraints	The existing truth table is filtered, with only rows which fulfil the constraints allowed to remain.
Changing Algebraic Inputs	The truth table is re-generated, taking the new algebraic and numeric inputs into account.
Don’t Care Reduction	The existing truth table is reduced down into an alternate form with redundancies removed and replaced with DC Terms.
Sorting	The existing truth table rows are reordered to reflect the user’s chosen sorting method.

Table 5.10: Summary of operations which change the TruthTable data structure

The reason for this system, as opposed to one where all of the operations are performed in one go, is best explained through an example. Suppose the truth table has two output constraints $X = 0$ and $Y = 0$ applied to it, and the user subsequently deletes the latter. Rather than re-generate the truth table from scratch and then apply the new set of output constraints, Issie simply uses the cached version of the full table (stored in TableMap), and applies the new output constraint set to that. This approach reduces the time-penalty for the given operation, as unnecessary steps are cut out.

Another key feature of the system is that the parameters for each operation are stored independently in the model. When an operation parameter, such as the algebraic input set or the constraint set changes, the update function first updates the stored parameter value in the model, then sends a command carrying the message to execute the appropriate operation. Storing the parameters has two uses; the first is that it allows for them to be displayed and tracked. For example, constraints that are currently being applied are displayed to the user using tags, and are used to validate new ones. However, the more interesting effect of this concept is that users do not lose their preferences no matter what other operations they do on the truth table. This too is best explained through an example. Suppose the truth table from before still has the output constraint $X = 0$ applied to it, but the user then chooses to add a new input constraint $A = 0$. When the user clicks the *Add* button in the popup, the *AddInputConstraint* message, which contains the new constraint, is sent – this is then processed by the Update function which stores it in the Model. Following this, a command containing the message *RegenerateTruthTable* is sent. In the next call of the Update function, the truth table is regenerated, and once this is completed the rest of the stages execute in the correct order. As the output constraints are stored independently in the model, the system remembers to apply $X = 0$ to the new TableMap. In this vein, any pre-existing sorting method, column order, or hidden columns are also preserved after the re-generation. This means that **only** the operation instructed by the user occurs, meaning that the user is not left confused by unintended side effects of their interactions with the truth table.

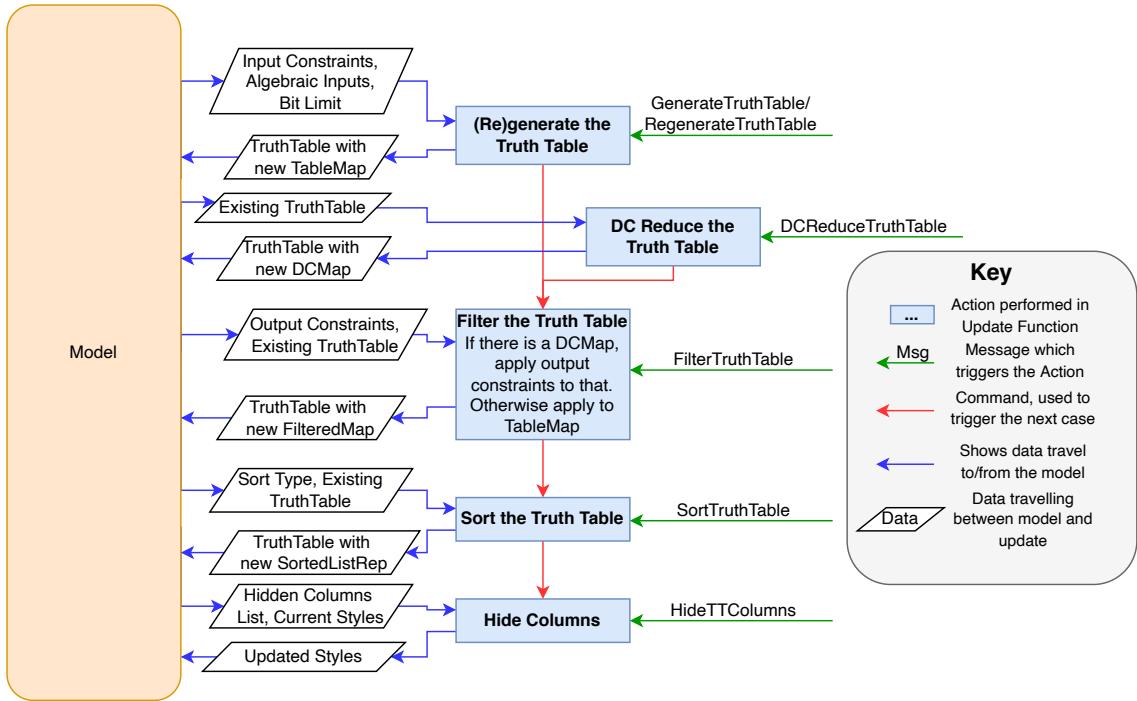


Figure 5.12: How operations update the Truth Table

5.10 Rendering the Truth Table

Displaying a truth table involves taking the `TruthTable` data structure and transforming it into a `ReactElement` which can be rendered by React. Two methods were considered for this task; Fulma tables and CSS Grids. The former was used initially, but due to limitations that will be explained in this section, the latter was eventually favoured.

5.10.1 Method 1: Using Fulma Tables

Fulma, as described in 2.4.3, is an F# library which provides ready-to-use front-end components for Fable React applications. Component libraries like Fulma have many advantages over plain HTML components that are manually styled using CSS. All Fulma components are pre-styled with specific themes, therefore consistently using Fulma components across an application provides consistency in styling. Additionally, components have style modifiers which can be easily selected by the developer without manually implementing them in CSS. For example, a Fulma table can be made zebra-striped using the `Table.IsStriped` modifier [38]. Certain behaviour, such as automatic re-sizing and automatic wrapping of content is also implemented in Fulma tables. Due to these advantages, and given that Fulma components are used extensively throughout Issie, Fulma tables were initially chosen for rendering the truth table. Listing 2.1 in Section 2.4.3 shows the syntax for generating a table with Fulma. A table heading tag (`thead`) contains a table row tag (`tr`), which contains the numerous heading cells (`th`) in the column-order they will be displayed. Similarly, a table body `tbody` tag wraps multiple table row tags, which wrap multiple table data `td` tags. This tag hierarchy is akin to a nested list representation of a table; therefore it was easy to convert a list representation of the truth table into a Fulma table. Each `TruthTableCell` was mapped to a `td` component, and the row containing those cells as mapped to a `tr` component.

Issue with using Fulma Tables

Table components (both Fulma and HTML) have one major limitation: they are not interactive. The positioning of content in the table is solely dependent on the order during the definition of the table. Once the table is defined, there is no way to change the order in which rows or columns appear in the table at render time. Therefore, all user operations on the table must update the underlying truth table data structure. The previous Section (5.9) showed how truth

tables operations are executed and cached in the delivered version of Issie; however when tables were rendered using Fulma, hiding/un-hiding columns and changing the order of columns used to also change the `TruthTable` data structure and cache the result. For example, when a user moved a column, the order of cells in every `TruthTableRow` was changed to reflect the new order. This process involved first copying the sorted list representation into an array, then mutating the array to match the IO order defined by the user. The effect of the quadratic time complexity, coupled with the repeated copying of data and slow element-wise access of lists, was that re-ordering operations on large truth tables (1024 rows, more than 5 columns) took an average of 608.4ms. According to Robert Miller [41], such a delay does not appear instantaneous to the user and can therefore make the application feel sluggish rather than responsive. The performance of this method was deemed unacceptable, as requirement **E1.8** clearly states that graphical manipulation operations must appear instantaneous. Therefore, an alternative approach was considered.

Method	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Method 1	607	605	611	610	609	608.4
Method 2	21	25	22	23	27	23.6

Table 5.11: Time taken to move a column in a Truth Table under different methods (ms)

5.10.2 Method 2: Using CSS Grids

The CSS Grid layout [55] is a system which allows the developer to define the positioning of specific elements in two dimensions. Unlike tables, where the positioning of elements is defined inline in the HTML using tags, all display properties of grid elements such as positioning, size, and behaviour are defined within a CSS style. Each grid element is therefore defined within a `div` with the appropriate style applied to it. The advantage of this is that changing element positioning does not require changing the underlying data structure and rebuilding the grid – simply updating the styles will change how the grid is rendered. This ultimately means that grids can be made interactive while tables can not. CSS grids were chosen due to this interactivity property, as it would greatly speed up column hiding and re-ordering as manipulation of the truth table data structure would no longer be required. This hypothesis was proven to be true; Table 5.11 compares the time taken to column move a column under both methods, and shows that manipulating CSS styles is significantly faster than changing the underlying data structure. One of the reasons why Fulma tables were initially chosen was that they would maintain UI consistency with the numerous other Fulma components used in the application. To keep up this UI consistency, the implemented grid has been styled to mimic the look and feel of Fulma tables.

When the truth table is initially generated, all the IOs in the truth table are arranged in the default simulation order. All of the IOs in the truth table are extracted in this initial order using the `TruthTable.IOrder` member function. Listing 5.14 shows how a mapping between each of these IOs and a list of CSS Properties is created. The `ttGridColumnProps` returns the appropriate CSS Properties for each IO; most of these properties relate to formatting and appearance, however the `GridColumnStart` and `GridColumnEnd` properties are dependent on the index of the IO. These properties determine the position of an element in the truth table. The reason why the Start position has 1 added to it is that CSS Grids begin indexing from 1, while F# lists begin indexing from 0. This Map is then stored in the Model as `TTGridStyles` for use during the truth table viewing process, which turns the truth table data structure into a React Element. The last two properties: `OverflowX` and `OverflowWrap` are responsible for ensuring that long algebraic expressions wrap to the next line in the truth table instead of overflowing the cell that contains them.

Listing 5.14: Generating CSS Properties for each IO

```
let ttGridColumnProps index = [
    Border "1px solid gray"
    Padding "7px"
    FontSize "18px"
    TextAlign TextAlignOptions.Left
    GridColumnStart <| string (index+1)
    GridColumnEnd <| string (index+2)
```

```

    OverflowX OverflowOptions.Auto
    OverflowWrap "break-word"
]

let colStyles =
  tt.IOOrder
  |> List.mapi (fun i io -> (io,ttGridColumnProps i))
  |> Map.ofList

```

When the truth table is viewed, each `TruthTableCell` is transformed into a `div` instead of a `td`. The style applied to the `div` is obtained by looking up the `CellIO` in the `TTGridStyles` Map which was calculated and stored during initial generation. All of these individual grid elements are wrapped in a container `div`, which is styled property `Display DisplayOptions.Grid` to indicate that it is the element that contains the grid. Any subsequent manipulations to the position or visibility of specific IOs in the table can be achieved by updating the `TTGridStyles` Map rather than the truth table itself. The displayed truth table itself is zebra-striped; the background colour of rows alternates between white and off-white. This is done to improve the clarity and readability of the truth table. A study of 244 people concluded that task performance when using zebra-striped tables is better, or at least no worse than alternative table styles [56].

5.11 Column-based Operations

Operations on the truth table in Issie can be divided into three classes: regenerative, row-based and column-based. Row-based operations manipulate the truth table data structure because they filter or re-order rows based on the specific values stored in cells in each row. As truth tables are stored row-wise in both Map and List representations, manipulating rows is relatively inexpensive. Column-based operations on the other hand do not require reading values from the truth table itself, and are more expensive than row-based operations. For this reason, they are carried out by manipulating the CSS applied to grid elements. As discussed in the previous section, the styles for each column in the truth table are stored in the `TTGridStyles` Map, which maps an IO to a list of CSS properties. Different column-based operations update the stored property list in the map accordingly.

5.11.1 Moving Columns

Truth table columns can be moved using the left and right arrows on either side of the IO Label in the heading of each column. The current order of the columns is stored in the model under the field `TTIOOrder`; this is an array of IOs in the order they currently appear in the truth table. When the user clicks an arrow to move a column, the message `MoveColumn`, containing the IO and the direction it moves (right or left), is sent. When this message is processed, `TTIOOrder` is mutated to reflect the new order. The styles map `TTGridStyles` is also updated to reflect the new position of the column in question. During the next view function call, the updated styles will be applied to the grid elements, resulting in the order changing on the user's screen at render-time.

