Imperial College London

MEng Individual Project

Imperial College London

Department of Electrical and Electronic Engineering

A Verilog Compiler for Issie

Author:
Petra Ratkai

Supervisor:
Dr Thomas Clarke

Second Marker:
Dr Christos Papavassiliou

June 21, 2023

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.

I have used ChatGPT, as an aid in the preparation of my report to improve the style of certain sentences as well as to generate some test cases used for evaluation, see section 7.1.

Acknowledgements

First and foremost, I would like to extend my heartfelt appreciation to my supervisor, Dr. Thomas Clarke, for granting me the opportunity to work on this highly rewarding project and for his invaluable assistance and guidance throughout the academic year.

I would also like to express my gratitude to Yujie Wang for providing assistance in resolving numerous build issues that I encountered during the project.

Furthermore, I am deeply grateful to my friends, whose companionship has made the past four years truly remarkable. I would also like to acknowledge the unwavering support of my parents, who have stood by me throughout these years and enabled me to pursue my education at one of the world's top universities.

Abstract

This paper presents the development and enhancement of the Issie software, an interactive schematic simulator and integrated editor. Issie adheres to a set of principles aimed at providing a user-friendly and intuitive experience for both novice and experienced users. This project delivers an extensible SystemVerilog compiler and synthesis system for Issie, with extensive testing, and high quality error messages, fully integrated into the Issie app. The report delves into the design decisions made during the project, highlighting the chosen language features and additional language restrictions imposed as well as the detailed implementation of the context-free parser, the context-sensitive semantic error handler and hardware generation process. The compiler supports behavioural and structural SystemVerilog features, including combinational and clocked always blocks, blocking and nonblocking assignments, conditional and case statements together with module instantiation statements. Through this project, Issie's SystemVerilog compiler has been enhanced, providing users with an improved tool for designing and simulating hardware circuits.

Contents

Chapter 1

Introduction

The Interactive Schematic Simulator and Integrated Editor (ISSIE) [1] is a digital circuit design and simulator application. As Issie is developed for students who want to learn about digital electronics, the application must be intuitive to use with error messages that are easy to understand [1]. Issie is useful for creating hierarchical designs from schematic components with the application's outstanding user interface, and at the moment it also provides basic support for writing modules in Verilog. Since the software is used for university teaching (Digital Electronics for first year students at Imperial College London) it would be very useful to have extensive support for Verilog so that students can have experience with hardware description languages from very early on and to allow them to create complex circuits quickly.

Issie aims to distinguish itself from other simulators by adhering to the following principles:

P1 No need for manual or training to use it

P2 Mistakes should be explained by Issie in a way that makes them easy to correct

P3 The UI should provide visual prompts for all desired user operations

P4 Operation should be obvious: there should not be unexpected hidden state that affects what happens

P5 Easy use by complete novice

P6 Efficient use by experienced individuals

Issie currently has a proof-of-concept Verilog compiler using a Nearley [2] parser. This supports simple concurrent assignments for combinational logic. The main goal of this project is to extend this compiler and allow it to handle clocked logic as well as behavioural and structural Verilog. By adding these features, Issie could be used to not only teach Digital Electronics for first year students but also second and potentially third and fourth year undergraduate students as there are numerous modules in the Department of Electrical and Electronic Engineering at the university where students have to implement Verilog projects. Although there are various Verilog (and other HDL) simulators and synthesizers that are suitable for use by beginners, such as Icarus Verilog [3], a lot of them are pure command line tools. For example a third party software, GTKWave [4] is needed to display the wave-forms generated by Icarus Verilog[5], possibly making it difficult for a beginner to learn digital electronics concepts. Other simulators that provide a graphical interface, such as Quartus [6], are often targeted at experienced hardware engineers. With an extended SystemVerilog compiler, Issie would allow the end users to easily design and simulate complex digital circuits described by SystemVerilog (or a mix of schematic diagrams and SystemVerilog components) using a single application.

This report first presents the necessary background material needed for the project in chapter 2 by introducing the Issie technology stack, the most important concepts in SystemVerilog as well as concepts in the field of compilers along with various parsing methods. Chapter 3 discusses the high level overview of the required and desirable features of the project. The high level decisions and implementation choices made during the planning and development of the compiler can be found in chapter 4. Chapter 6 quantitatively evaluates the correctness of the compiler whereas chapter 7 qualitatively evaluates the final product focusing on the quality of the error messages, the

compilation performance and the maintainability of the code base, and it compares the outcomes of the project with the requirement capture. Finally, chapter 8 reflects on the work done and highlights the most challenging tasks completed. Section 8.1 explores additional features that could be implemented in the future.

Chapter 2

Background

2.1 Issie technology stack

The compiler is built on top of the existing Issie software, so it is important to have an insight into the Issie infrastructure and the advantages of using the given frameworks, tools and programming languages to implement the compiler.

Issie is a 36K line cross-platform application mainly written in F# [7]. Issie runs on top of the Electron [8] software framework, which embeds Node.js [9] and Chromium [10], hence enabling the developers to use web technologies to create desktop apps [8]. To run the F# files, these are converted to JavaScript files using the Fable [11] transpiler, which allows the application to run in the Chromium browser. To make sure that Issie works on all major operating systems, the .NET [12] framework and the NPM [13] package manager software are used.

2.1.1 F#

F# is an open source cross-platform [7] functional programming language. F# allows developers to write succinct, robust and highly performant code [14]. F# - as many other functional programming languages – is concise: its minimalistic and streamlined syntax is devoid of unnecessary elements such as curly brackets and semicolons, thus reducing the level of "noise" present [15]. The strongly typed nature and the expressiveness of the language allows the developer to write correct code easily and spend less time on debugging and more on the implementation, which makes it a great language to use for large projects such as the Issie Verilog compiler. Another important feature of F# is that objects are immutable by default [14] which is particularly great for developing a compiler where the Abstract Syntax Tree (AST) is a constant data structure. Having an immutable AST avoids the problem of accidentally mutating the data.

2.1.2 Fable

Fable is a general-purpose F# to JavaScript compiler. Using Fable, code written in F# can run anywhere JavaScript runs [11], which is how the Issie F# project can run inside the Chromium browser. The main advantage of using Fable is that the developers can leverage the benefits of the F# language discussed above as well as the speed of JavaScript programs.

2.1.3 Elmish MVU

Issie comprises more than 60 modules which implement a single component Elmish Model-View-Update (MVU) [16] application (see Figure 2.1) hence the Verilog compiler also has to follow this form. This pattern contains the state of the application in a single Model type, and it separates the internal state changes from the UI controls [17] which enables the developer to focus on the internal logic of the software rather than the UI and front-end parts. In this architecture Messages represent the events that can change the state of the Model, for example the user interface can trigger Messages. The Update function returns the next state based on the triggered Message and the current state, and the Render or View function builds the user interface from the current state [18]. The MVU is advantageous, since the separation of the UI and the state makes the code easy to maintain and test, and it also simplifies the debugging process since the UI and the internal logic

can be tested independently. This pattern makes the state of the application clear and predictable, which reduces the possibilities for errors. On the other hand, in real world application, the MVU gets complicated as the size of messages and the length of the update function get large [19].

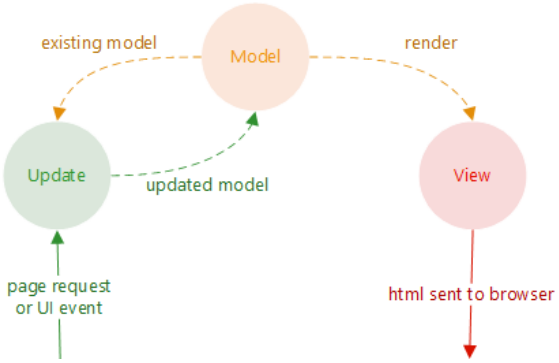


Figure 2.1: Model-View-Update pattern [17]

2.2 Verilog and SystemVerilog overview

Verilog is a hardware description language (HDL) that is used to model the behaviour of hardware and SystemVerilog is an extension of it which comes with many useful features compared to Verilog [20]. Verilog is used to simulate, test and verify digital electronic designs [21] and it provides a means to describe the interconnection and hierarchy of various components in a digital system. The semantics and syntax of Verilog form important background to this project, and so are reviewed below.

2.2.1 Data types

The SystemVerilog value set consists of the following basic logical values [22]:

- 0 – represents logic 0 or false condition
- 1 – represents logic 1 or a true condition
- x – represents an unknown logic value
- z – high impedance state

As Issie allows only two states (0 or 1), the focus was on these two states when designing the compiler (see section 4.2).

The two main groups of data objects are nets and variables. Nets are used to connect different hardware entities [23], they represent wires and they do not store their value. Nets can be written by one or more continuous assignments [22] (described in subsection 2.2.3). The most important net data object is `wire`

Variables on the other hand can be driven by procedural statements (see subsection 2.2.4) or by a single continuous assignment. The most important data types are `reg` and the SystemVerilog `logic` [24]. The issue with using the `reg` type is that beginners often assume that it always represents a piece of storage [25], which is not true, as it can also represent a wire if it is driven by combinational logic.

All `wire`, `reg` and `logic` are by default unsigned, 1 bit wide (scalar) [26] and they are all 4-state data types. The `bit` data type available in SystemVerilog is the 2-state equivalent of `logic`. These data objects can take the `signed` modifier, and they can also be declared as vectors to represent buses.

2.2.2 Expression widths

SystemVerilog determines the width of an expression based on the bit lengths of the operands and by the context of the expression [22]. Self-determined expressions are expressions whose bit length

is solely dependent on the expression itself, whereas context determined expressions are expressions where the length of the expression is dependent on the bit length of the expression itself as well as by the fact that it is part of another expression [22]. The operand widths of a self-determined expression are never extended [27]. The steps for evaluating an expression are the following [22]:

1. Determine the right-hand side expression (and every subexpression) bit length based on the standard rules of expression size determination
2. Determine the signedness of the right hand side expression
3. Propagate the size and type of the expression down to the context-determined nodes. Any context determined operand should be the same size and type as the result of the operator.
4. An operand should be sign extended only if it is signed.