5.11.2 Hiding Columns

Using the toggles in the *Hide/Un-hide Columns* menu section, users can change the visibility of output columns in the truth table. The list of IOs whose columns are hidden is stored in the model under the field `TTHiddenColumns`. Toggling column visibility to *Hidden* adds the respective IO to the list, and toggling it back to *Visible* removes it from the list. Once this field is updated, a command containing the message `HideTTColumns` is issued, meaning that the styles are updated in the next call of the update function. The style corresponding to a hidden IO is updated with the CSS properties shown in Listing 5.15. Each hidden column is moved to the end of the grid (hence the requirement for the `gridWidth` argument), given a width of 0, and has its `visibility` property set to `hidden`, obscuring it from view. The styles of the remaining visible columns are also updated so that there are no visible gaps in the truth table. A key detail of this system is that the IO itself is not removed from the stored `TTIOOrder` – only its style is changed. This means

that when the IO is eventually un-hidden, it will re-appear in the same place as it was before, maintaining continuity.

Listing 5.15: Generating CSS Properties for a Hidden IO

```
let ttGridHiddenColumnProps gridWidth= [
    GridColumnStart (string <| gridWidth + 1)
    GridColumnEnd (string <| gridWidth + 2)
    Width 0
    OverflowX OverflowOptions.Hidden
    Visibility "hidden"
]
```

Chapter 6

Testing and Results

The delivered improved version of Issie was tested from three perspectives. These were **correctness**, **performance**, and **user experience**. Testing the correctness of the improvements made to Issie included checking that the implemented functionality worked as intended, as well as ensuring that the application was stable, not crashing regardless of user input. Quantitative performance testing was conducted by first measuring the average time taken for specific actions and processes, and then those measurements to Robert Miller's [41] classes for perceived responsiveness of an application. Finally, the quality of the user experience was evaluated by giving survey participants a series of tasks in Issie, measuring their performance, and collecting their feedback.

6.1 Testing Application Stability

An application is considered stable when the user is not subjected to undefined behaviour or inexplicable crashes while using the application. The stability of the improved version of Issie delivered by the project was verified by analysing cases in the codebase which could trigger a system crash, and systematically evaluating that they could not occur. There are two scenarios which can cause Issie to crash:

- Un-handled exceptions: exceptions can either be generated by the developer or can be generated by library functions. If these exceptions are not caught and handled appropriately, they cause the application to crash.
- When `failwithf` is called in the code: this function is called when an unexpected case that should never occur is matched in the code. An example of this would be if the `Update` function receives the instruction to reduce a truth table with Don't Cares when no truth table exists.

6.1.1 Exception Analysis

The code added to Issie was analysed to explore the scenarios in which exceptions could be raised. Exceptions are uncommon in F# – instead the use of Monad types such as `Option` and `Result` is preferred. When using Monad types, failure of an operation is an option and the programmer is made to account for the possibility of it while programming. In contrast, functions which return exceptions place the onus of verifying the arguments prior to the call on the developer. An example of this is the difference between the functions `Map.find` and `Map.tryFind`; both functions look up a key in a Map and return the corresponding value, but return the value in different ways. When a key exists, `Map.find` returns the value, but when it does not exist it raises an exception. If the developer fails to account for this, the program may crash unexpectedly. In contrast `Map.tryFind` wraps the returned value in the `Option` type, and returns `Some value` if the key exists in the Map, or `None`. Therefore, when using `Map.tryFind`, the developer must always actively handle the failure case at that point in the code, either providing alternate behaviour or manually making the choice to fail the application with `failwithf`. This significantly reduces the chance of application crashes due to an oversight made by the developer with regard to exception handling.

All library functions used in the code written during the project were analysed, and it was checked which of the ones used could possibly return any exceptions. For each of these functions a strategy was devised to check whether an exception could occur in practice:

Function	Check	Pass/Fail
<code>List.except</code>	Ensure that the items to exclude sequence can never be null.	Pass
<code>List.head</code>	Ensure that the supplied list is not empty either through a conditional statement or pattern match case on the list.	Pass
<code>List.updateAt</code>	Ensure that the index is valid (between 0 and length-1) either through a bounds check or from properties.	Pass
<code>Seq.allPairs</code>	Ensure that the both sequences can never be null.	Pass
<code>Seq.append</code>	Ensure that the both sequences can never be null.	Pass
<code>Seq.init</code>	Ensure that the count can never be negative.	Pass

Table 6.1: Library functions in project code which can throw exceptions

Exceptions can also be raised in the code by the developer if required, and caught using a `try-catch` block. This project only adds one such exception; the `AlgebraNotImplemented` exception which is raised in the Fast Simulation function `fastReduce` when algebra is passed to an unsupported port or component. An exception is used here because manually propagating a `SimulationError` up the call stack would be very impractical. The `AlgebraNotImplemented` exception contains a `SimulationError` data structure; whenever there is a scenario in which algebra is fed to the Fast Simulation, there is a `catch` block waiting to catch any exception and return the `SimulationError` contained within. Additionally, the XML documentation of functions in the Fast Simulator has been updated to reflect the new exception. Therefore, it can be said that the code added by this project to Issie is stable from an exception point-of-view.

6.1.2 Failure Analysis

To ensure that the application could not fail during use, every `failwithf` case in the project code was inspected, and a summary made of the scenarios in which the function would be called. Once the scenario was ascertained, one of two actions were taken. If the fail case was related to application state, it was ensured through thorough inspection of the code that such a state could never occur. If the fail case was related to the UI, testing involved attempting to create those circumstances in the application. In all cases, attempts to achieve scenarios that would call the application to fail and exit were unsuccessful.

In addition to exception and failure analysis, the completed application has been used for other purposes for long periods of time. The process of repeatedly testing the correctness of other features totalled over 3 hours, in which multiple tasks were carried out using the application. Additionally, during user experience testing, not a single user reported any crashes or undefined behaviour. This supports the notion that the code added to the project is robust and stable, and that all situations with erroneous user input have been handled.

6.2 Correctness Testing

Correctness of the implemented features was tested by testing the features on 6 different circuits, each of varying complexity. These circuits can be described as:

- A two-input multiplexer circuit implemented only using gates
- A circuit containing a Mux4 component with 2-bit inputs to each data line of the multiplexer
- A circuit which calculates the bitwise And of two 8-bit inputs
- A Full Adder circuit using a Half Adder custom component, all built exclusively from logic gates
- A circuit which either adds or subtracts two 16-bit values depending on the selected mode

- An 8-bit ALU, which is a custom component in the design of an 8-bit CPU
- An 8-bit CPU design

A summary of the tested features can be found in Table 6.2. All of the tested features worked as intended for all schematics, indicating that they have been implemented correctly and will therefore work reliably when distributed to real-world users.

Table 6.2: List of all features which were manually tested

Feature Checked	Pass/Fail
For all combinational circuits, a numeric truth table can be generated for the whole sheet.	Pass
For all valid selections of a schematic, a numeric truth table can be generated: <ol style="list-style-type: none"> 1. The selected canvas is corrected successfully, with newly generated input or output components. 2. Newly generated IOs are labelled based on which component ports they connect to. 3. Truth tables for selections can be generated even if errors or sequential components exist elsewhere in the schematic. 	Pass
If there are errors in the schematic, the <i>See Problems</i> button appears in place of the <i>Generate Truth Table</i> button, and clicking it conveys the error to the user.	Pass
Users are prevented from generating truth tables for schematics containing sequential logic, and this reason is conveyed to them.	Pass
For all circuits where the total width of all the inputs combined exceeds 10 bits, a truncated truth table of 1024 rows is displayed, along with a warning notification informing the user about the truncation.	Pass
The base of numbers in the truth table can be changed using the base selector.	Pass
Truth table related functionality can be shown and hidden by expanding and closing menu sections.	Pass
Input constraints can be applied to the truth table and successfully change the input state so that rows that were previously truncated are now generated and displayed.	Pass
Significantly restrictive input constraints will reduce the input space enough so the truth table will no longer be truncated.	Pass
Output constraints filter the existing truth table.	Pass
There is an intuitive interface for adding/removing constraints: <ol style="list-style-type: none"> 1. Clicking on the <i>Add</i> button opens the constraint editor popup. 2. Changing the chosen IO in the IO selection section updates the IO displayed in the editor section. 3. Real-time validation of constraints is performed as they are entered, with errors clearly conveyed to the user. 4. Constraints that are currently being applied are clearly displayed to the user with tags, and can be deleted easily by clicking the cross next to them. 5. All constraints can be cleared in one go with the <i>Clear All</i> button. 	Pass
Columns in the truth table can be hidden by using the column hider toggles.	Pass

Continued on next page

Table 6.2: List of all features which were manually tested (Continued)

Feature Checked	Pass/Fail
Rows in the truth table can be sorted based on a chosen IO, in ascending and descending order.	Pass
The sorting information is conveyed to the user successfully through the highlighting of the relevant sorting arrow.	Pass
The order of columns in the truth table can be changed.	Pass
Non-truncated truth tables can be reduced with Don't Cares, with the resultant table containing no redundant rows.	Pass
Users are prevented from DC reducing truncated truth tables.	Pass
Users can change some or all inputs to algebra in the truth table: <ol style="list-style-type: none"> 1. Clicking the algebra button always spawns the algebra selector popup, where inputs can be toggled between numeric and algebraic values. 2. If a certain input is not supported as being algebraic, the user is informed of the reason why. They are prevented from applying the incompatible algebraic inputs. 3. When algebraic inputs are in the truth table, the outputs are informative algebraic expressions which is a correct function of the inputs. 	Pass
Algebraic reduction rules are applied correctly, yielding correctly simplified algebraic expressions at the outputs	Pass
Algebraic expressions are printed correctly in the truth table.	Pass
When the truth table tab is open, width of the right section can be changed using the draggable dividerbar.	Pass
The truth table is responsive: <ol style="list-style-type: none"> 1. Changing the width of the right section changes the width of the truth table (therefore of the columns in the table). 2. When the content inside a truth table cell is too wide, the height of the row automatically adjusts so that the content can wrap to the next line. 3. Operations on the truth table appear instantaneous. 	Pass
The waveform simulation can successfully be run from the new Waveform Simulator sub-tab.	Pass

6.3 Quantitative Performance Testing

The three following schematics were used for measuring the performance of the application, with each having slightly different attributes. All tests were performed on a MacBook Air, featuring an M1 CPU with 8GB of RAM.

1. A circuit containing a Mux4 component with 2-bit inputs to each data line of the multiplexer. In total, this circuit has 5 2-bit inputs, meaning that it will generate exactly 1024 rows (no truncation required). This is a simple circuit with only one component, so should be quick to simulate. However, there are many redundancies in the truth table, so Don't Care reduction may take longer.
2. A circuit which either adds or subtracts two 16-bit values depending on the selected mode. This circuit features slightly more complex logic than the previous one, and also will result in the truth table being truncated. This will be referred to as the 'Add Or Sub' schematic.

- An 8-bit ALU, which is a very complex schematic featuring lots of library and custom components. This should test the generation algorithm to the maximum.

6.3.1 Generating Numeric Truth Tables

The time taken to generate a numeric truth table was measured for all three aforementioned schematics.

Sheet	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Mux4	52	58	51	49	51	52.2
Add Or Sub	75	72	69	81	78	75
ALU	697	702	700	697	702	699.6

Table 6.3: Time taken to generate a numeric truth table (ms)

6.3.2 Algebraic Truth Tables

The time taken to calculate an algebraic truth table for the 8-bit ALU was measured. The circuit had five inputs in total: A and B which were 8 bits, X and F which were 3 bits, and Cin which was 1 bit. A, B, and Cin were set as algebraic inputs for the experiment. Five trials were carried out, and the average time was calculated.

Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
45	42	44	47	45	44.6

Table 6.4: Time taken to generate an algebraic truth table for the ALU schematic (ms)

6.3.3 Don't Care Reduction

The time taken to reduce an existing truth table using Don't Cares was measured for two schematics. The Mux4 schematic was first tested, as it contains multiple redundancies and therefore requires repeated rounds of reduction. Secondly, the 'Add Or Sub' schematic was tested. As this schematic exceeds the bit limit, the input space was limited; inputs A and B were constrained to between 0 and 16. This schematic was chosen as there are no redundant rows in the truth table, as every input contributes to the output. This represents a scenario in which no DC rows are valid, and therefore the process will stop after one round of attempted reduction.