When evaluating an assignment, the width of the left-hand side of the assignment is part of the context, so the size of the expression will be the maximum of the left hand side width and the expression width, and this width will be propagated down to the context determined nodes. See Appendix A for expression bit lengths of the different operators.

2.2.3 Combinational logic – continuous assignments

Continuous assignments change the value of the left-hand-side variable (scalar or vector) whenever the value of the variables in the right-hand-side logical expression changes [28], hence, Verilog can describe complex logical circuits with a simple textual representation. An example for continuous assignments can be seen in Listing 2.1.

```
wire x, y, z;
assign x = y | z;
```

Listing 2.1: Continuous assignment for an OR gate

Verilog also supports net declaration statements or implicit continuous assignments. Implicit continuous assignments allow the assignment to be done when the net is declared [29] shown in Listing 2.2.

```
wire y, z;
wire x = y | z;
```

Listing 2.2: Implicit continuous assignment for an OR gate

2.2.4 Sequential blocks and procedural statements

Verilog also supports procedural blocks in which the statements are executed sequentially. Procedural blocks include the **initial**, **always** and **final** blocks. Note that the **initial** and **final** blocks are non-synthesizable constructs [30]. The general syntax for always blocks can be found in Listing 2.3.

```
// Syntax for always blocks
always @(event) //sensitivity list
    {statement}
```

Listing 2.3: Always block syntax

The always block can also include multiple statements, in this case the statements must be enclosed by the **begin** and **end** keywords. Always blocks are executed at an event specified by the sensitivity list [31].

Always blocks can be combinational or sequential (clocked). Sequential always blocks that are given a clock signal generally behave as flip-flops: for example, they become active on the positive edge of the clock. Combinational always statements are similar to the continuous assignments: whenever a variable in the inputs changes, the combinational always block becomes active.

The always block can have blocking or non-blocking assignments. In the case of blocking assignments, the right hand side expression is evaluated, and it is assigned to the left hand side variable and the next line of code will be executed after this. When there is a non-blocking assignment, the right hand side expression is evaluated first and the left hand side variable will

only be updated at the end of the always block. In continuous assignments, the "wire" data type can be used for the left-hand side, whereas in always blocks the "reg" type must be used [32].

In general, mixing blocking and non-blocking assignments should be avoided because when they are mixed, it is up to the synthesis tool to decide how they will be synthesised [33], hence combinational and sequential logic should be separated. In combinational always blocks one should use blocking assignments and in clocked always blocks non-blocking assignments should be used as shown in Listing 2.4 and Listing 2.5.

```
module m1 ( input a,
            input b,
            output reg c );
always @(*) begin //sensitivity list - any change on the RHS
    c = a | b;
end
```

Listing 2.4: Combinational always block with blocking assignment

```
module m1 ( input clk,
            input rst,
            output reg counter );
always @(posedge clk) begin //sensitivity list - any change on the RHS
    if(!rst)
        counter <= 0;
    else
        counter <= counter + 1;
end
```

Listing 2.5: Clocked always block with non-blocking assignments

Always blocks can also contain case statements and conditional statements. SystemVerilog also provides `always_comb` and `always_ff` blocks. The `always_comb` block is a much improved version of `always @*`, which makes it much safer to use than the plain always construct and hence `always_comb` is always preferred to `always @*`. The plain `always @*` can cause unexpected behaviour, for example with function calls as seen in Listing 2.6 [34]. The `always_ff` construct is used to model sequential flip-flop behaviour. The `always_ff` procedure only allows exactly one event control in the sensitivity list[34]. `Always_comb` and `always_ff` also report an error if any of their left hand side variables are written to in any other always blocks [34].

```
// unexpected behaviour with plain always
module test;
  logic a, b, c, always_d, always_comb_d;

  function logic my_func(input logic m_c);
    my_func = a | b | m_c;
  endfunction

  always @* // this will only trigger when the value of c changes and not when
    a and b change
    always_d = my_func(c);

  always_comb
    always_comb_d = my_func(c); // this will trigger if any of a, b and c
    changes

  initial begin
    $monitor("@%0t: a = %d, b = %d, c = %d, always_d = %d, always_com_d = %d",
    $time, a, b, c, always_d, always_comb_d);
  end

  initial begin
    a = 0;
    b = 0;
    c = 0;
    #10 a = 1;
    #10 b = 1;
    #10 c = 1;
  end
endmodule
```

Listing 2.6: Unexpected behaviour with Verilog always [34]

2.2.5 Structural Verilog

Structural Verilog connects different blocks of code (modules) and hence achieves a hierarchy in the design[35]. In structural Verilog the hardware design is split up into modules or components. Each module has a header which describes the interface of the module, including input and output ports. The body of the module provides the implementation of the module behaviour [36]. Modules can be instantiated in other modules using module instantiation statements [37] that specify how the child component connects to the wires and registers of the parent component's variables.

The port mapping in module instantiation statements can be named or ordered, as illustrated in Listing 2.7 [38].

```
module DFF (Q, D, CLK);
  input D, CLK;
  output reg Q;
  always @ (posedge CLK)
    Q <= D;
endmodule

module SYNCHRO1 (ASYNC,SYNC,CLOCK);
  input ASYNC;
  input CLOCK;
  output SYNC;
  wire C1_ASYNC;
  // ordered port mapping:
  DFF DFF1 (C1_ASYNC, ASYNC, CLOCK);
  DFF DFF2 (SYNC, C1_ASYNC, CLOCK);
endmodule

module SYNCHRO2 (ASYNC, SYNC, CLOCK);
  input ASYNC;
  input CLOCK;
  output SYNC;
  wire C1_ASYNC;
  // named port mapping:
  DFF DFF1 (.D (ASYNC), .CLK (CLOCK), .Q (C1_ASYNC));
  DFF DFF2 (.D (C1_ASYNC), .Q (SYNC), .CLK (CLOCK));
endmodule
```

Listing 2.7: Named and ordered module instantiation statements [38]

2.2.6 Verilog design principles

The Verilog language grants the user a lot of freedom. To avoid the numerous pitfalls of the language it is advisable to adhere to the following best practices based on Stuart Sutherland's Standard Gotchas - Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know [39]:

- Separate combinational and sequential logic: in clocked always only use non-blocking assignments, in combinational always only use blocking assignments. If there is a blocking assignment in a clocked always block, it can synthesize to a flip flop or a wire, depending on the synthesizer [33]
- In SystemVerilog only use `always_ff` or `always_comb` instead of `always`: the `@*` wildcard only infers inputs that are directly referenced in the always block
- When an array is needed in the sensitivity list, `always_comb` or `always @*` should be used
- Only single bit items should be used with `posedge` or `negedge`
- In the always sensitivity list "or" and "|" are different, again need to use `always_comb`
- Designers need to be sure to assign a default value to each variable in if-else statements and case statements, otherwise the synthesized hardware might contain latches
- Nested if-else statements might need begin and end for correct behaviour - parser / editor could potentially help with the indentation

- Do not have multiple procedural blocks writing to the same variable, `always_comb` and `always_ff` will give an error
- Do not have any duplicate case items in a case statement
- Modularised designs are good
- Use the logic data type to declare all point to point nets (wires or busses), variables driven by always blocks, all input ports and all output ports [40]
- Use `always_comb` rather than `always_comb @(*)` [41]
- Do not use latches, or code that can synthesize to an unexpected latch (it can cause timing issues) [42]

As Verilog allows developers to compose sub-optimal or unsynthesizable code, the Verilog compiler project provides a great opportunity to only support a restricted subset of the language by mandating the user to adhere to the Verilog best practices. Conforming to these concepts makes sure that the compiler is aligned with the Issie principles and would make it easier for the students to learn the concepts of hardware design even without prior knowledge and would teach them good design principles very early on. Section 4.1 details how some of these best practices apply to the project.

2.3 Parsing

A large part of compiler design, and the one which determines some of the error messaging, is the parser. This section reviews parsing technology and discusses the techniques most suitable for this project.

2.3.1 Terminology: languages and grammars

A language in computer science is a set of sentences of finite length constructed from a finite alphabet [43] and it can be described by a formal grammar. A grammar is a set of production rules through which the sentences in the language can be generated [44]. The grammar is composed of terminal symbols - symbols contained in the sentence generated by the grammar - and nonterminal symbols which take part in the generation of the sentence but are not part of the generated sentence [45]. Production rules are rewrite rules that define how a nonterminal symbol can be rewritten as a string of terminal and nonterminal symbols. Production rules are often of the form $\alpha \rightarrow \beta$ which specifies that the nonterminal α can be rewritten as β.

There are four types of grammars, from the narrowest set to the widest these are regular, context-free, context-sensitive and recursively enumerable grammars [46], see Figure 2.2. As defining most arithmetic expressions requires recursive context-free grammars, programming languages in general – including Verilog – are also defined by recursive context-free grammars. Note that for semantic error detection and synthesis, the compiler requires the context of the entire SystemVerilog file, hence for these processes context-sensitive logic is needed.

2.3.2 Compilers

A compiler is a program that translates computer code written in one language into an equivalent program in another language, for example machine code, byte code or a different programming language [47]. The different phases of a compiler are lexical analysis, syntactic analysis or parsing, semantic analysis, intermediate code generation, optimisation and code generation [48]. The flow diagram of these phases can be seen on Figure 2.3

During the lexical analysis, the lexer transforms the source code (stream of characters) into a stream of tokens[50]. The tokens in the lexer are defined by regular expression. If there are any errors at the stage of the lexical analysis, a good compiler must give the user meaningful error messages. According to the Compilers: Principles, Techniques, and Tools [48] using a lexer has the following advantages:

1. Separate lexical and syntactical analysis allows for simpler design and the compiler will be more maintainable

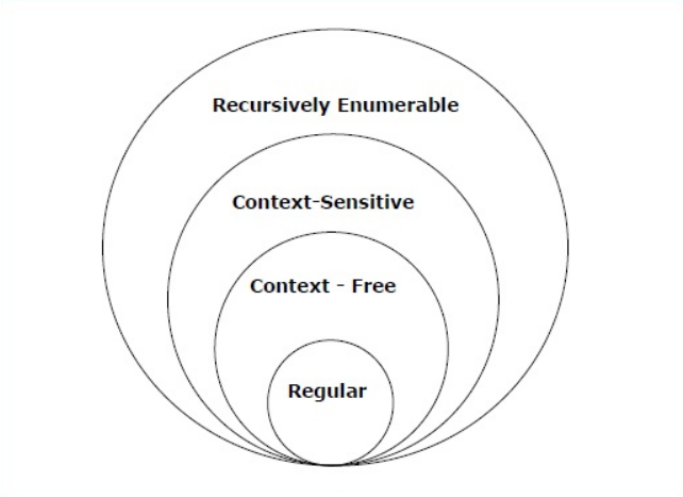


Figure 2.2: The Chomsky hierarchy of grammars

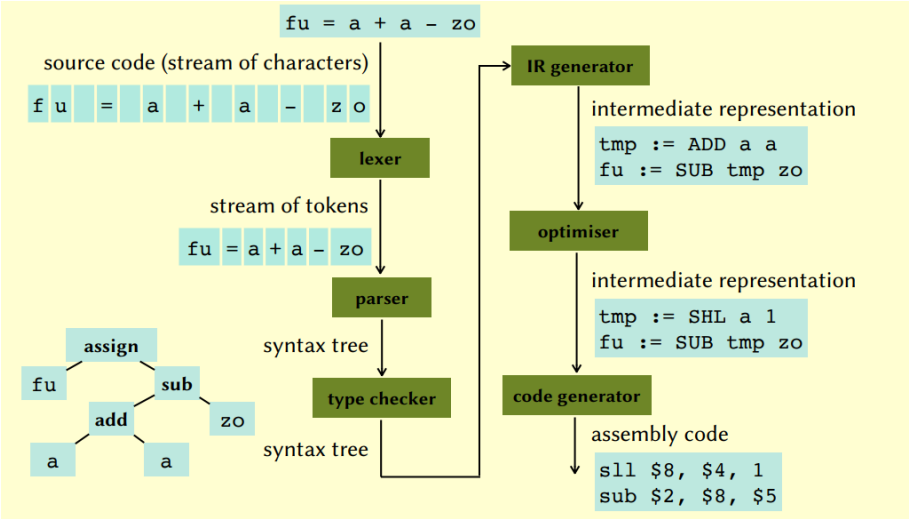


Figure 2.3: Phases of a compiler [49]

2. Improved compiler efficiency

3. Improved compiler portability

After the lexical analysis, the parser takes in the tokens generated by the lexer and turns them into a tree-like grammatical structure as an abstract syntax tree (AST). The parser describes the source language using a grammar made up of production rules which use terminals and non-terminals. Production rules define a structure (left-hand side) from a pattern of terminal and non-terminal symbols [51]. Terminal symbols are the most elementary symbols in the grammar, they only appear on the right-hand side of the production rules and non-terminal symbols are defined using other symbols. A production rule could look like the following: `expr ::= expr + term | expr - term | term` . This rule defines `expr` as either `expr + term`, or `expr - term` or just a `term`. Similarly to the lexical analysis, the generation of understandable error messages is crucial in this stage too.

Once the parser generates the AST, the intermediate code generator outputs an intermediate representation of the code, then the compiler performs machine independent optimisation and finally the code generator outputs the resulting code in the target language.

In the case of the Issie Verilog compiler, the source language is Verilog (with some SystemVerilog features) and the target language is an Issie sheet comprised of Issie components. The emphasis of this project is on developing the parser and handling syntactic and semantic errors well. For good user experience it is important that the simulations are fast, hence one of the extensions of

the project is adding optimisation which will include simplifying the AST for a given program so that the simulator would need to evaluate fewer gates, as well as potentially investigating the use of an intermediate language that the simulator can run or developing a separate fast simulator for Verilog inputs.

2.3.3 Parsers overview

One aspect of the project is selecting the right parser, hence the trade-offs of different parsers were considered. Different parsers were compared based on qualities such as the supported set of grammars and time complexity.

The parsing methods commonly used by compilers are top-down and bottom-up parsing [48]. Top-down parsers start looking at the highest level of the parse tree and work down from there to the leaf nodes using the production rules of the grammar, whereas bottom-up parsers start at the leaf nodes and work up the parse tree by using the production rules of the grammar [52]. In both cases, the input string is parsed from left to right, one symbol at a time [48]. The hierarchy of different types of compilers can be seen on Figure 2.4.

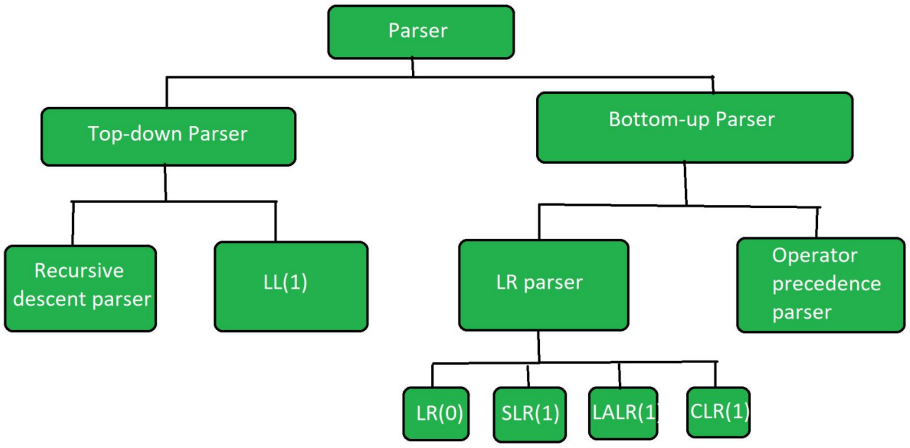


Figure 2.4: Hierarchy of parsers [53]

Top-down parsers

Top-down parsers use leftmost derivation [54], which means that the production rule is applied to the leftmost variable at every step [55]. It is important to note that top-down parsers cannot handle left recursion and ambiguity [56] hence left recursive grammars need to be transformed. Top-down parsers are for example recursive descent parsers and LL parsers.

Bottom-up parsers

The bottom-up parsing technique uses rightmost derivation [54] (the production rule is applied to the rightmost variable at every step [55]). Bottom-up parsing can be done using backtracking, but usually it is done by a shift reduce parser.

Shift Reduce parsers structure the input string by reducing it to the starting symbol of the grammar [52]. In bottom-up parsing, at all times the parser has a sequential form of the input string that is obtained through a series of reductions [57]. This sequential form is split up into the already parsed symbols – the stack, which can be a mix of terminals and non-terminals – and the rest of the input that has not been parsed, as seen in Figure 2.5. At every step of parsing, the parser can either shift the input or do a reduction on the stack [57], which can be seen in Figure 2.6 and Figure 2.7 where the triangles represent the partial AST structures. When the parser encounters a syntax error it will report an error back and if there are no more reductions the parse of the input is returned. Simple shift reduce parsers cannot handle some context-free grammars: for some constructs, the parser cannot decide if it should do a reduce or a shift step (shift/reduce conflict) or it can't decide which possible reduction to do (reduce/reduce conflict) [48].

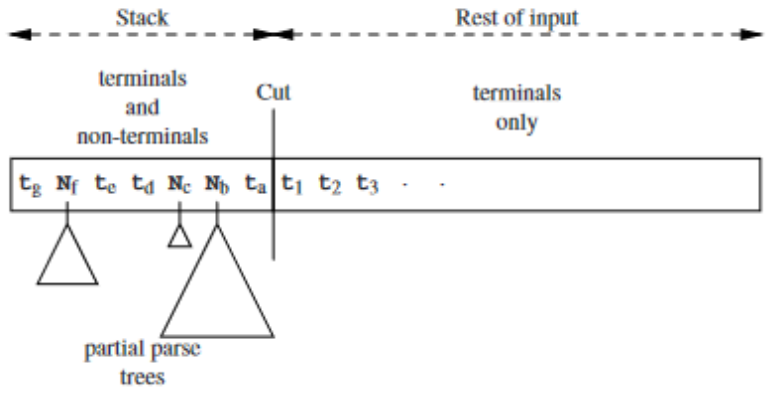


Figure 2.5: Structure of shift reduce parsing [57]

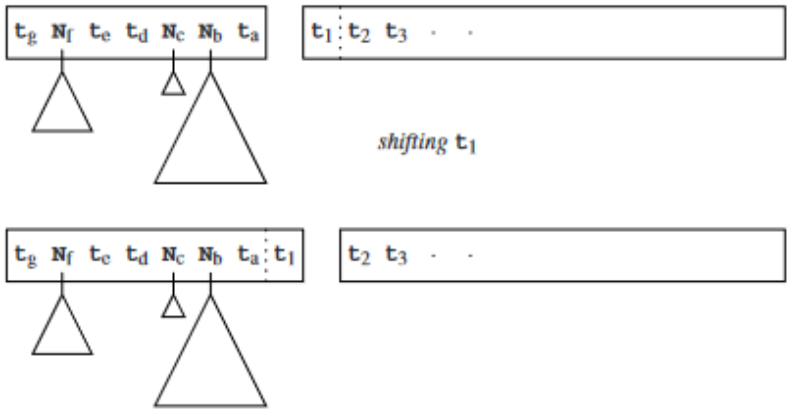


Figure 2.6: Shift reduce parsing shift move [57]

2.3.4 Operator precedence parsers

Operator precedence parsers are simple shift reduce parsers that can parse an operator precedence grammars, which is a subset of the LR(1) grammars. Operator precedence grammars are grammars in which the right-hand side of any production rule is not empty and contains no adjacent non-terminal symbols [58]. These parsers can be fast when there are a lot of operator precedence levels for binary expressions [59]. However, grammars that are not operator precedence grammars must be transformed, which would make the implementation of the Verilog parser more complex.