Sheet	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Mux4	2590	2586	2589	2616	2598	2595.8
Add Or Sub	265	263	260	259	263	262

Table 6.5: Time taken to reduce a numeric truth table using Don't Cares (ms)

6.3.4 Graphical Manipulation of Truth Tables

Using the truth table generated for the ALU schematic, the time taken to perform three graphical manipulations was measured. These manipulations were: hiding an output column, sorting the truth table, and moving a column. The ALU schematic was chosen as it will generate a truncated truth table, meaning that the maximum possible 1024 rows will be rendered. It also contains 8 IOs, which is a sizeable amount.

Operation	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Sorting Truth Table	35	33	36	41	39	36.8
Moving a Column	21	25	22	23	27	23.6
Hiding a Column	43	38	37	39	38	39

Table 6.6: Time taken to conduct different graphical manipulations on a truth table (ms)

6.4 User Experience Testing

The success and long-term viability of any user-facing application or platform is highly dependent on the user experience it provides. For this reason, feedback must be collected from users to evaluate whether the application achieves its purpose, and whether it is user-friendly. User feedback was collected to answer the following questions:

1. Do the added features fulfil the project aim, which is to add effective novel ways of visualising combinational logic in Issie, delivering an improved education and hardware design platform?
2. Are these features implemented in a user-friendly manner?

The participants of the user feedback survey consisted of a group of 12 engineering students across multiple departments at Imperial College, most of whom had some beginner-level experience with combinational logic design. Participants were chosen this way so that they would not be confused by the digital electronics concepts, and would therefore be able to fully appreciate and use the added functionality. The feedback collection process consisted of two stages; in the first stage participants were provided with the updated version of Issie, then tasked with investigating five 'mystery' Issie sheets and summarising the logical function implemented by each. They were asked to use the truth table functionality while doing so, but were not pointed to where it was located in the application. The second stage consisted of a questionnaire, in which participants were given the opportunity to evaluate their experience using Issie.

6.4.1 Stage 1: Mystery Sheets

Participants were provided with five schematics labelled *mystery1* to *mystery5*, and were asked to describe what each sheet did. While the instructions encouraged them to "use truth tables and other features in the Truth Tables tab", they were not told where this tab was in the app's top-level UI, nor were they instructed on how to use any of the features. This was done on purpose as Issie meant to be easy to use without explicit guidance. Through this the *intuitiveness* and *obviousness* of the UI for truth tables was tested. The mystery sheets increased in complexity, with the initial sheets being simple to introduce the user to the application, and later sheets being more complex to push the user to explore more and use many of the added features. Table 6.7 provides a summary of each mystery sheet, as well as the rationale behind using it in the survey. The purpose of this exercise was to ascertain whether the added features did indeed help users gain a better understanding of combinational logic designs. Sheets Mystery 1 to Mystery 4 had a perfect record, while only one user failed to understand what Mystery 5 was doing.

6.4.2 Stage 2: Questionnaire

Once the participants had a chance to use Issie's truth table features, they were asked a series of questions about their experience. These questions fell into two categories. In the first section, they were presented with a five statements and asked to respond to them ranking them on a scale of 1 - 5, with 1 corresponding to "Strongly Disagree" and 5 corresponding to "Strongly Agree". The aim of this part of the questionnaire was to judge the quality of the user experience from different angles. The responses to this first set of questions were aggregated into a score by taking the average. Table 6.8 shows the score for each statement, as well as the rationale behind why the statement was included in the questionnaire.

The second set of questions in the questionnaire focused on determining the discoverability of the implemented features. Features in an application are only useful if they are discovered by the user, so observing which features are found is important. In addition to asking whether a named feature was found, the questions also asked if they found the feature useful. The rates of discovery and usefulness – i.e. the percentage of participants who discovered the features and found them useful, is shown in Figure 6.1.

Diagram	Explanation of Mystery Sheet	Score
	Mystery 1: Simple And Gate The first mystery sheet was intentionally made very easy. This was done to allow users to familiarise themselves with the application and the truth table generation method. The truth table for an AND gate is very basic and is one of the first things engineering students learn. The idea was that participants would be able to use this as a reference to understand the look and feel of truth tables in Issie.	100%
	Mystery 2: Multiplexer The second mystery sheet features a single MUX2 component which takes multi-bit inputs. This circuit was chosen as it exposes the user to truth tables which contain numbers other than 0 and 1. It also lets them DC Reduce the truth table, making it clear that B does not matter when SEL is 0, and A does not matter when SEL is 1.	100%
	Mystery 3: Full Adder The third mystery sheet is a full adder, which uses half adders custom components. Users can generate a truth table for the half adder alone by selecting it. This schematic also allows the user to explore algebraic reduction, as the arithmetic relationship can be inferred from the truth table.	100%
	Mystery 4: Bitwise And The fourth mystery sheet contains a large schematic with custom components and multiple wires. Such a schematic can be difficult to understand simply by looking at it. In contrast, algebraic reduction in the truth table identifies the schematic as being a simple bitwise And between the two inputs.	100%
	Mystery 5: Addition or Subtraction The fifth mystery sheet incorporates Issie's built-in arithmetic components; the N-bit Adder and the N-bit XOR. The MODE input controls whether the circuit adds the two inputs, or subtracts one from the other. This circuit is more complex than the previous mystery circuits, and the inputs A and B are both 16 bits wide. This introduces the user to truth table truncation. However, using algebra, the function of the circuit can be discovered.	90.1%

Table 6.7: Mystery Sheets

Statement	Rationale	Average Score
Finding the Truth Table tab in Issie's top-level UI was easy	Evaluates the top-level UI redesign of the application. If the user can find the truth table tab easily, this means that they will likely be able to find all simulation activities.	4.45
The functionality related to Truth Tables is clear and obvious in the UI.	Obviousness is a core principle of Issie, so evaluating whether the added features are presented in a way such that they are easy to find and understand.	4.63
Once I found a feature, it was easy to figure out what it did and how to use it.	Intuitiveness is a core principle of Issie, and this question aims to evaluate how easily the user can intuitively understand what features do.	4.45
Using truth tables made it easier to understand the relationship between inputs and outputs in combinational logic compared to looking at the schematic.	Evaluates the effectiveness of the truth table feature set as a whole in improving the visualisation of combinational logic in Issie.	4.36
The algebraic expressions in the truth table make it easy to understand the intended function of the circuit.	Evaluates the effectiveness of the algebraic expressions in particular for clearly communicating what the circuit does.	4.54

Table 6.8: First set of questions asked in questionnaire

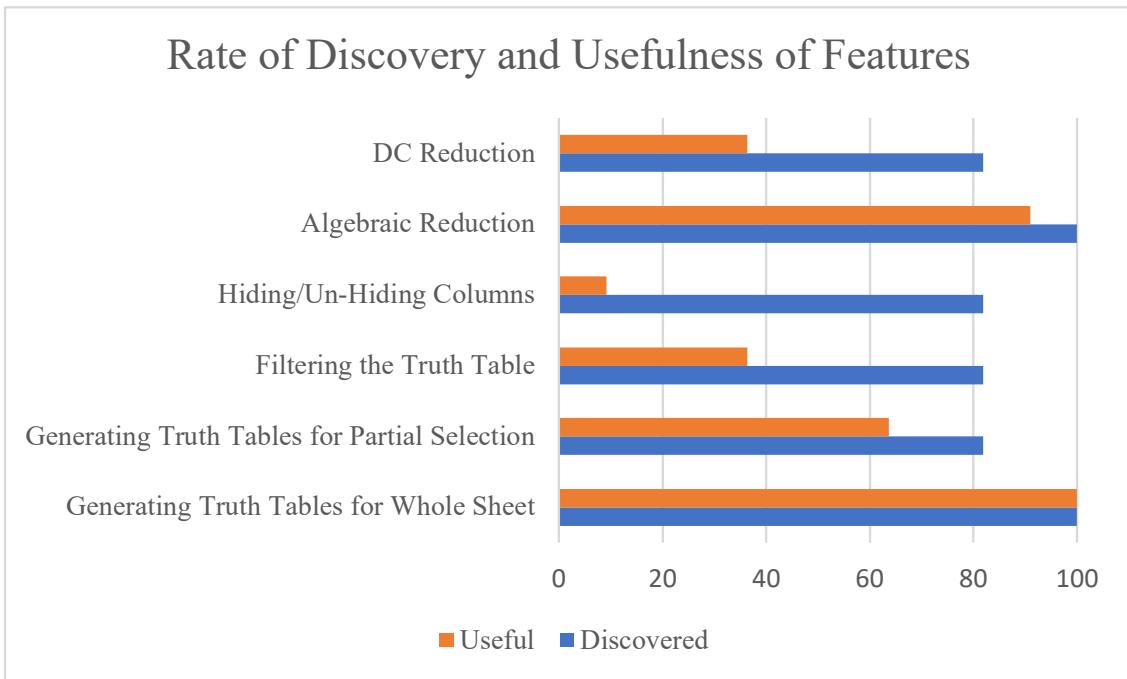


Figure 6.1: Graph showing the percentage of participants which discovered each feature and found it useful

6.5 Testing on the 8-bit ALU

To evaluate both the usefulness and correctness of the new combinational logic visualisation methods added to Issie, an 8-bit ALU was analysed using the added features. The schematic diagram of this ALU can be found in Figure C.1 in Appendix C. The inputs into the schematic have a total width of 23 bits, yielding a theoretical truth table size of over 8 million rows. The truth table is generated in under 1 second, but truncated to 1024 rows. Due to its size, and given that the exact specification of the ALU was not known at that point, the numeric truth table was not massively useful. However, had the specification been known prior to analysis, the numeric truth table would have undoubtedly been useful for checking the correctness of multiple input/output combinations. In order to further understand the ALU functions, algebraic reduction was used. The inputs A , B , and CIN were set as algebra; this reduced the truth table size from over 8 million rows to 64. The first 18 rows of this algebraic truth table can be seen in Figure 6.2. The other inputs, X and F , eventually propagate to the SEL ports of multiplexers and therefore cannot be set as algebra. However, this works out very well – these inputs control the behaviour of the ALU, meaning that it is better for their values to stay as numbers in the truth table. For each combination of these control inputs, a different expression at the output of the ALU (OUT) can be observed. From the algebraic truth table, the logical function of the ALU was described using eight short statements:

1. When $X = 0$ and $F = 0$, OUT is the sum of A and B
2. When $X = 0$ and $F = 2$, $OUT = A - B$
3. When $X = 0$ and $F = 6$, $OUT = CIN + A - B$
4. When $X = 0$ and $F = 4$, OUT is the sum of A , B , and CIN
5. When $X = 0$ and $F[0] = 1$ (i.e. F is odd), $OUT = B$
6. When $X = 1$, $OUT = A \& B$
7. When $X = 2$ or $X = 3$, OUT is the XOR of A and B
8. Otherwise, OUT is B right-shifted by 1.

$\langle X \rangle$	$\langle CIN \rangle$	$\langle F \rangle$	$\langle B \rangle$	$\langle A \rangle$	$\langle COUT \rangle$	$\langle OUT \rangle$	$\langle V \rangle$
0x0	CIN	0x0	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	(A[7] + B[7] + carry((A[6:0] + B[6:0])))::(A[6:0] + B[6:0])	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x4	CIN	0x0	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	0::B[7:1]	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x2	CIN	0x0	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	A + B	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x6	CIN	0x0	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	0::B[7:1]	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x1	CIN	0x0	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	A&B	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x5	CIN	0x0	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	0::B[7:1]	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x3	CIN	0x0	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	A + B	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x7	CIN	0x0	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	0::B[7:1]	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x0	CIN	0x1	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	B	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x4	CIN	0x1	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	0::B[7:1]	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x2	CIN	0x1	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	A + B	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x6	CIN	0x1	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	0::B[7:1]	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x1	CIN	0x1	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	A&B	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x5	CIN	0x1	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	0::B[7:1]	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x3	CIN	0x1	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	A + B	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x7	CIN	0x1	B	A	carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))	0::B[7:1]	carry((A[6:0] + B[6:0])) + carry((A[7] + B[7] + carry((A[6:0] + B[6:0]))))
0x0	CIN	0x2	B	A	carry((A[7] - B[7] + carry((A[6:0] - B[6:0])) - 1))	(A[7] - B[7] + carry((A[6:0] - B[6:0])) - 1)::(A[6:0] - B[6:0])	carry((A[6:0] - B[6:0])) + carry((A[7] - B[7] + carry((A[6:0] - B[6:0])) - 1))
0x4	CIN	0x2	B	A	carry((A[7] - B[7] + carry((A[6:0] - B[6:0])) - 1))	0::B[7:1]	carry((A[6:0] - B[6:0])) + carry((A[7] - B[7] + carry((A[6:0] - B[6:0])) - 1))

Figure 6.2: First 18 rows of the Algebraic Truth Table for 8-bit ALU

Chapter 7

Evaluation

In Chapter 1, the key aims for the project were established and were used to define the project. This chapter will evaluate whether these aims and deliverables have been met through a variety of methods. From these aims, a set of formal requirements was defined in Chapter 3, describing in detail what the project must complete in order to be considered successful. This chapter evaluates the final project deliverable against the aforementioned aims and requirements. This is done through a variety of methods, including evaluation against Issie's core principles, evaluation of key functionality in the application, analysis of performance data, as well as interpretation of user feedback.

7.1 Evaluation against Project Aims

The principal aim of the project was to explore novel ways in which interactivity could be added to automatic schematic-derived truth tables, and how these could be used as a fast aid to design combinational logic. Alongside the research, the main deliverable was stated as an updated version of Issie which featured these interactive truth tables. Additionally, an overarching aim was defined for the project; improving Issie in such a way that it is easier for students to understand the use of combinational logic in digital design. This section evaluates the improved version of Issie delivered by this project against these aims, ultimately concluding whether or not the aims were fulfilled.

Automatic schematic-derived truth tables have been implemented using repeated simulation of the schematic. These truth tables are generated through repeated simulation of all input combinations, with the whole process taking under 1 second. A caveat on these schematic-derived truth tables is that they can only handle an input space 10 bits wide, placing a limit of 1024 rows on truth table size. While this is a theoretical limitation, as discussed during the design of the system, in practice, it is unlikely that any user would be able to effectively comprehend a truth table of such a large size anyways. Additionally, the user has the option to re-generate the truth table with a tighter set of input constraints or with algebraic inputs to gain a more informative truth table. Both of these are examples of where *interactivity* has been added to the original schematic-derived truth table. Using input constraints, users can limit the input space and find/verify patterns in the resultant outputs. Seeing all the cases in a well-formatted list is more convenient than having to repeatedly run simulations and remember the result. Output constraints enable the reverse; users can specify a set of outputs and see which inputs into the logic cause them to occur. In addition to these features, truth tables can be sorted, rearranged, and can have some of their output columns hidden. These are all interactive features which allow the user to re-organise or reduce the truth table to make it more informative.

Establishing that interactive schematic-derived truth tables were added and discussing the possible uses of these features is only one part of the evaluation. Ultimately, the project can only be considered successful if the features added to Issie augment users' understanding of the combinational logic they are designing. To establish whether this was the case, 11 engineering students with beginner's level experience were provided with the updated version of Issie and asked to analyse five

'mystery' circuits using the truth table tab. Following this, they were asked a series of questions about their experience using the application. The methodology for this user experience testing is described in detail in Section 6.4, along with the results. 100% of participants were able to correctly identify the combinational logic function implemented by mystery circuits 1 to 4, while just over 90% were able to identify mystery sheet 5. These results imply that the added features to Issie do help users understand the relationships between inputs and outputs in combinational logic circuits. Additionally, the participants were themselves asked to evaluate the extent to which the updates were helpful. They were asked how much they agreed with the following statements:

Using truth tables made it easier to understand the relationship between inputs and outputs in combinational logic compared to looking at the schematic.

The algebraic expressions in the truth table make it easy to understand the intended function of the circuit.

The first statement had a score of 4.36, while the second statement had a score of 4.54. This indicates that most participants either agreed or strongly agreed with the statements. Therefore it can be said that the participants of the user experience survey agreed that the added methods for visualising combinational logic in Issie have helped them to better understand the relationships between inputs and outputs in the logic.

To further verify this conclusion, an 8-bit ALU was analysed using the truth table functionality added by the project. The purpose of the test was to use the features to ascertain the logical function of the circuit. The successfully deduced function, along with part of the algebraic truth table used to deduce it, is described in Section 6.5. The ALU, which has an input space of over 8 million possible combinations, was reduced to a truth table with only 64 rows through algebraic reduction. Following analysis of the algebraic truth table, the behaviour of the ALU was narrowed down to 8 distinct cases. The fact that the relationships between the inputs and outputs for such a complex schematic, featuring numerous components (including nested custom-components) can be inferred in this way suggests that the new methods for visualising combinational logic in Issie are effective and successfully fulfil the project aims.

7.2 Evaluation against Issie's Core Principles

Chapter 1 outlined the three key principles of Issie. These stated that all new features added to Issie must be robust, obvious, and intuitive. This section looks over the nature of the additions made to Issie, and evaluates whether they align with these principles.

7.2.1 Robustness

Software is considered robust when it is able to behave correctly under exceptional circumstances, such as when presented with erroneous or malformed user input. In Section 6.1, the process of analysing the codebase to verify against exceptions and failures is described. This process found that the majority of functions used in the project's codebase were exception-free, and the few functions which could throw exceptions had checks or system constraints implemented which would prevent invalid arguments being passed to them. A similar verification process was also undertaken for cases where `failwithf` is called, and also concluded that these cases would not occur due to checks and constraints. Finally, it was observed that even after hours of manual and user testing, no crashes or undefined behaviour were reported. These hours of use included a variety of interactions with the UI, many of which were malformed and would have resulted in errors were it not for the error checking and handling systems built into Issie.

This displayed resilience to crashes indicates that the decision to continue developing Issie in F# was the correct one. The majority of the application's data is immutable – therefore by extension most of the functions are referentially transparent (deterministic). The resulting lack of dependency on shared mutable state means that function behaviour is far more predictable. Another key aspect of F# which greatly adds to the robustness of the codebase is the fact that every language construct is an expression. Every expression must return a value, and all conditional branches must return a value of the same type. The effect of this is that every case must be considered and explicitly handled, reducing undefined behaviour. When combined with F# 's powerful type

inference, courtesy of its Hindley–Milner type system, this means that even complex conditional statements or pattern matches are correctly analysed by the compiler. The `evalExp` function, which is responsible for implementing the reduction rules for algebraic expressions, is an example of a large pattern match expression with many complicated cases. As well as generating warnings when specific expression cases weren't matched, it also did so when the cases were ordered in such a way that one would never be matched. This compile-time checking meant that many potential errors were caught as the code was being written, significantly reducing the likelihood of bugs being encountered at runtime.

In summary, following analysis of exception and failure cases, alongside observed stability in user trials and language constructs, it can be said that the updated version of Issie delivered by this project is robust.

7.2.2 Obviousness

Issie is built to be obvious – it should be clear to the user what is happening on their screen without any additional explanation. The user interface has been designed to be as clear as possible. Buttons are labelled with verbs which succinctly describe what they do, and are colour coded in a consistent manner with Issie's existing buttons. An example of this is the colour coding of the *Generate Truth Table* button. It is green if everything is correct and a truth table can successfully be generated, while it is yellow when there are issues that the user must resolve. In this scenario, clicking on the button clearly describes the error and highlights its source on the canvas. When the schematic contains sequential logic, the button is still green, but light in colour, communicating that while the schematic is correct, truth table generation is unavailable. Clicking this button then results in an error notification which informs users that truth table generation is only supported for combinational logic. In situations where the user is prevented from performing a certain action, such as applying an invalid constraint or DC reducing a truncated table, the button that would usually let them perform that action is greyed out and un-clickable. This disabling of the button, along with the changed mouse cursor, clearly communicates to the user that the action is forbidden under the current circumstances. A helpful warning message is either displayed near the button or as a tooltip to inform the user of the reason. This prevents them from becoming confused. Toggles are used in two places: hiding/un-hiding truth table columns and toggling inputs between numeric values and algebra. The toggle states are differentiated by colour as well as text. This means that from a glance it is obvious which of the two states is currently selected for each toggle.

While analysis of the various design patterns of the extensions made to Issie appears to show that they appear obvious, ultimately the obviousness of the features is best evaluated by real end users. The user questionnaire presented participants with the following statement:

The functionality related to Truth Tables is clear and obvious in the UI.

They were then asked to rank their opinion from Strongly Disagree (1) to Strongly Agree (5). As shown in Table 6.8, the average response score for this question was 4.63 out of 5. This indicates that most users either Agreed or Strongly Agreed that the truth table features in Issie are clear and obvious. Following the questionnaire, participants were given a free-form text box to voice any general feedback in. One response collected through this method stated that while most truth table functionality was clear, it was not hugely obvious what the "Reduce" button did at first glance. As the sheet that was open at the time did not contain any redundancies, the truth table did not change even after the user clicked the button out of curiosity. As a result, they were left slightly confused. This issue could be mitigated by 1) re-naming the button to something like "Remove Redundancies" to increase clarity, and 2) displaying a message stating there are no redundancies in the table if that is the case. With the exception of this comment, it is clear that the user experience concerning the obviousness of the UI is positive. This indicates that while small improvements could be made, on the whole, this project's additions to Issie align with its principle of obviousness.

7.2.3 Intuitiveness

Issie must be intuitive; users should not have to spend time and effort learning how to use features. This means that lab time can be spent learning Digital Electronics concepts, rather than figuring out the application. Consistency is a key factor in building an intuitive environment – if multiple

features work in the same predictable way then there is less for the user to deduce from the application. Care has been taken to maintain consistency with Issie's existing features, with code and behaviour being reused from the existing Step Simulator wherever possible to improve the experience of both the user and future developers. To ascertain the intuitiveness of the updated version of Issie, the user questionnaire presented participants with the following statement, and asked to what extent they agreed with it.

Once I found a feature, it was easy to figure out what it did and how to use it.

As shown in Table 6.8 this statement had an average response of 4.45, implying that as a whole users agreed or strongly agreed that features were easy to understand and use once they were found. This in turn indicates that the features are intuitive, and therefore align with Issie's core principles.

7.3 Evaluation of Application Performance

Section 6.3 describes the measured performance for the application, along with the methodology used to obtain those results. From this, it is clear that the performance criteria set out in the requirements have been met. The average time taken to generate a truth table for a complex schematic is around 700ms, while other operations (bar one) on the truth table take under 100ms. The worst performing operation on the truth table by far is Don't Care reduction. On sheets with no redundancies, it still takes over 250ms on average to detect this. On sheets with multiple redundancies, such as a simple Mux4 circuit, it took up to around 2.6 seconds to completely reduce the truth table. This poor performance is likely due to brute-force nature of the reduction algorithm, which has a poor time complexity. If the n represents the number of rows in the truth table, and m represents the number of inputs, each call of the reduction algorithm has a worst case complexity of $O(m^3n^3)$. For a truth table with r recursive inputs, the recursion function is called $r + 1$ times. This terrible time complexity is likely due to the fact that the algorithm applies a brute-force approach to DC reduction. For each input, all possible 'Don't Care' rows are calculated, and these are then each tested to check whether they are valid. Switching to some heuristic-based method will likely bring performance improvements.

The use of immutable data structures also affects application performance. Operations that would usually mutate contents of data structures instead copy large chunks of data – this adds overhead. Updating the values at specific indices in a list is particularly expensive. This operation is $O(n)$ by itself as F# lists are implemented as singly-linked lists, and on top of this the entire list must be copied to a new location in memory. Therefore, it is likely that badly performing functions, such as the brute-force reduction algorithm, could be optimised by replacing immutable data structures (e.g. Lists and Maps) with mutable structures (e.g. Arrays). This technique was successfully used to improve the efficiency of the Step Simulator earlier during Issie's development. However, even with its performance issues, DC reduction still takes less than 4 seconds, meaning that it fulfils the performance requirements outlined in Chapter 3.