2.3.5 LR parsers

LR(k) parsing (**L**eft to right, **R**ightmost derivation first) is a type of bottom-up shift reduce parsing, where k stands for the number of symbols lookahead needed to make the parse decisions. LR parsers use a stack, an input buffer and an LR parsing table. In all LR parsers, the stack, the input buffer and the parsing algorithm are the same, the only difference is in the parsing table.

LR parsers are very agreeable since they are not recursive, they do not use backtracking (which would be computationally expensive), they can parse most programming constructs, and they can recognise the ambiguities of the grammar [60]. Furthermore, the grammars that can be handled by LR parsers is a proper superset of the grammars that LL parsers can handle, i.e. LR parsers can handle a wider range of grammars than LL parsers [48]. LR parsers can also accept left recursion [61] unlike most top-down parsers, and they process deterministic context-free languages in linear time in the length of the input string [62].The main disadvantage of LR parsers is that they are too much work to construct by hand, so a parser generator is needed (for example `yacc`) [48].

LR(0) parsers are a more advanced version of the shift reduce parser but the simplest of the LR

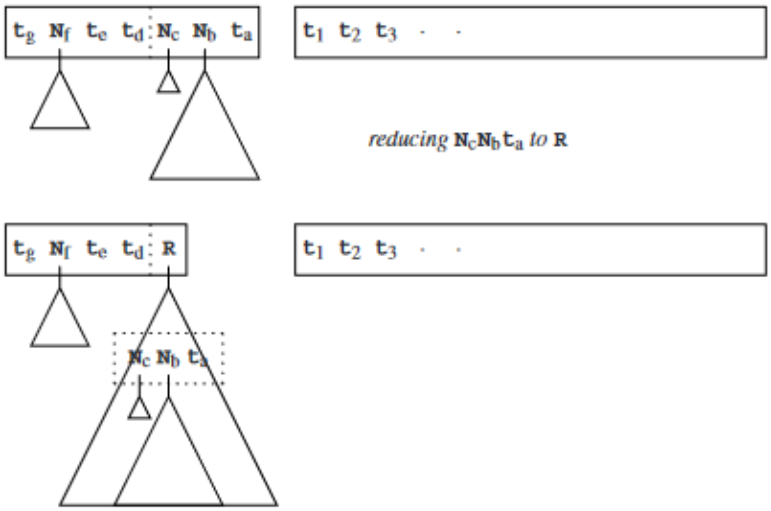


Figure 2.7: Shift reduce parsing reduce move [57]

parsers. LR parsers have an internal finite state automaton, which keeps track of all information about the current parse, including the next possible production rules to use and production rules in progress. Based on the state machine, a parse table is generated, from which the parser can decide if a shift or a reduce statement should be done next. Instead of shifting the input as the shift reduce parser, the LR parser shifts the state [48]. By keeping track of more possibilities as part of the state machine, the LR parser can avoid some shift/reduce conflicts [48]. However, LR(0) parsers are too weak, they can only handle a small set of grammars [63].

SLR(1)

Simple LR or SLR(1) parsers improve the problem of shift/reduce conflicts in LR(0) by maintaining a follow set for every production rule which contains the next possible symbol for that particular rule. The SLR(1) parser only does a reduce step if the next input token is an element of the follow set of the reduction rule [63]. This means that the SLR(1) can handle a wider group of grammars than the LR(0), since this improved parser can describe some grammars that would have caused a shift/reduce conflict in the LR(0) parser. The SLR(1) parser has very few states, and hence it is fast to construct [64]. Although SLR(1) is a much improved version of the LR(0) parser, the SLR(1) parser does not utilise all the information it has – this parser only stores the information about the next input elements and not what is below the next element on the stack [63] hence the SLR(1) parser still only works on a small class of grammars [64]. This is what the LR(1) parser improves on.

CLR

Canonical LR parser or LR(1) is a more advanced and more complex parser than the LR(0) and SLR(1) parsers. In LR(1) parsers, there is a terminal symbol lookahead for 1 symbol. This extra information is stored in the states of the automaton: every production rule item has the next possible symbol. However, this requires splitting up each state, resulting in a large number of states in the finite automaton. With the one symbol lookahead, the LR(1) parser can make parse decisions in a lot of cases where SLR(1) cannot, which makes the LR(1) a more powerful parser [63]. On the other hand since the LR(1) parser splits up the states based on the lookahead symbols hence the number of states in this parser can be an order of magnitude higher than in the case of SLR(1) parsers [65] which makes the parser very slow to construct [64].

LALR(1)

The LALR (Look Ahead LR) parser merges similar states in order to reduce the number of states in the finite automaton [65]. LALR(1) grammars are a subset of LR(1) and a superset of SLR(1)

grammars [65] which means that the parser is weaker than the LR(1) parser but with a much smaller state space, and it is more powerful than the SLR(1) parser, they work on the entire class of LR(1) grammars [64] but they still cannot handle the entire class of context-free grammars.

2.3.6 Earley

The Earley parser is a type of generalised parser [48] which can parse any context free grammars, including ambiguous ones and left recursion [66]. Generalised parsers also report at runtime where all the ambiguous points are in the grammar, so the input won't be parsed incorrectly in silence [67]. However, this also means that one needs to find an ambiguously parsed input to find out that the grammar is ambiguous [67]. The biggest advantage of the Earley parser is that it can process a large group of grammars and so the developer does not need to adhere to restrictions in the grammar, which makes it easier to write the grammar for Earley parsers compared to other compiler tools [68]. To handle all context free grammars and to potentially return all possible parses of a grammar has a cost in the speed: the worst case time complexity of this parser is cubic in the length of the string and for unambiguous grammars it is quadratic [68]. In the case of the Issie Verilog compiler this complexity will not be a problem since the size of Verilog programs is unlikely to be very long so the time taken to parse the input is not going to be significant. Most Verilog modules written in Issie are expected to be up to around a 100 lines long, however, if multiple modules can be instantiated in the same file, the block sizes could reach up to 1000 lines. For the detailed performance evaluation of the compiler, see section 7.2.

2.3.7 Nearley

Nearley is a JavaScript parser that uses the Earley algorithm and is highly optimised for LL(k) grammars [2]. Nearley can successfully parse any context-free grammars, including left recursion and ambiguous grammars. One of the biggest advantages of the Nearley parser is the error handling: it provides detailed error messages which makes debugging easier [2] and it makes it easier for the developer to also provide good and detailed error messages to the user. Note that Nearley is a scannerless parser [69], meaning that it does not require the use of a lexer, it simply splits the input into a stream of characters. The Nearley parser is a suitable choice for the Issie Verilog compiler because it can handle any context free grammars, so writing the Verilog grammar is simple: there is no need to refactor the grammar, the standard Verilog grammar can be used and hence it will make the development quicker. The detailed error handling is also an advantage since it makes it easier to conform to the Issie principles. As mentioned earlier, the Earley algorithm and hence the Nearley algorithm has a worse time performance than LR parsers, but for the expected input size this will not be a problem. Furthermore, Issie already uses Nearley to parse simple Verilog files, which means that the continuation of using this parser will substantially reduce the development effort required.

2.3.8 Conclusion

There are many parsers that the Verilog compiler could use. Operator precedence parsers and LR parsers only support a subset of context free grammars, which means that constructing the production rules of the parser would be a complex task, since some production rules would have to be transformed such that the grammar falls into the supported set of grammars. In contrast, Nearley supports all context-free grammars, which means that the Verilog grammar does not need to be changed much from the official Verilog grammar, which will save many development hours. LR parsers such as yacc are very fast, they provide linear time complexity in the length of the input string whereas the Earley's algorithm and hence the Nearley parser have a worse performance, quadratic time complexity for the Verilog grammar. However, this trade-off in performance is not expected to be an issue since the Issie Verilog programs are not expected to be long. Finally, the Nearley parser is used in the current Verilog compiler which means that if a different parser were chosen, the grammar would have to be rewritten for the already supported features, creating more development hours. Based on these properties of the parsers, the most suitable parser for the Issie Verilog compiler is Nearley. Choosing Nearley for the Verilog compiler allows for a clean implementation and fewer development hours spent on implementing the grammar, enabling the development to focus on perfecting the new features and the error handling.

2.4 The legacy compiler

This section assesses the features and limitations of the legacy compiler which had been implemented in Issie before the start of the project.

2.4.1 Features

The original Verilog compiler in Issie handles simple Verilog modules. This includes the following features:

- Input and output port declarations – both old and new (ANSI C) style
- Continuous assignments
- Wire declarations with continuous assignment
- Most combinational logic operators
- Constant indexed bit select
- Bus slices

Along with these features, the compiler generates useful and detailed error messages in the case of malformed inputs. The errors appear in the form of the incorrect token being underlined and the error message appearing on hover. Furthermore, the original codebase also implements a table for extra error messages – these are more detailed error messages that occasionally also contain suggestions for corrections.