From a qualitative perspective, the application feels lightweight and responsive. No action (other than DC Reduction) creates a delay that loses the user's attention, and all UI operations appear instantaneous. This perception aligns with the measured timings in accordance with Robert Miller's descriptions of perceived responsiveness, which were summarised in Section 2.7. This responsiveness means that the updated version of Issie, much like its predecessor, does not lose the user's attention and has greater perceived interactivity, which in turn improves the learning experience.

7.4 Comparison Against Issie 3.0.0

As of the time of writing, the latest release of Issie is version 3.0.0. In this version of Issie, combinational logic can only be visualised by either viewing the schematic diagram or simulating specific input combinations in the step simulator. This section compares the updated version of Issie delivered by the project with Issie 3.0.0, highlighting the different areas in which the updated version improves Issie. Issie 3.0.0 is referred to as the base version, while the version of Issie delivered by the project is referred to as the updated version.

Consistent Top-Level UI Issie 3.0.0's top-level UI was analysed in 2.5. One major issue found with it was its lack of consistency for running simulations. In Issie 3.0.0, the step simulator is accessed via a tab in the right section, while the waveform simulator is launched via a button in the top bar of the application. This results in a fourth tab appearing in the right section, which contains the actual wave simulation interface. This was deemed too inconsistent and confusing, and was therefore improved upon in the updated version of Issie. Now, all simulation activity (Step Simulator, Truth Table, Waveform Simulator) is contained in sub-tabs underneath a main simulations tab in the right section.

Viewing all Input/Output relationships For smaller schematics (those that will not be truncated), truth tables allow for all pairs of input and output combinations to be viewed in one go. This is in contrast to the base version of Issie, where the user has to manually run multiple simulations. The results of these simulations are not stored anywhere, so to view all relationships in one place the user must write down the result of each simulation prior to simulating the next input combination. In addition to describing the whole schematic in a single, organised structure, this functionality is also useful in situations where the user may want to verify a specific batch of relationships hold true for some logic.

Easier to verify schematic behaviour A major advantage of being able to view multiple pairs of input and output combinations at once is that it becomes much easier to verify whether the circuit built by the user actually implements the required function. In Issie 3.0.0, users can only verify circuit behaviour by manually running different simulations. This is a time-consuming and tedious process, meaning that users are likely to not do it properly. In contrast, not only do truth tables allow multiple simulation results to be viewed in one go, but they can be filtered with input constraints to allow results for specific cases to be viewed together. Algebraic truth tables take this one step further, with algebraic expressions representing the logical function for even easier verification.

Debugging parts of a schematic When designing a schematic, there may be situations where a user has some part of their logic working correctly, but has errors (either syntactical or logical) elsewhere in the schematic. The ability to generate truth tables for partial selections of the sheet means that users can investigate or verify the behaviour of specific parts of the schematic, regardless of whether the rest of the schematic is complete. This has many benefits; when designing schematics users can test parts of the sheet as they go. When analysing/debugging schematics, users can take a divide and conquer approach by establishing what specific parts of the sheet do. In the base version of Issie, sub-parts of a design can only be simulated if they are contained within separate sheets. Therefore, if a user wished to debug a specific part of the logic in a sheet, they would have to recreate it in a separate sheet. This is a tedious process compared to simply selecting that part of the logic and generating a truth table for it.

Redundancy Detection Through reduction with Don't Cares, redundancies in a logic design can be identified in the updated version of Issie. While this feature was envisaged as an analysis tool, it also aids logic design. If, due to some mistake an input is redundant in the designed logic, Don't Care reduction would clearly label it as so. In contrast, it may be difficult to spot such an issue when simply running simulations with the step simulator.

7.5 Evaluation of Filtering Methods

The size of the truth table can be reduced by filtering it with input and output constraints. The section for adding these constraints is collapsed under the *Filter* menu option in the truth table tab. Input constraints are generally an effective way of manipulating an existing truth table to check for specific cases, especially when used on algebraic truth tables. For example, when analysing the ALU in Section 6.5, the input combinations which led to the 8 distinct cases were isolated and verified using input constraints on X and F .

Output constraints can be used to observe which input combinations result in some specified output combination(s). This feature is often most effective when used in conjunction with other features. Prior to the implementation of algebraic truth tables, output constraints were very useful

for finding which input combinations resulted in specific behaviour. For example, the following steps were used to determine the cases where the ALU was performing subtraction:

1. Generate the numeric truth table
2. Using input constraints, set the value of A to a number, and B to a smaller number (e.g. 21 and 7)
3. Add the output constraint where OUT is equal to $A - B$ (e.g. 14)
4. Observe the values of the other controlling inputs X and F

While useful, there are limitations on output constraints. They are not as useful on truncated truth tables, because they filter the existing truncated truth table (1024 rows), rather than the complete theoretical truth table like the input constraints do. This means that the user may make incorrect deductions about the logic. Users are warned about this when they add output constraints, but there is a possibility that they might miss this warning. Unfortunately, there is not much that can be done to address this issue completely, as applying output constraints requires simulating the entire input space and then filtering the corresponding output space; this would be far too time-consuming for schematics with large input spaces. Another limitation of output constraints in Issie is that they are purely numerical, meaning that they cannot be applied to algebraic truth tables. Pure numerical output constraints also lack the ability to filter numeric truth tables based on relationships. Both these limitations could be overcome by introducing algebraic output constraints. The possibilities unlocked by implementing these in the future are discussed in Section 8.1.1.

7.6 Evaluation of Reduction Methods

7.6.1 Algebraic Truth Tables

Truth tables can be reduced by turning them into Algebraic truth tables. Users set specific inputs to algebraic terms, and the outputs in the truth table are displayed as functions of the algebraic terms. In the user experience questionnaire, a majority of participants either agreed or strongly agreed that algebraic expressions in the truth table made it easier to understand the intended function of the circuit. This, in addition to the successful analysis of the ALU, indicates that the implementation of algebraic expressions in Issie has been effective and in line with the aims of the project. One of the major factors behind this effectiveness is the comprehensive set of algebraic reduction rules defined in Section 5.7.2, which can recognise and interpret a wide array of patterns. However, there were only a finite number of cases which could be implemented over the course of the project, meaning that there are some limitations to the extent certain schematics can be interpreted.

The N-bits Adder component in Issie is used for arithmetic operations, and this behaviour is used when constructing algebraic expressions during algebraic simulation. However, users can define their own ripple carry adders. Ripple-carry adders implement n-bit addition using multiple full adder components, meaning that they are built purely out of logic gates. This means that detecting arithmetic done using a ripple-carry adder requires a very complex reduction rule which checks for a specific combination of Boolean operations. A complex rule has already been written to detect full adders, but the additional challenge with ripple carry adders is that they can be n-bits wide. While the detection of n-bit ripple carry adders was investigated, it was ultimately not implemented due to the complexity of the task and time limitations of the project. However, it could be implemented in the future.

7.6.2 Don't Care Reduction

Non-truncated truth tables in Issie can have redundant rows removed via reduction with Don't Cares. This is implemented using a recursive algorithm. This functionality was tested on multiplexers, as well as combinations of logic gates and was found to work as intended. While powerful in specific cases, it has been overshadowed by algebraic reduction, which reduces the truth table to a greater extent and is far more versatile. In fact, when prompted for general feedback in the user experience questionnaire, one participant wrote:

"While the reduce button does work, the algebra feature is much better so it's kind of redundant?"

While it is ironic that a feature implemented to remove redundancies in the truth table may itself be considered redundant by a user, the participant does raise a somewhat valid point. In most situations, algebraic reduction results in a smaller truth table, while the algebraic expressions are far more readable and information dense compared to numerous Don't Care (X) terms spread across rows. Another limitation of DC reduction is that it only marks inputs as redundant when they do not affect **all of the outputs** in that row. In contrast, information on whether a given input affects a specific output can be obtained from algebraic expressions; if the output expression does not contain a specific input label then that input is redundant with regard to that output. However, DC reduction may be useful in circuits where certain key inputs into the logic may be passed to the SEL port of a multiplexer, resulting in algebraic reduction not being as useful.

A more solvable issue with DC reduction is that seems to not align completely with the Issie principle of obviousness. *Reduce* is quite a vague term and it may be a better idea to change the button text to something like "Remove Redundancies". In addition to this, users are currently not informed when no redundancies are found in the truth table. As a result, it appears that pressing the button did nothing. Two possible solutions to this issue could be:

1. Reduce button behaves like *Start Simulation* or *Generate Truth Table* button. The table is DC reduced in the background, and if no redundancies are found the button is greyed out.
2. The system stays as it currently is, but if no redundancies are found in the reduction process the user is informed using a popup.

While the former would be more intuitive and obvious, it would involve reducing the truth table in the background. This is a time-consuming process which could decrease the responsiveness of the application. On the other hand, while the latter approach would require an extra click, it would be much easier to implement. It is likely that one of these solutions will be implemented prior to the merger with the master branch of Issie.

Evaluating DC Reduction as a whole, while it is inferior to algebraic reduction, there are specific niche cases where it may still be useful. Once the features added to Issie by this project are tested by a wider user base during Digital Electronics labs at Imperial College London, a decision can be made on whether it is worth keeping in the application.

7.7 Evaluation against Requirements

This project has two main deliverables; an extended and improved version of Issie, and any appropriate documentation for the application. At the beginning of the project, a series of requirements were formalised which determined what the project should aim to accomplish, and under what circumstances could it be considered successful.

Table 7.1: Evaluation against Requirements

Requirement	Comment	Implemented?
LOGIC VISUALISATION		
E1.1	A numeric truth table can be generated for a sheet containing combinational logic. This truth table is exhaustive when the sum of input widths is under 10. Otherwise, it is still correct, but truncated.	Yes
E1.2	A numeric truth table can be generated for a partial selection of a sheet. This truth table is exhaustive when the sum of input widths is under 10. Otherwise, it is still correct, but truncated.	Yes
E1.2.1	New inputs and outputs are created to form a correct Issie schematic.	Yes

Continued on next page

Table 7.1: Evaluation against Requirements (Continued)

Requirement	Comment	Implemented?
E1.2.2	The newly generated inputs and outputs have intelligently inferred labels based on which component port they are connected to.	Yes
E1.3	The truth table generation algorithm is versatile and can handle any combinational Issie schematic.	Yes
E1.3.1	Multi-bit inputs and outputs are supported. Temporary inputs/outputs created while generating a truth table for a selected logic block have correct widths inferred using either <code>WidthInferrer</code> or the connected component.	Yes
E1.3.2	Custom Components (sub-sheets) are supported, including when they are part of selections. However, all custom component ports must be connected so as to allow <code>WidthInferrer</code> to find port widths.	Yes
E1.3.3	Inputs, Outputs, and Viewers are all shown in the truth table	Yes
E1.4	Users have the option to reduce the truth table using Don't Cares. This is only applicable to un-truncated tables.	Yes
E1.5	Filtering of truth tables with input and output constraints has been implemented. Caveat on output constraints is that they only filter the generated table, so does not return the full set for a truncated table.	Yes
E1.6	Truth tables are displayed in a clear and easy to understand format, with striping to make differentiating rows easier. Features involving truth tables are presented in a menu which can be easily explored and clicked through. Messaging is consistent and guides the user.	Yes
E1.7	Truth table generation takes place in under 1 second, while reduction times for the largest possible table are consistently under 3 seconds. The fast generation time is due to truncation, but there is not much value in generating more rows. Furthermore, generating more rows than this makes the UI feel sluggish.	Yes
E1.8	Graphical manipulation operations on the Truth Table, such as re-ordering rows, sorting etc. appear instantaneous (i.e. take less than 100ms).	Yes
E1.9	Through truth table reduction and viewing of algebraic truth tables, complex relationships implemented by large circuits, such as ALUs, can be summarised using a few different expressions.	Yes
D1.1	Algebraic truth tables have been added to Issie. Expressions are calculated using algebraic simulation.	Yes

Continued on next page

Table 7.1: Evaluation against Requirements (Continued)