2.4.2 Limitations

Even though the base compiler works correctly for most inputs users are expected to write, the compiler also exhibits some limitations:

1. Cycles are avoided by not allowing output ports on the right-hand side of assignments and wires being assigned to in the wire declaration. This means there is no explicit cycle detection.
2. Variable indexed bit select is a key missing feature – this is useful for implementing memories
3. The associativity of most of the logic operators is wrong. For example, `5-1+2` would parse as `5-(1+2)` and result in 2 instead of 6.
4. Missing logical operators including: `==`, `!=`, `<`, `<=`, `>`, `>=`, no support for multiplication
5. The syntax errors are not always correct or helpful
6. The grammar described by the parser is ambiguous for some inputs
7. No simple net/variable declarations, only wire declaration with continuous assignment
8. Wire is 4-state, so simulation output is not identical to SystemVerilog behaviour
9. The legacy compiler has only been manually tested
10. All assignments and binary operators must have the same bit width on the left-hand side and the right-hand side

2.4.3 Legacy compiler design

The control flow of the base compiler can be seen on Figure 2.8. When the user updates the Verilog code in the editor, the updated code is passed in to the Nearley parser which then returns an error if there is a syntax error present, otherwise it returns the AST in the form of a JSON object [70]. The JSON is then parsed into an F# record structure using a standard JSON parsing function (Fable.SimpleJSON [71]). Finally, when the user saves their code, the AST is passed to the sheet creator function, which generates Issie components and connections from the given AST.

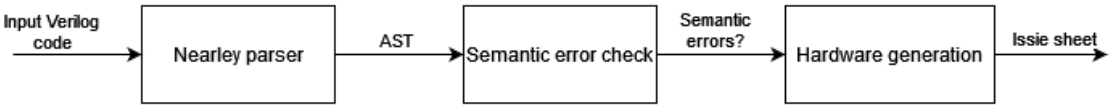


Figure 2.8: Base compiler control flow

As there is no lexer in the compiler, the syntax error messages given by the parser are only regarding the unexpected character, and it contains information about the expected character (not tokens). To improve the syntax error messages, the errors returned by the parser are processed in JavaScript, but this is not safe (for example because JavaScript has no type safety [72]) and the processed error messages are still not always correct.

The original compiler's AST data structure can be seen on Figure 2.9. The top level node of the AST is the VerilogInput which contains the Verilog module `ModuleT`. The module contains the `PortList` and the `ModuleItems`. A `ModuleItem` represents either a Statement or a port declaration. The `StatementT` in this data structure is always a continuous assignment and hence it contains an Assignment which stores the LHS and RHS expressions. The module item contains different optional fields: the field that is `Some _` determines the type of the item.

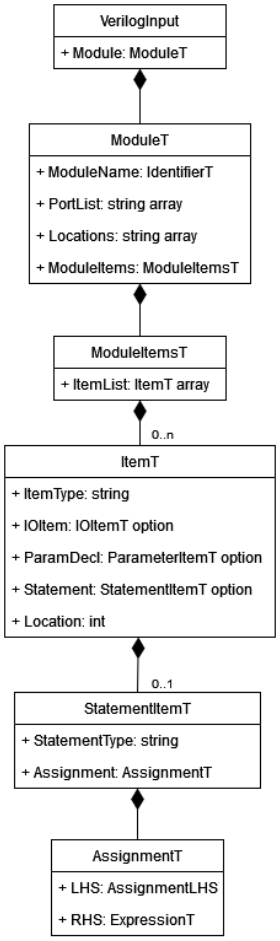


Figure 2.9: UML of the old data structure

Chapter 3

Requirements capture

This chapter gives a detailed specification of the requirements and objectives of the project. First, the state of the legacy compiler was assessed which was then discussed with Dr Thomas Clarke, the core maintainer of Issie, along with the possible features to implement during the project. The project requirements were finalized based on these discussions while taking into consideration the Issie principles, as described in chapter 1.

In the case of the Verilog compiler, the Issie principles mean that it is crucial that error handling is well done (**P2**): the user must be able to easily identify and correct syntactical and semantic errors. Using the UI for the Verilog editor should be intuitive to use without training (**P1**) and the error messages should also be informative and easy to understand for a complete beginner (**P5**).

By the end of the project, the compiler needs to support a subset of SystemVerilog. This restricted set of the HDL must be chosen carefully such that it allows students to describe hardware in a Verilog-compatible HDL as a bridge to later learning Verilog and encouraging good practices (discussed in subsection 2.2.6) when using HDL languages, as well as avoiding any unnecessary complexity in the implementation. The major challenges of this project include providing high quality error messages to the users, as well as thoroughly evaluating the compiler's correctness.

The objectives of the project can be summarised as the following set of essential (E) and desired (D) features:

E1 Determine the exact subset of the language to support as well as decide on additional restrictions, conforming to the Issie principles

E2 Add support for combinational always blocks

E3 Implement clocked always blocks

E4 Support blocking and non-blocking assignments

E5 Include multiple statements in an always construct by allowing sequential blocks (begin ... end statement)

E6 Support conditional statements

E7 Implement case statements

E8 Add informative error messages for the new features

E9 Fix associativity bug in legacy compiler grammar, described in subsection 2.4.2

E10 Full compatibility with SystemVerilog to allow students to move into real SystemVerilog later

D1 Add support for more language features, such as: structural Verilog (module instantiation statements), for loops and/or module parameters, arrays

D2 Develop an automated test bench, comparing the outputs to a third party simulator

D3 Improve syntax errors

D4 Add support for variable bit select and potentially variable bus select

D5 Make sure compiler is extensible and maintainable, so it can be further developed in the future

D6 No grammar ambiguities

D7 Add support for more logical operators such as `==`, `!=`, `<`, `<=`, `>`, `>=` as well as multiplication

D8 Ensure that the performance of compilation does not degrade the usability of the application

Section 4.1 illustrates the importance of the selected core features, and it describes how the desired features were prioritised during the implementation. Section 4.3 details the further language restrictions imposed by the Issie compiler.

Chapter 4

Analysis and design

This chapter discusses the high level design of the compiler, with a focus on the subset of SystemVerilog chosen to be supported.

4.1 Implemented features

Throughout the development process, the supported language concepts were chosen carefully based on the SystemVerilog design principles described in subsection 2.2.6. The features considered and their advantages and disadvantages are reviewed in table 4.1.

Feature	Advantages / Disadvantages	Implemented?
General `always` blocks	+ Gets the students used to simple Verilog − Can describe overly complex hardware such as latches − `always_comb` and `always_ff` are better to use	no
`always_comb` blocks	+ Separates combinational logic from clocked + Does not allow other procedural blocks or continuous assignments writing to its variables − Only supported by SystemVerilog not by Verilog	yes
`always_ff` blocks	+ Separates clocked logic from combinational + Does not allow other procedural blocks or continuous assignments writing to its variables − Only supported by SystemVerilog not by Verilog	yes
Conditional and case statements	+ Core feature of procedural blocks	yes
Sequential blocks	+ Core feature of procedural blocks	yes
Variable bit select	+ Core missing feature for Issie + Enables the implementation of signle bit multiplexers and demultiplexers − Requires a moderate amount of changes in the AST and the error handling	yes

Constant width variable part select	+ Core missing feature for Issie + Would enable the hardware description of N bit multiplexers and memories – Requires a moderate amount of changes in the AST and the error handling	no
Arrays	+ Core missing feature for Issie + Useful for implementing memories programatically – Rather complex implementation	no
Module instantiation statements	+ Encourages modularised designs + Hardware generation is simple as Issie already supports this	yes
For loops	+ Along with module parameters, for loops would allow the Verilog components to be "templated" which is a new feature to Issie + Hardware generation is simple (unrolling the loop) once always blocks are implemented – Would mostly be useful if module parameters were also implemented	no
Module parameters	+ They would allow Verilog components to be "templated" + Not very complex to implement for a single block containing several modules – Implementation of general module parameters is complex, it would require core Issie types to be updated and it is not straightforward how to do it	no

Table 4.1: Advantages and disadvantages of the language features considered for the compiler

As the main goal of the project was to add support for procedural blocks (see requirement capture, chapter 3) it was important to decide what types of always blocks to implement. The flexibility of general `always` blocks would allow users to implement unnecessarily complex hardware, and it would be difficult for beginners to learn to write good Verilog code with such flexibility. Hence, by the Issie and Verilog principles always blocks need some restrictions:

- Only allow `always@*` and `always @(posedge clk)` blocks
- Blocking assignments in combinational always blocks and non-blocking assignments in clocked always blocks
- Variables cannot be driven by multiple procedural blocks or continuous assignments

These restrictions are identical to the restrictions `always_ff` and `always_comb` blocks provide. As these always blocks are safer to use than the simple `always` blocks, the more restrictive SystemVerilog `always_ff` and `always_comb` blocks were implemented, and simple always blocks are not supported by the final compiler. The compiler also must support conditional statements, case statements as well as sequential blocks as these are key constructs when describing hardware in SystemVerilog. Variable bit select of buses is also an important feature missing from Issie, hence this is also supported.

Supporting arrays would be a useful enhancement for implementing memories and other data structures. However, the same functionality can always be achieved by using buses, thus arrays were not prioritised during the development, and they were not implemented in the final design.

As modularity of hardware design is essential, adding module instantiation statements to the compiler was of high priority. An advantage of module instantiation statements is that this is a standalone feature, in contrast with for loops and module parameters that would mostly be beneficial if they were implemented together. Module instantiation statements allow the user to write reusable code by utilising predefined modules across multiple designs. The hierarchy resulting from the nested structures also allows better description of the relationship between different components than a flat structure. Simplicity and readability of modularized Verilog components was also an appealing advantage. Module instantiation statements align with the Issie principles and even though the application already supports this for the schematic editor, Verilog module instantiation statements were determined to be a desirable functionality and hence they form a part of the final feature set.