Requirement	Comment	Implemented?
D1.1.1	Multiplexers: Both library and gate-level multiplexers are recognised correctly. Library N-bit Adders are supported, as well as Half and Full adders defined by the user using gates. User-defined ripple carry adders are not recognised perfectly, but do still give an idea of the addition.	Yes
D1.1.2	Algebraic truth table can be calculated for all combinational circuits, with the only limitation being that algebra cannot be passed to SEL ports.	Yes
D1.2	A fairly interactive truth table interface has been provided, with responsive and intuitive manipulations available. However, some additional work could be done to make the truth table more interactive.	Yes
D1.2.1	Implementing functionality where mousing over parts of the truth table could highlight parts of the schematic was considered, however this would require changing draw block behaviour. Not only would implementing this be time consuming, but outside of annotating specific IOs not much potential was seen in this approach. Other interactivity, such as highlighting cells in truth tables was considered, but was not implemented due to time constraints.	Yes
D1.2.2	Users can rearrange order of columns/rows in the truth table.	Yes
D1.2.3	Users can sort the truth table in ascending and descending order.	Yes
D1.1	The user can access truth table related functionality easily – all features are contained within the truth table tab, and are either presented on the truth table itself (sorting and moving columns), or grouped under a labelled menu section. The user experience survey backed this up.	Yes
SOFTWARE/DOCUMENTATION QUALITY		
E2.1	The project has delivered performant, working, bug-free code which adheres to Issie's code guidelines and other principles such as "MVU-ness". Performance has been tested quantitatively, while correctness and resistance to failure has also been verified.	Yes
E2.2	XML comments have been written for all functions in the delivered code, alongside other inline comments to explain how certain key parts work in order to make the codebase more maintainable for future developers.	Yes

Continued on next page

Table 7.1: Evaluation against Requirements (Continued)

Requirement	Comment	Implemented?
E2.3	Code has been written with maintainability in mind. Care has been taken to use standard library data structures and functions as much as possible, and any newly introduced types and processes have been documented extensively in the code.	Yes
D2.1	Certain UI changes, such as moving the Waveform simulator and fixing bugs related to the dividerbar were implemented. The UI was not redesigned , however it was evaluated and its current form appears to be adequate.	Partially
D2.2	The Issie website was not updated over the duration of the project itself. However, it will be updated in due course.	Not yet

7.8 Summary

In this section, the updated version of Issie delivered by this project was evaluated against the initial project aims, Issie's Core Principles, and the requirements set out at the beginning of the project. In the first two sections, using user questionnaire results, observations made about the application, and other results from the testing stage, it was established that all project aims had been met and all core principles had been maintained. In the evaluation against requirements, it was established that all Essential requirements, and all bar one Desired requirements, have been fulfilled by the project. Certain areas of possible improvements were also identified. Three issues with Don't Care reduction were found – these are a lack of obviousness, possible redundancy in the system, and performance. While the first issue has a fairly straightforward fix that will most likely be implemented soon, the others will require further analysis and effort to solve. Cases where certain features could be improved were also identified. This includes expanding the algebraic reduction rules to cover more cases, and implementing algebraic output constraints.

To summarise, the updated version of Issie delivered by this project comfortably fulfils all of the metrics and requirements defined for it at the beginning of the project. Therefore it, and the project as a whole can be considered a success.

Chapter 8

Conclusion and Further Work

In conclusion, this project delivered an updated version of Issie, featuring interactive automatic schematic-derived truth tables which can be filtered, manipulated, and reduced using either Don't Care terms or Algebra. These truth tables offer users of Issie a novel way to visualise combinational logic relationships in their designs, aiding the design process and improving users' understanding of Digital Electronic design. With the majority of survey participants either agreeing or strongly agreeing that the added truth tables and algebraic expressions made it easier to understand the relationships between combinational inputs and outputs, it is evident that this novel visualisation technique is a valuable tool in Issie's arsenal for improving the learning experience for those learning digital logic design. Algebraic truth tables in particular have been very effective, condensing the millions of possible combinations in a complex ALU circuit into only 64 rows, which could then easily be further reduced by hand to specification comprising only eight lines. Through the collection of user feedback, it has been established that the novel logic visualisation methods added to Issie make it easier for users to better understand the relationships between inputs and outputs in combinational logic. All of the project aims and requirements have been met, meaning that this project can be considered successful.

This project has presented many unique and interesting challenges, and the process of overcoming them has been immensely rewarding. One of the initial challenges of the project was simply getting started with the implementation, as it involved working on an existing codebase which exceeded 20,000 lines of F# code. Therefore, prior to the implementation phase, a thorough analysis of the codebase was undertaken to create a solid platform of understanding upon which the new features could be built.

Writing the canvas correction algorithm, which enables users to generate truth tables for partial selections of a sheet, was also particularly challenging. From the outset, Issie has been effective at clearly describing errors to users, but has never taken steps to automatically correct these errors itself. As a result, all pre-existing simulation code is expected to take only syntactically correct canvas states. A major challenge with correcting a partial canvas state is that the user is free to make any selection possible. Therefore there is no guarantee of the form of the canvas state the algorithm may receive. Inferring the user's intentions from this unpredictable canvas state and generating a useful truth table with correct widths and informative IO labels was a complex process which posed both conceptual and implementation challenges.

Another task which was conceptually challenging was the design of the new formal language for algebra, in particular the definition of reduction rules. Some reduction rules, such as those for arithmetic simplification, appear simple on the surface but are far more difficult to generalise within a given system. Mixing Boolean algebra, fixed-width arithmetic, and other bus operations to create a novel algebraic system was in itself a stimulating task, and implementing methods to reduce these expressions was even tougher. However, the hard work undertaken to overcome these challenges has paid off; the delivered system is capable of simplifying complex circuits into succinct and informative algebraic truth tables.

8.1 Possible Further Work

While this project has undoubtedly been a successful endeavour, there are a few areas where its work could be extended to achieve an even better outcome. This section highlights some of these possible extensions.

8.1.1 Algebraic Output Constraints

As mentioned during the evaluation, output constraints can currently only be applied to numeric truth tables. One possible improvement to be implemented in the future could be the introduction of output constraints whose right-hand side is an algebraic expression. This would have two advantages. The first is that the outputs of the algebraic truth table could be successfully constrained – only algebraic expressions which matched the output constraints would remain in the displayed table. The second is that algebraic constraints would also allow for a more intelligent filtering of numeric truth tables. For example, the output constraint $OUT = A + B$ would filter the table so that only rows in which the output was the sum of inputs A and B would be permitted. Such a feature would likely help users better analyse and understand the logic being designed.

8.1.2 More Algebraic Reduction Rules

Expanding the set of formal algebraic reduction rules would allow for the recognition of more complex constructs. One such example would be the recognition of ripple-carry adders. Currently, the algebraic truth table does recognise ripple-carry adders to some extent. For example, the output of a two-bit adder is:

$$(A[1] + B[1] + \text{carry}(A[0] + B[0])) :: (\text{carry}(A[0] + B[0]))$$

However, ideally this would be simplified to $A + B$. The pattern to be checked for is recursive in nature, and would require a more formal definition prior to implementation in the future.

8.1.3 Adding Algebra to the Step Simulator

During the project, the Fast Simulation code was extended to support algebraic expressions. This was done so that algebraic truth tables could be implemented. The Step Simulator also uses the Fast Simulator to run simulations – therefore adding support for algebraic inputs and outputs to the step simulator would only require a few changes, with the bulk of those being changes to the UI to allow an input to be toggled between algebra and numeric values. For the purposes of consistency, UI elements like the algebra selector popup could be re-used from the existing truth table codebase.

8.2 User Guide

Instructions on how to install Issie can be found on the Github page: <https://github.com/tomcl/issie>. This page also contains a link to the Issie website, which contains a guide on how to use the application.

Bibliography

- [1] “Issie - an intuitive cross-platform hardware design application.” (), [Online]. Available: <https://github.com/tomcl/issie>. (accessed: 10.11.2021).
- [2] M. Selvatici, *Deflow: An extensible hardware design platform for teaching digital electronics*, 2020.
- [3] “A model-based approach for robustness testing.” (), [Online]. Available: <https://dl.ifip.org/db/conf/pts/testcom2005/FernandezMP05.pdf>. (accessed: 03.01.2022).
- [4] K. J. Malmberg, J. G. Raaijmakers, and R. M. Shiffrin, “50 years of research sparked by atkinson and shiffrin,” *Memory and Cognition*, vol. 47, pp. 561–574, 2019. DOI: <https://doi.org/10.3758/s13421-019-00896-7>.
- [5] S. McLeod. “Multi-store model of memory.” (2021), [Online]. Available: <https://www.simplypsychology.org/multi-store.html>. (accessed: 03.01.2022).
- [6] N. Fleming and C. E. Mills, “Not another inventory, rather a catalyst for reflection,” 1992. [Online]. Available: https://vark-learn.com/wp-content/uploads/2014/08/not_another_inventory.pdf, (accessed: 04.01.2022).
- [7] B. D. Ictenbas and H. Eryilmazb, “Determining learning styles of engineering students to improve the design of a service course,” *Procedia - Social and Behavioral Sciences*, vol. 28, pp. 342–346, 2011. DOI: <https://doi.org/10.1016/j.sbspro.2011.11.065>.
- [8] F. Guay, S. Larose, and M. Boivin, “Academic self-concept and educational attainment level: A ten-year longitudinal study,” *Self and Identity - SELF IDENTITY*, vol. 3, pp. 53–68, Jan. 2004. DOI: [10.1080/13576500342000040](https://doi.org/10.1080/13576500342000040).
- [9] S. McLeod. “Constructivism as a theory for teaching and learning.” (2019), [Online]. Available: <https://www.simplypsychology.org/constructivism.html>. (accessed: 04.01.2022).
- [10] “Combinational logic.” (), [Online]. Available: https://en.wikipedia.org/wiki/Combinational_logic. (accessed: 10.11.2021).
- [11] J. R. Gregg, “The basic functions of boolean algebra: And, or, and not,” in *Ones and Zeros: Understanding Boolean Algebra, Digital Circuits, and the Logic of Sets*. 1998, pp. 22–36. DOI: [10.1109/9780470545423.ch1](https://doi.org/10.1109/9780470545423.ch1).
- [12] Arm. “Modular programming.” (2019), [Online]. Available: https://www.keil.com/support/man/docs/a251/a251_in_modular.htm. (accessed: 15.05.2022).
- [13] N. M. Morris, *Logic Circuits*. 1969. [Online]. Available: <https://books.google.co.uk/books?id=4T7jVPmu40cC>.
- [14] J. Y. Hsu, *Computer Logic Design Principles and Applications*, eng, 1st ed. 2002. New York, NY: Springer New York, 2002, ISBN: 1-4613-0047-9.
- [15] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vicentelli, “Espresso-signature: A new exact minimizer for logic functions,” eng, *IEEE transactions on very large scale integration (VLSI) systems*, vol. 1, no. 4, pp. 432–440, 1993, ISSN: 1063-8210.
- [16] T. U. of Iowa. “Input file format for espresso (pla-file).” (), [Online]. Available: <http://user.engineering.uiowa.edu/~switchin/0ldSwitching/espresso.5.html>. (accessed: 03.06.2022).
- [17] *Three-input multiplexer datasheet*, CD74AC251, CD74ACT251, Texas Instruments, 1998.
- [18] P. Carter, D. Syme, and N. Schonning. “What is f#.” (2021), [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/fsharp/what-is-fsharp>. (accessed: 10.01.2022).

- [19] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, “A large-scale study of programming languages and code quality in github,” *Commun. ACM*, vol. 60, no. 10, pp. 91–100, Sep. 2017, ISSN: 0001-0782. DOI: 10.1145/3126905. [Online]. Available: <https://doi.org/10.1145/3126905>.
- [20] L. M. M. Damas, “Type assignment in programming languages,” Ph.D. dissertation, University of Edinburgh, Edinburgh, Scotland, UK, 1984.
- [21] Microsoft. “Discriminated unions.” (), [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions>. (accessed: 05.06.2022).
- [22] L. Hoban, “Clr inside out - f# fundamentals,” *MSDN Magazine*, vol. 25, no. 6, 2010.
- [23] “Fable: Javascript you can be proud of!” (), [Online]. Available: <https://fable.io/>. (accessed: 11.01.2022).
- [24] Github. “Electron docs.” (2021), [Online]. Available: <https://www.electronjs.org/docs/latest>. (accessed: 11.01.2022).
- [25] B. Geary. “Web apps vs native desktop apps.” (2018), [Online]. Available: <https://www.andplus.com/blog/web-apps-vs-native-desktop-apps>. (accessed: 15.05.2022).
- [26] M. Shpilt. “9 must decisions in desktop application development for windows.” (2018), [Online]. Available: <https://michaelscodingspot.com/9-must-decisions-in-desktop-application-development-for-windows/>. (accessed: 15.05.2022).
- [27] “Electron memory usage compared to other cross-platform frameworks.” (2017), [Online]. Available: <http://roryok.com/blog/2017/08/electron-memory-usage-compared-to-other-cross-platform-frameworks/>. (accessed: 15.05.2022).
- [28] “React: A javascript library for building user interfaces.” (2022), [Online]. Available: <https://reactjs.org/>. (accessed: 11.01.2022).
- [29] “The elm architecture.” (2022), [Online]. Available: <https://guide.elm-lang.org/architecture/>. (accessed: 11.01.2022).
- [30] “Introduction to elm.” (2022), [Online]. Available: <https://guide.elm-lang.org/>. (accessed: 11.01.2022).
- [31] “Elmish.” (2022), [Online]. Available: <https://elmish.github.io/elmish/>. (accessed: 11.01.2022).
- [32] T. Bandt, “Suitability of functional programming for the development of mobile applications: A comparison of f# and c# involving xamarin,” IUBH University of Applied Sciences, Bad Honnef, Germany, 2019, (DOI: <http://dx.doi.org/10.2139/ssrn.3459433>).
- [33] P. Poudel. “Virtual dom.” (2018), [Online]. Available: <https://elmprogramming.com/virtual-dom.html>. (accessed: 12.01.2022).
- [34] “Rendering elements.” (2022), [Online]. Available: <https://reactjs.org/docs/rendering-elements.html>. (accessed: 12.01.2022).
- [35] K. Khosravi. “Options for optimizing caching in react.” (2021), [Online]. Available: <https://blog.logrocket.com/options-caching-react>. (accessed: 25.01.2022).
- [36] “Fulma.” (2022), [Online]. Available: <https://fulma.github.io/Fulma/>. (accessed: 12.01.2022).
- [37] “Bulma.” (2022), [Online]. Available: <https://bulma.io/>. (accessed: 12.01.2022).
- [38] “Fulma table example.” (2022), [Online]. Available: <https://fulma.github.io/Fulma/#fulma/elements/table>. (accessed: 12.01.2022).
- [39] M. D. L. Riva. “Why consistency is so incredibly important in ui design.” (2021), [Online]. Available: <https://careerfoundry.com/en/blog/ui-design/the-importance-of-consistency-in-ui-design/>. (accessed: 15.01.2022).
- [40] G. W. Ph.D. and G. W. Ph.D., “Conceptualizing and measuring the perceived interactivity of websites,” *Journal of Current Issues & Research in Advertising*, vol. 28, no. 1, pp. 87–104, 2006. DOI: 10.1080/10641734.2006.10505193. eprint: <https://doi.org/10.1080/10641734.2006.10505193>. [Online]. Available: <https://doi.org/10.1080/10641734.2006.10505193>.

- [41] R. B. Miller, “Response time in man-computer conversational transactions,” in *AFIPS '68 (Fall, part I)*, 1968.
- [42] *Agile Software Development Current Research and Future Directions*, eng, 1st ed. 2010. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN: 1-282-92565-2.
- [43] A. Deshpande, “Comparing and contrasting the more traditional prince2 and newer agile project management methods,” Imperial College London, Tech. Rep., 2021, Unpublished essay for Managing Engineering Projects module coursework.
- [44] M. Rehkopf. “Stories, epics, and initiatives.” (2022), [Online]. Available: <https://www.atlassian.com/agile/project-management/epics-stories-themes>. (accessed: 18.01.2022).
- [45] T. Chow and D.-B. Cao, “A survey study of critical success factors in agile software projects,” *Journal of Systems and Software*, vol. 81, no. 6, pp. 961–971, 2008, Agile Product Line Engineering, ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2007.08.020>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121207002208>.
- [46] D. P. Voorhees, “Introduction to software design,” in *Guide to Efficient Software Design: An MVC Approach to Concepts, Structures, and Models*. Cham: Springer International Publishing, 2020, pp. 1–15, ISBN: 978-3-030-28501-2. DOI: [10.1007/978-3-030-28501-2_1](https://doi.org/10.1007/978-3-030-28501-2_1). [Online]. Available: https://doi.org/10.1007/978-3-030-28501-2_1.
- [47] M. Mishra and S. Akashe, “High performance, low power 200 gb/s 4:1 mux with tgl in 45 nm technology,” *Applied Nanoscience*, vol. 4, Mar. 2013. DOI: [10.1007/s13204-013-0206-0](https://doi.org/10.1007/s13204-013-0206-0).
- [48] “Map<'key, 'value> type.” (), [Online]. Available: <https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-fsharpmap-2.html>. (accessed: 05.06.2022).
- [49] Microsoft. “Sequences.” (), [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/sequences>. (accessed: 06.06.2022).
- [50] P. Hudak, “Conception, evolution, and application of functional programming languages,” *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, Sep. 1989, ISSN: 0360-0300. DOI: [10.1145/72551.72554](https://doi.org/10.1145/72551.72554). [Online]. Available: <https://doi.org/10.1145/72551.72554>.
- [51] G. Jäger and J. Rogers, “Formal language theory: Refining the chomsky hierarchy,” *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 367, no. 1598, pp. 1956–1970, 2012. DOI: [10.1098/rstb.2012.0077](https://doi.org/10.1098/rstb.2012.0077).
- [52] D. G. Hays, “Chomsky hierarchy,” in *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003, pp. 210–211, ISBN: 0470864125.
- [53] D. D. McCracken and E. D. Reilly, “Backus-naur form (bnf),” in *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003, pp. 129–131, ISBN: 0470864125.
- [54] Z. Durrani, *Lecture 4: Boolean algebra*, Oct. 2018, (accessed: 11.11.2021).
- [55] C. House. “A complete guide to grid.” (Mar. 2022), [Online]. Available: <https://css-tricks.com/snippets/css/complete-guide-grid/>. (accessed: 12.06.2022).
- [56] J. Enders. “Zebra striping: More data for the case.” (Sep. 2008), [Online]. Available: <https://alistapart.com/article/zebrastripingmoredataforthecase/>. (accessed: 12.06.2022).

Appendix A

Screenshots of Issie 3.0.0

Screenshots taken on MacOS.

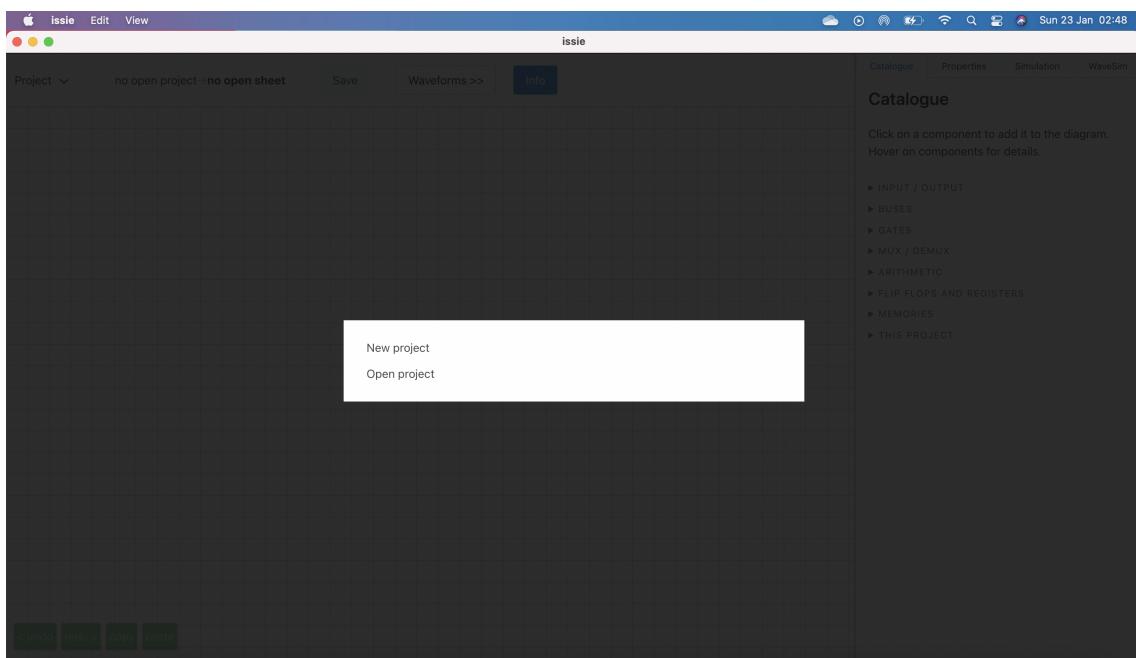


Figure A.1: Issie's opening screen

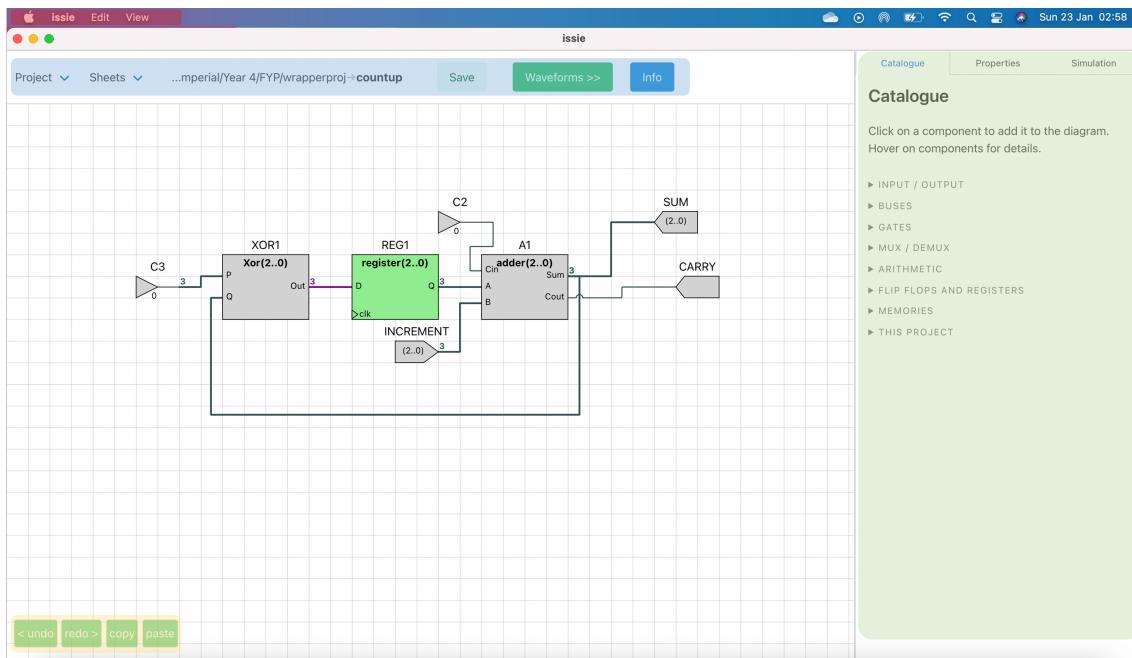


Figure A.2: A sheet (schematic) open in Issie, with a component and wire selected