Finally, for loops with module parameters (constants passed into modules to parameterise the logic that can define the width of data paths, memory sizes or the characteristics of other components in the module) are a powerful feature of Verilog. They would provide more flexibility by enabling the user to reuse the same module with varying configurations without duplicating the code. Parameterised modules also benefit from scalability and maintainability. However, these two features should be implemented together; one without the other provides much less flexibility to the user. Furthermore, after some investigation it was established that the implementation of module parameters requires changing some of the core types of Issie, making the development highly complex and so these features were not added to the application. If a single Verilog block can contain multiple Verilog modules (producing a single top level component), the implementation complexity can be reduced as the values of the module parameters will be part of the AST, so the Issie types do not need to be updated anymore. If module parameters were supported this way, the parameterised modules would be less reusable than in the first case as they can only be reused in a single block, not across different blocks, hence it was decided that the more restrictive version of module parameters would also not be supported.

4.2 Supported data types

As discussed in section 2.4, the base compiler only supports the wire data type which works for continuous assignments, however Verilog requires the use of variables (reg, logic or bit) in procedural blocks, so one of the important design decisions was to determine which data types to add support for. Based on subsection 2.2.1 deciding between using wire or reg is confusing for beginners. Furthermore, one should always use logic as reviewed in subsection 2.2.6 and in the case of using the logic type, the compiler can infer the correct piece of hardware to generate (register or wire) instead of expecting the user to do it, potentially incorrectly. However, as the logic type has 4 states and is initialised to unknown (x), this would cause compatibility issues between Issie and the language standard [22] since Issie only differentiates between 2 states for each bit: 0 and 1. The 2-state logic in Issie is equivalent to using the bit data type in SystemVerilog, which has the same advantages of using logic. Hence, even though logic is the more commonly used data type, to adhere to third party simulators, in the final compiler design solely the bit data type is supported. According to the SystemVerilog standard, ports without type specifiers are by default defined as wires [22] thus every variable and port must be declared explicitly as type bit.

4.3 Further language restrictions

As pointed out in subsection 2.2.6, further language restrictions were necessary to enable beginners to learn quickly and to meet the Issie principles. These restrictions are summarised in table 4.2.

Restriction	Verilog principle or other reason
Only blocking assignments in always_comb and non-blocking assignments in always_ff	Separate combinational and sequential logic

A variable can only be driven by continuous assignments, a single always block or a module instantiation statement	Bit cannot be driven by multiple always blocks. Issie also doesn't support multi-driver components
Case statements items must be constants	Case items must be known at compile time according to the SystemVerilog standard [22]
Combinational variables must be assigned to in every branch of always_comb blocks	There are no undefined values in Issie
Clock signal cannot be used other than in the sensitivity list of always_ff and only `posedge clk` is allowed	Otherwise complex hardware could be generated accidentally. The clock signal is also not available for use in Issie
No cycles allowed in combinational logic	The Issie simulator does not support this
No unused variables or ports	It is good practice not to have unnecessary variables
Variables in an always_comb block must not be read first and then written	This can create unexpected hardware when synthesized
No assignments allowed where the assignment right-hand side width is greater than the left-hand side expression	Prevents accidental data slicing
Module instantiation ports must be the correct width	If the user wants to connect the lower bits or pad the value with zeros, they should be explicit about it
Module instantiation inputs and outputs must be primaries, expressions are not allowed	Expression bit lengths are not always obvious to the user
No duplicate case items in case statement	Verilog principle

Table 4.2: Language restrictions imposed and the reasoning behind them

4.4 Lexical analysis

In the base compiler, there is no lexer (see subsection 2.4.3, which means that the Nearley parser reads the input character by character. As described in subsection 2.3.2 adding a lexer to the compiler is beneficial in terms of efficiency and maintainability, so it was important to consider adding a lexer to the Issie compiler. However, the main benefit of adding a lexer in this case was to improve syntax errors. By tokenizing the input, the parser is able to produce meaningful syntax error messages. For example, a possible syntax error without the lexer was "Unexpected character 'x', expected: 'e', 'd'". In contrast, with the lexer this error would be: "Unexpected {IDENTIFIER} token, expected: 'endcase', 'default'", which is undoubtedly much more helpful for the user than the previous error. The improved syntax errors also meant that the tedious post-processing of the generated syntax errors was eliminated, improving the parser's maintainability.

Even though adding a tokenizer meant that the grammar in the parser had to be significantly reworked, the remarkable enhancement of the syntax errors was decided to be worth the extra development time and a lexer was added to the compiler design, see Figure 4.1. When choosing a lexer, the main priority was compatibility with Nearley. As Nearley supports and recommends Moo [69], a highly optimised lexer [73], Moo was a good option to pick. There are other lexers compatible with Nearley [69] but as Moo is the recommended lexer, there are a lot of resources available on using Nearley with Moo including numerous examples, so for the sake of simplicity Moo was chosen as the lexer.

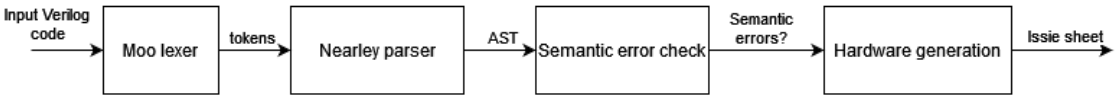


Figure 4.1: Compiler control flow with the lexer

4.5 AST data structure

The Nearley parser returns the AST in JSON format which is then converted into F# records using the F# JSON library (see section 2.4 for details), with the error handling and the Issie component generation are done on this same data structure.

In the original data structure Statements are always continuous assignments (either to a wire or to an input port) so in the new data structure they were renamed to ContinuousAssign. Based on the SystemVerilog grammar always constructs are a type of module item, so in the new design a module item (`ItemT`) is either a port declaration, a continuous assign or an always construct. An always construct needs to store the type of the block (always_ff or always_comb) and a Statement. The Statement can then either be a non-blocking assignment, a blocking assignment, a case statement, a sequential block or a conditional statement. The UML diagram of the planned data structure can be seen on Figure 4.2.

The old data structure contains many optional fields to represent a parent class that can be either one of its children. This could be also represented as a DU (discriminated union [74]) in F#. Handling many Options can make the code less legible. To add a new type of module item, the old pattern could be used, and an optional `AlwaysConstruct` field would be added to the `ItemT` record. Another possibility is to change the `ItemT` record into a DU. To do the latter, the JSON AST would still be parsed into the records using options – as the JSON library cannot parse into a DU type – but then this intermediate data structure would be converted into a new data structure that uses DU's which the rest of the processing would be done on. This would make the codebase more legible, and it could potentially make it easier to add new features in the future. However, this requires substantial changes in the current code base, which would take a significant amount of development hours. Furthermore, converting from one data structure to the other adds another layer of error handling which also makes the development more complex and representing the same information with two sets of data structures also adds extra complexity. Thus, to support procedural blocks, the old pattern with options was used in the project. The details of how the AST was then processed for error handling and synthesis are discussed in the Implementation section.

As there are two types of always constructs the compiler must support, the `AlwaysType` stores whether the construct is `always_ff` or `always_comb`. Another possibility for storing always blocks was to have separate types for clocked and combinational always blocks. However, since the two types of procedural blocks are nearly identical for the majority of error handling and hardware generation, they were represented by a single record to abstract the shared logic.

The structure of the StatementT type was designed similarly to the module items: the various optional fields determine the type of the statement. The types NonBlockingAssignment and BlockingAssignment contain an Assignment so that the common logic between continuous assignments and procedural assignments can be abstracted. Sequential blocks (SeqBlock) represent the compound statement (set of statements enclosed by 'begin' and 'end') and they simply contain an array of statements. Conditional statements need to store the condition expression, the statement of the 'if' branch and an optional statement for the 'else' branch. Case statements also store a condition expression as well as an array of case items. Each case item contains an array of case item expressions along with a single statement to represent a branch of the case statement.

4.6 Hardware generation

For synthesizing the AST, two options were considered. One of them was to follow the base compiler's code generation and construct hardware in the form of Issie components; and the other one was to write a special Verilog simulator that takes in the values at each port in the previous clock cycle and simply returns the values at the next clock cycle. As both of these alternatives were feasible solutions, their benefits are discussed in Table 4.3.