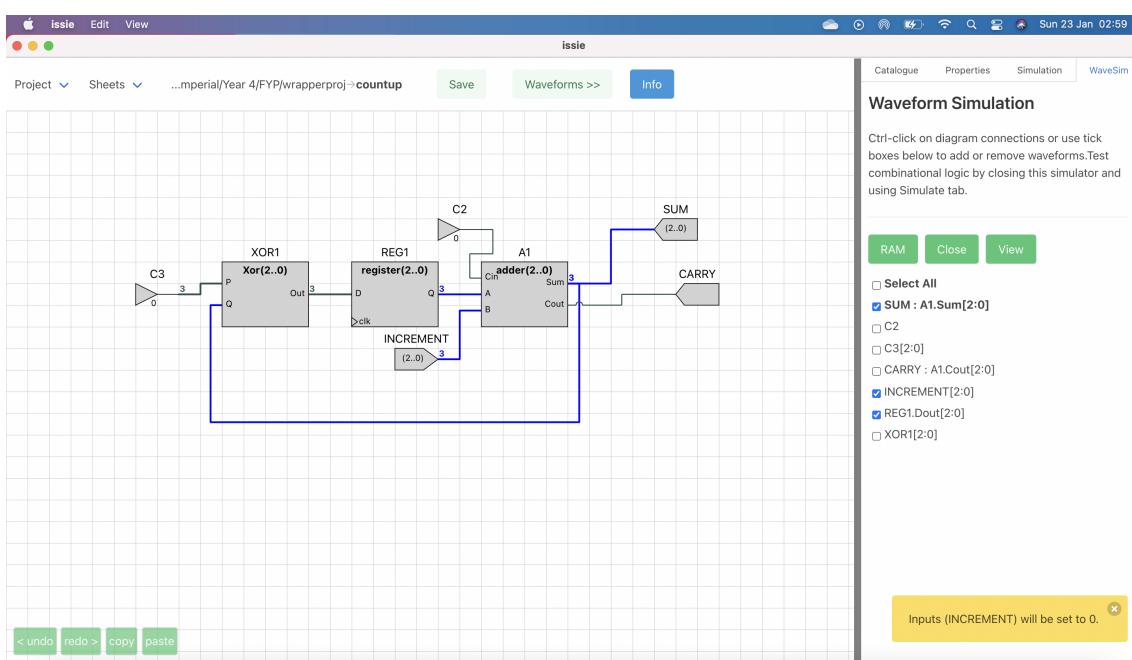


Figure A.3: Issie's Waveform Simulator selection menu

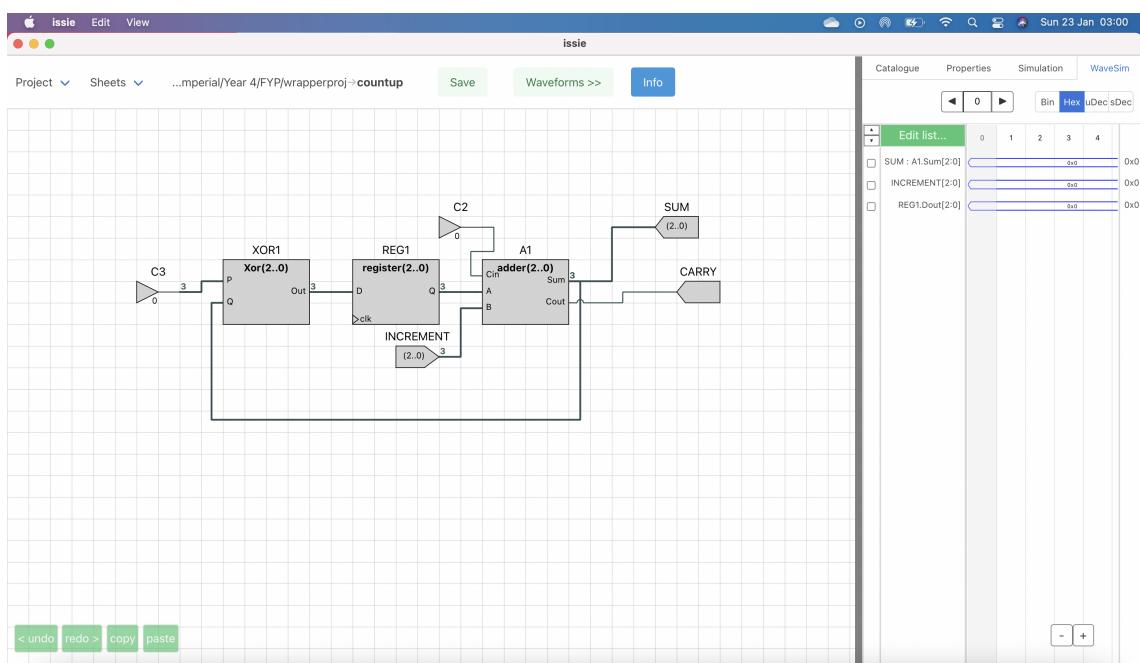
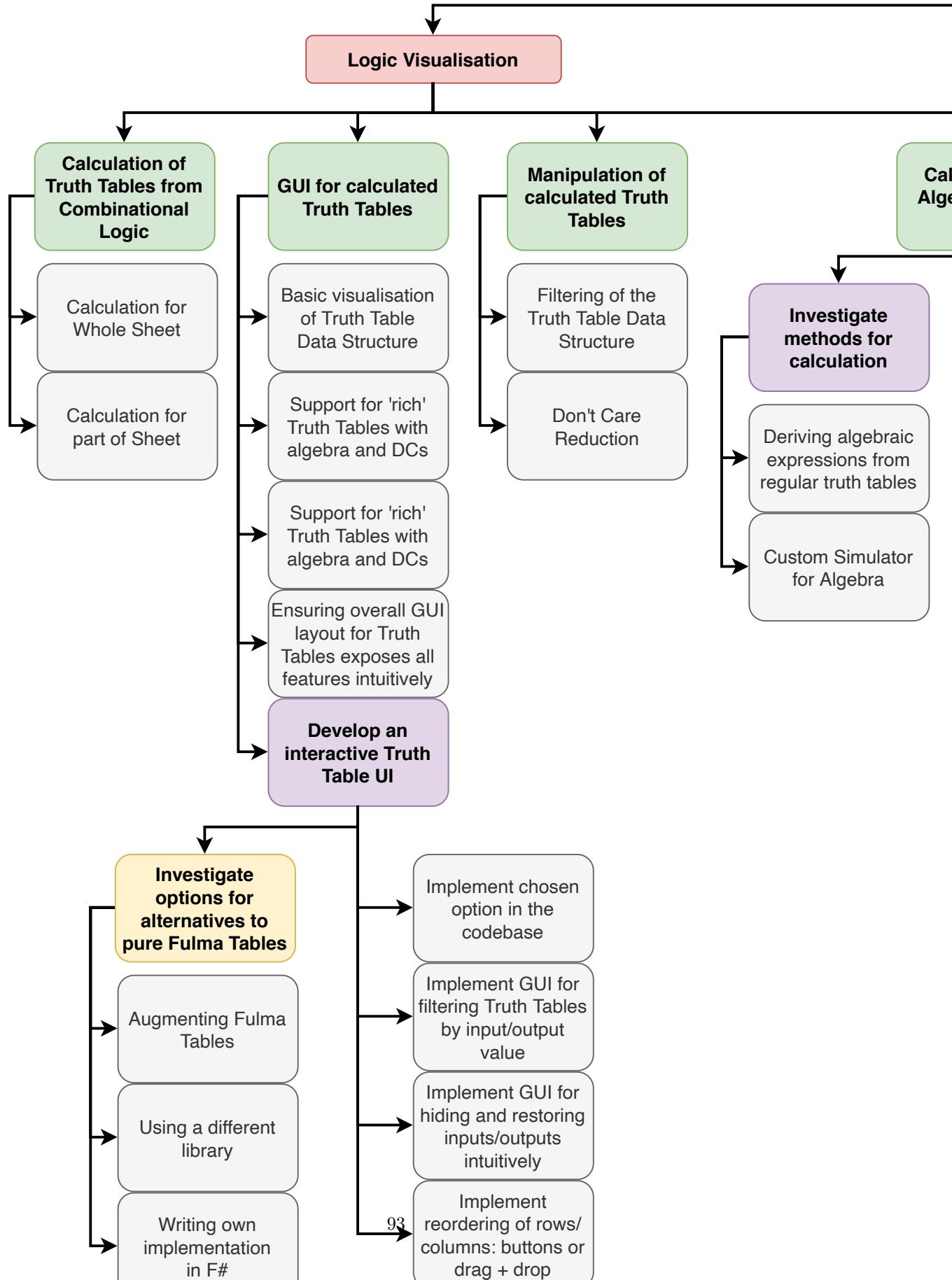


Figure A.4: Issie's Waveform Simulator

Appendix B

Work Breakdown Structure



Appendix C

8-bit ALU Schematic

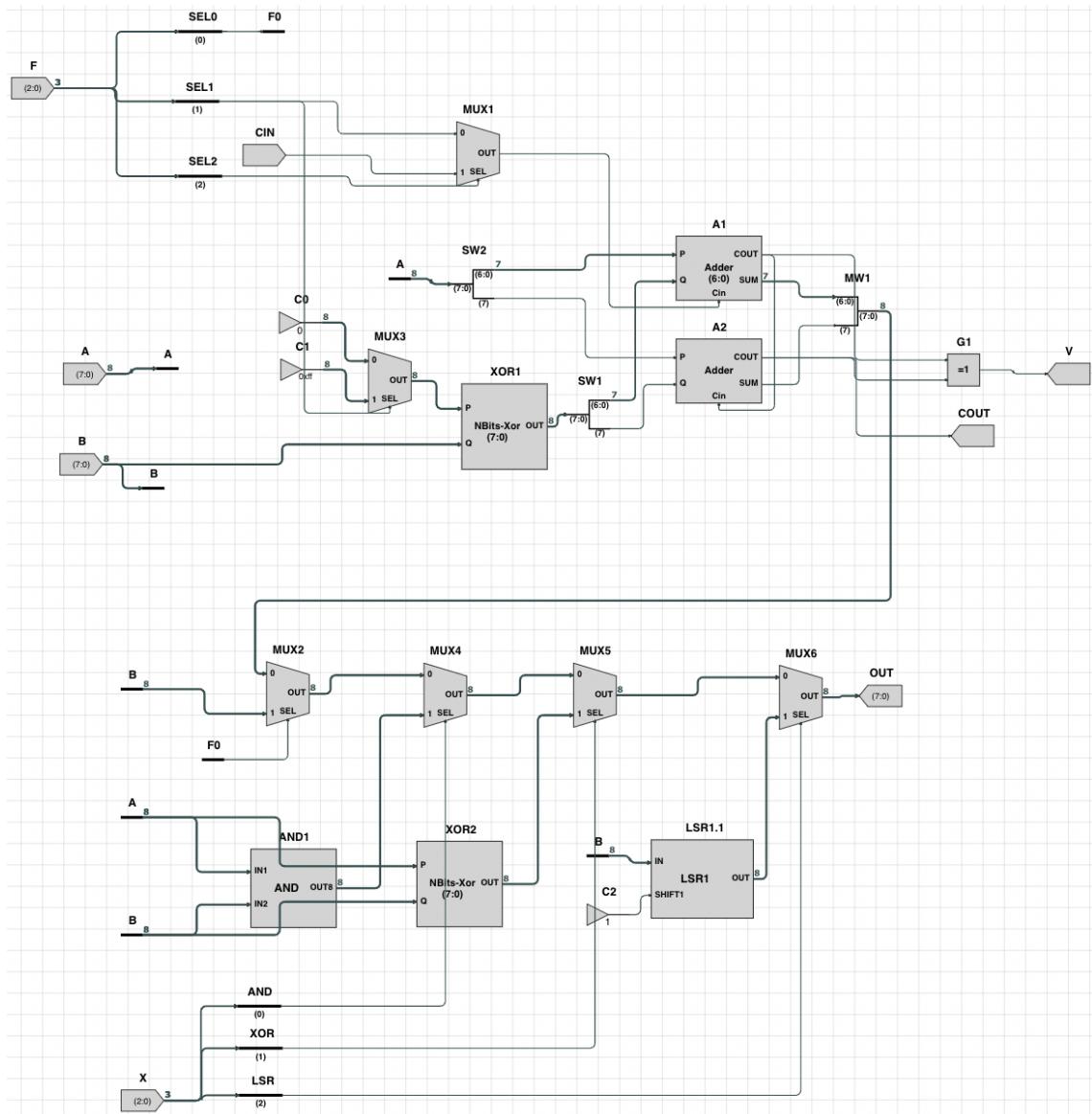


Figure C.1: Schematic Diagram for an 8-bit ARM ALU