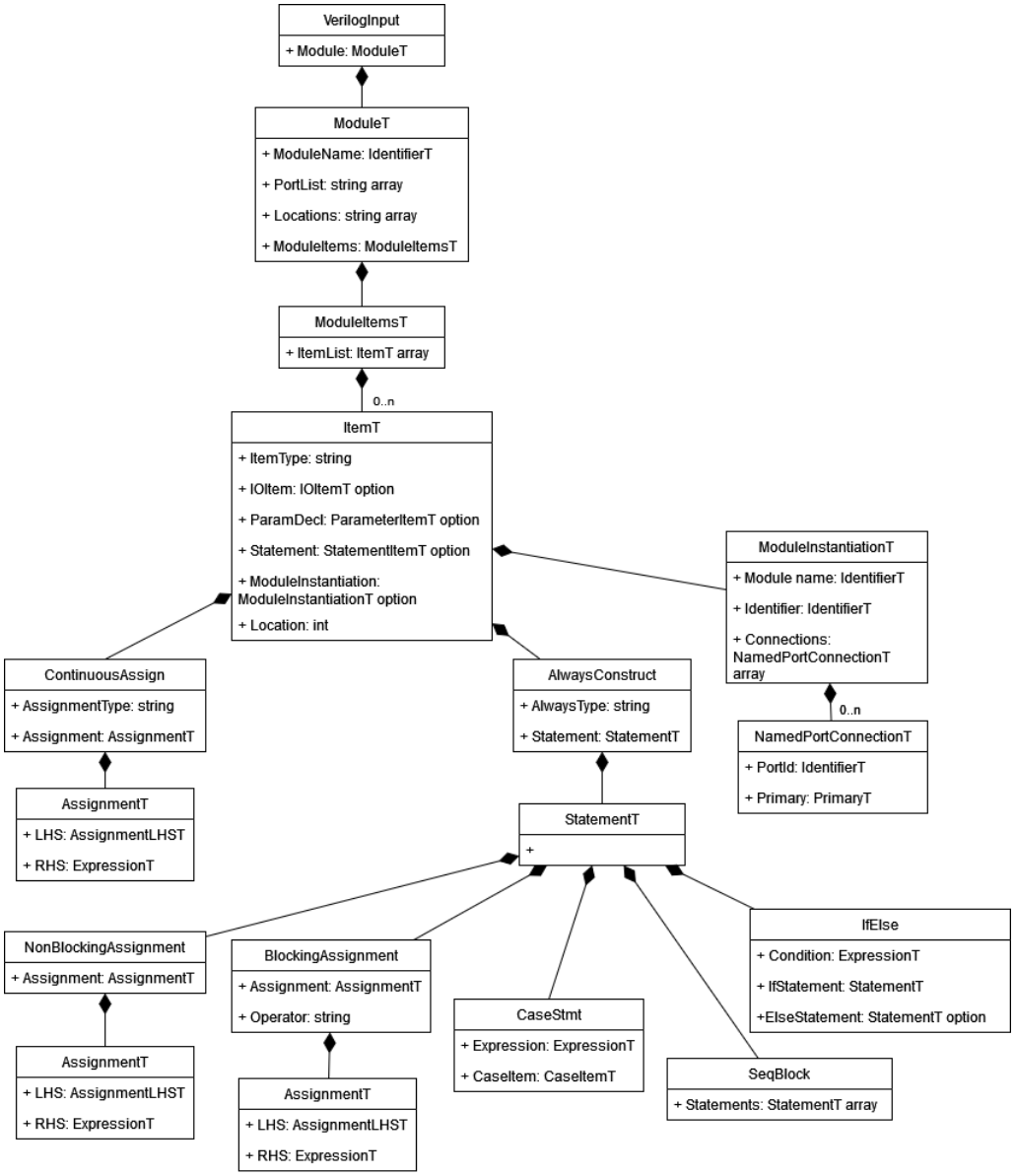


Figure 4.2: UML of the new data structure

Issie component generation	Special Verilog simulator
+ It could allow the students to see the generated hardware and hence understand the underlying structures	+ Straightforward implementation
+ By testing the Verilog compiler against third party simulators also tests the entire Verilog simulator	+ Simple testing, as there is no need to use the Issie simulator
+ Enjoyable to implement	

Table 4.3: The advantages of the two hardware generation alternatives

After considering the advantages of the two options, it was decided that Issie components would be generated as the educational benefit outweighed the ease of implementation of a special Verilog simulator.

4.7 The final product

To summarise this chapter, the final product supports the following language features:

1. Flip-flop-type always_ff blocks
2. Simple combinational always_comb blocks, no cycles allowed
3. Blocking assignments in always_comb blocks
4. Non-blocking assignments in always_ff blocks
5. Variable bit select
6. Variable declarations (type bit)
7. Most operators in Verilog, including multiplication
8. Expression evaluation matching the SystemVerilog rules (see subsection 2.2.2)
9. Module instantiation statements
10. Buses and N bit variables
11. Continuous assignments
12. Port declarations (type bit)

Furthermore, lexical analysis is done before parsing for useful error messages and a performance boost, and detailed error handling was added for providing helpful semantic error messages with potential fixes. Chapter 5 describes in detail how the different features were implemented.

Chapter 5

Implementation

This Chapter discusses how the compiler was implemented, focusing on parsing, semantic error handling as well as the hardware generation.

5.1 Representing the clock signal

One important design decision of the project was to determine how the clock signal would be represented. In Issie, there is a single, internal clock signal which is implicitly the input of all clocked components (for example registers) and this clock signal is not available for reading. Users also cannot easily define custom clock signals. One option for representing clocked logic in Verilog was to change this and make the clock signal available for the users. Nonetheless, the development of this would have been overly complex as the core application logic would have had to be updated and being able to customize the clock or having to input it to all clocked components would make the application more difficult to use than it is now. Another option was to follow the current representation of the clock signal in the Verilog editor too: the clock signal would only appear in the sensitivity list of the `always_ff` block and nowhere else (so not in any of the expressions). The clock signal cannot be a real input of a component in Issie, so it would have been simple not to allow the clock signal as the input, however, this would create a compatibility issue with SystemVerilog as the identifier in the sensitivity list of the sequential always blocks would be an undefined variable.

Hence, for ease of use and compatibility with SystemVerilog, the clock signal is represented as `clk` and it must be an input port of the module in order to be used in the sensitivity list of an `always_ff` block.

Representing the clock has important considerations regarding module instantiation statements as well. In Verilog, the clock signal must be contained by the port mappings in case of a clocked component. Hence, for compatibility reasons, the Issie compiler should handle clocked module instantiations in the same way. However, the only way to determine if a component is clocked in Issie currently is by recursively looking through all sheets and components in the project [1]. As the compiler can generate hundreds of components for some inputs, this would significantly increase the error check and synthesis times, thus it was decided that when instantiating modules in a Verilog block, the clock signal need not be included in the port mapping.

5.2 Grammar

The base of the grammar used in Issie was the 2017 IEEE standard for SystemVerilog (IEEE 1800-2017 [22]) as this is the most recent stable release available [75]. The complete SystemVerilog grammar is extremely large and since only a small subset of the language is supported in Issie, the grammar was simplified and only the relevant production rules were kept. The restrictions mentioned in section 4.3 above further simplified the grammar.

The new grammar added for always blocks is similar to the grammar described in Listing 5.1 (based on the IEEE Standard for SystemVerilog [22]):

[1] See here

```
ALWAYS_CONSTRUCT ::=
    always_comb STATEMENT
    | always_ff @ ( posedge clk ) STATEMENT

BLOCKING_ASSIGNMENT ::= VARIABLE_LVALUE = EXPRESSION

NONBLOCKING_ASSIGNMENT ::= VARIABLE_LVALUE <= EXPRESSION

SEQ_BLOCK ::= begin STATEMENT:+ end

CONDITIONAL_STATEMENT ::= IF ELSE:?

IF ::= if ( EXPRESSION ) STATEMENT
ELSE ::= else STATEMENT

STATEMENT ::=
    NONBLOCKING_ASSIGNMENT ;
    | BLOCKING_ASSIGNMENT ;
    | SEQ_BLOCK
    | CONDITIONAL_STATEMENT
    | CASE_STATEMENT
```

Listing 5.1: Always grammar

Note that the grammar used enforces some of the restrictions mentioned in section 4.3:

- The production rules only allow for always_ff and always_comb blocks, so no always blocks allowed
- The sensitivity list of the clocked always blocks must be `@(posedge clk)`
- The clock signal must be represented by the `clk` identifier

The grammar could have been more general, and then the above restrictions would have been implemented as semantic errors. However, this would have meant writing a more complicated set of production rules as well as extra semantic error handling needed. As the syntax errors provided by the parser for these restrictions are helpful, it was decided that the grammar would be strict, and it would already filter out some malformed inputs.

5.2.1 Grammar ambiguity

When writing a parser, it is crucial to make sure the grammar is not ambiguous. Ambiguous grammar means that there are multiple possible parses for the same piece of input, which means that some of these parses might be incorrect, so this means that the grammar has possible mistakes in it and hence grammar ambiguity should be avoided. As discussed in subsection 2.3.7, Nearley can handle ambiguity gracefully, and it returns all possible parses, which made it rather simple to see if an input had multiple possible parses. However, the parser does not give a warning or an error about ambiguity in the grammar, so a lot of care was needed to write the grammar and to check for ambiguous inputs while testing.

One major issue that caused ambiguity in the base compiler and during development was the way whitespaces are handled. Whitespaces are parsed as zero or more (or one or more) whitespace characters, and they must be a part of the Nearley grammar. However, the example in Listing 5.2 introduces ambiguity. Note that _ represents zero or more whitespace characters.

```
ALWAYS_CONSTRUCT ::=
    always_comb _ STATEMENT _
STATEMENT ::=
    | BLOCKING_ASSIGNMENT _ ; _
```

Listing 5.2: Ambiguity caused by whitespaces

In this case, both a statement and an always construct can be followed by any number of whitespaces. In this case, if the input grammar had a single whitespace character after the semicolon in an always construct, then this input would have two possible parses: one where the whitespace is part of the ALWAYS_CONSTRUCT rule and one where the whitespace is part of the STATEMENT rule. Although both of these parses are correct as it does not matter where the whitespace belongs, this creates a huge memory footprint as the number of possible parses grows linearly with

the number of whitespace characters present and combinatorially with the number of ambiguous rules. As Nearley stores all the possible parses, the memory load can cause the entire app to crash. Hence, the grammar was written with extreme care to avoid these types of ambiguities and the ambiguity present in the base parser was also removed.

5.2.2 The dangling if-else problem

The dangling else problem is another problem that causes ambiguity, which occurs when using nested if statements [76]. Consider the example in Listing 5.3.

```
if(expr1)
    if(expr2)
        statement1;
    else statement2;
```

Listing 5.3: Code with dangling else

In this case, there is a dangling else, as it is unclear which if statement the else branch belongs to. The grammar described in section 5.2 is ambiguous in the case of a dangling else: it would return two possible parses for this example. To resolve this ambiguity, most programming languages prefer to attach the dangling else to the innermost if statement [48]. To avoid the dangling if-else problem, two solutions were considered:

1. Require the begin and end keywords for every if/else statement

2. Update the grammar to follow the general rule of attaching the dangling else to the closest unmatched if statement [48]

Both of these options have their advantages. Requiring the begin and else keywords is very easy to implement, and this way the user would avoid the dangling else problem overall. Resolving the ambiguity in the grammar is also not very complex to implement, but it was established to be better than the first option, as the begin and end keywords can add a lot of extra 'noise' to the Verilog code, making it hard to read.

The grammar in section 5.2 was thus updated so that the statement appearing between any if and else must be matched, i.e. it must not end with an unmatched if statement [48]. Based on Compilers: Principles, Techniques, and Tools, the unambiguous production rules were written as shown in Listing 5.4.

```
STATEMENT
    ::= MATCHED_STATEMENT
    | OPEN_STATEMENT
MATCHED_STATEMENT
    ::= if ( EXPRESSION ) MATCHED_STATEMENT else MATCHED_STATEMENT
    |   OTHER_STATEMENT
OPEN_STATEMENT
    ::= if ( EXPRESSION ) STATEMENT
    | if ( EXPRESSION ) MATCHED_STATEMENT else OPENSTATEMENT
```

Listing 5.4: Solution to the danglin else problem

5.2.3 Module instantiation statements

As described in subsection 2.2.5, the module instantiation statement port mapping can be either named or ordered. As Issie does not define a specific order in the input and output ports of a component (they can be on any side of a custom component, in any order the user sets them to) named port mappings align more with the general Issie logic than the ordered port mappings. Furthermore, named port mappings make the end user aware of how exactly they are connecting the components in Verilog designs. Named port mappings also allow for better error messages in the case of missing ports or width mismatches. Hence, it was decided that only named port mappings would be supported in the Issie compiler. The grammar needed for supporting module instantiation statements can be seen in Listing 5.5

```
MODULE_INSTANTIATION_STATEMENT
    ::= IDENTIFIER IDENTIFIER ( LIST_OF_PORT_CONNECTIONS ) ;
LIST_OF_PORT_CONNECTIONS
    ::= NAMED_PORT_CONNECTION  { , NAMED_PORT_CONNECTION }:*
NAMED_PORT_CONNECTION
    ::= . IDENTIFIER ( MODULE_INSTANTIATION_PRIMARY )
// module instantiation primaries are variables or constant bit select
    expressions
```

Listing 5.5: Grammar for module instantiation statements

5.3 User interface

The UI of the Issie Verilog IDE is shown in Figure 5.1. Note that the user interface was not updated compared to the legacy compiler, except for the extra error message table being permanently visible next to the text editor. Note that in the case of syntax and semantic errors, dashed red underlines appear in the Verilog input. The error messages appear superimposed on the editor when hovering over the red underline. The table on the right of the editor contain a list of extra error messages that give more detailed information on the errors present. For certain errors, the table also contains a suggestion. When the user clicks the suggestion, the compiler automatically corrects the input program.

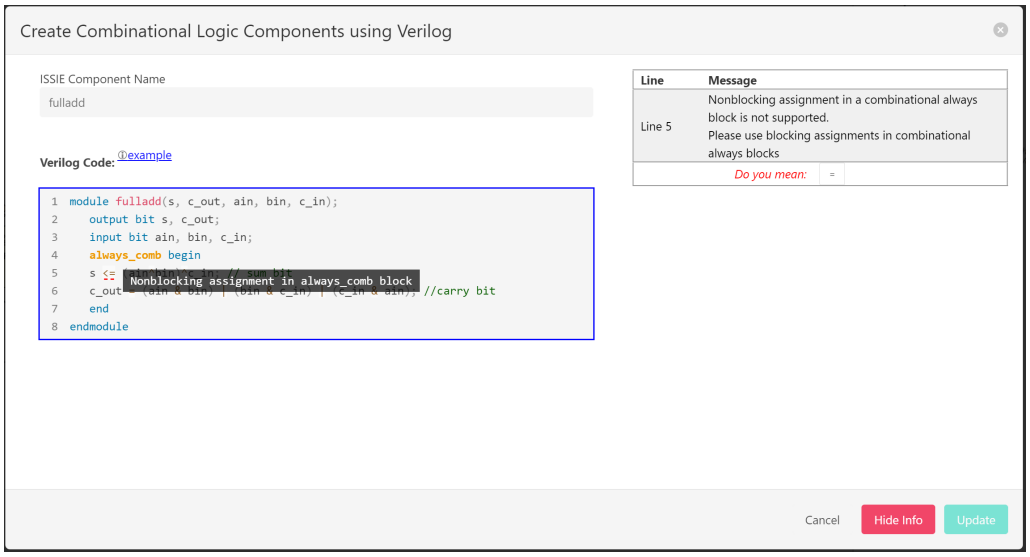


Figure 5.1: Issie Verilog IDE

5.4 Error handling

This section focuses on the error handling added for processing procedural blocks.

5.4.1 Interacting with the AST

As reviewed in section 4.5, the AST structure is tedious to work with because of the different optional fields. To make error handling (and hardware generation) easier, an `ASTNode` type was added. This type is an F# discriminated union that represents all the nodes in the AST. This allowed the development of error handling functions that can recursively iterate through the AST using simple `match` statements.

To make it even simpler to interact with the data structure, an AST fold function (see Appendix B) was added. This function is similar to the standard F# fold functions: it takes in a state of any type, a folder function and an `ASTNode`. Adding this function abstracted away a lot of the common logic needed for error handling and reduced the development time needed for error handling as well as the amount of repeated code compared to working with the raw AST. The

`foldAST` makes it easy to execute any function on each node of the AST. For example, getting all the expression nodes of the tree can be done as shown in Listing 5.6.

```
/// get RHS expressions from always, continuous assign, case stmt...
let getExpressions (expressions: List<ExpressionT>) (node: ASTNode) =
    match node with
    | Expression expr ->
        expressions @ [expr]
    | _ -> expressions
let expressions = foldAST getExpressions [] verilogInput // returns a list of
    all the expressions in the tree
```

Listing 5.6: Getting every expression in the AST using foldAST

5.4.2 Cycle detection

As cycles in combinational logic in Issie are not allowed, the error handling includes checking for cycles. The dependencies of a combinational circuit form a directed graph. To implement cycle detection, known graph search algorithms were considered to be used. Both depth first search (DFS) and breadth first search (BFS) could be suitable for cycle detection, and they have similar properties. They both have O(E+V) time complexity and O(V) space complexity [77] where E is the number of edges (number of dependencies, it roughly translates to the number of combinational assignments) and V is the number of vertices (number of variables used in the module). As DFS was easier to implement in F#, this graph search method was chosen. The implementation of `findCycleDFS` based on [78] is shown in Listing 5.7.

```
type Graph = Map<string, string list>
/// Looks for cycles in the general dependency graph using depth first search
let findCycleDFS (graph: Graph) : string list option =
    let rec dfs (node: string) (visited: Set<string>) (recStack: Set<string>) (
    path: string list) : string list option =
        if recStack.Contains(node) then
            // Cycle detected, return the path as Some
            Some (node :: path)
        elif visited.Contains(node) then
            // Node has already been visited, no cycle found
            None
        else
            // Add the node to the visited and recursion stack sets
            let visited' = visited.Add(node)
            let recStack' = recStack.Add(node)
            // Get the neighbors of the current node
            let neighbors =
                graph
                |> Map.tryFind node
                |> Option.defaultValue []
            // Recursively visit the neighbors. If a cycle is found, return it
    as Some, otherwise None
            let findCycle =
                (None, neighbors)
                ||> List.fold (fun acc neighbor ->
                    match dfs neighbor visited' recStack' (node :: path) with
                    | Some cyclePath -> Some cyclePath
                    | None -> acc)
            findCycle

    // Iterate through all the nodes in the graph and check for cycles
    graph |> Map.keys
          |> Seq.tryPick (fun node -> dfs node Set.empty Set.empty [])
```

Listing 5.7: Cycle detection using DFS

To detect cycles, first a dependency graph must be obtained. This is done by iterating through the AST and whenever a continuous assignment or an assignment in an always_comb block is encountered every bit on the right-hand side is added to the graph as a dependency of every bit on the left-hand side. In the case of if statements and case statements, the bits of all the variables in the condition expression also form a dependency of the variables assigned to in the conditional or case statement branches. Assignments in the `always_ff` blocks are ignored, since

these assignments represent a flip-flop which breaks a possible cycle. After the dependency graph is created, a standard DFS algorithm is run to determine if there are cycles. If there are cycles, their paths are printed out as part of the error message, marked at the end of the module.

Note that often it is not the case that all bits on the left-hand side of an assignment are dependent on all bits on the right-hand side. For example, in the assignment, `a=b;` the only dependency of `a[i]` is `b[i]`. However, when the right hand side expression is more complex, determining a more accurate dependency graph is a lot more complicated to implement than the method described above. It would also be useful for the user to see the location of the cycle, but as the cycle consists of numerous variables that can appear in any complex structure of if statements or case statements, this would have been also very complicated to implement and was decided to be outside the scope of the project.

5.4.3 Semantic error summary

Table 5.1 details the different error messages. This table contains the error messages appearing in the editor when hovering over the red underline, as well as the extra error messages appearing in the extra error message table (described in subsection 2.4.3) next to the editor to provide more detailed information. In some cases, the extra error message table also contains suggestions. When the user clicks on the suggestion, the error is automatically corrected in the editor. The location column describes where the red underline appears.

Error reason	Error and extra error message	Suggestion	Location
Blocking instead of nonblocking assignment	"Blocking assignment in always_ff block" "Blocking assignment in a clocked always block is not supported. Please use nonblocking assignments in clocked always blocks"	Replace '=' to '<='	'=' symbol
Nonblocking instead of blocking assignment	"Nonblocking assignment in always_comb block" "Nonblocking assignment in a combinational always block is not supported. Please use blocking assignments in combinational always blocks"	Replace '<=' to '='	'<=' symbol
Multi-driven variable	"Some ports or variables are driven by multiple always blocks or continuous assignments." "The following variables are driven by multiple always blocks or continuous assignments: <list of variables>. Please make sure that every port is driven by at most one always block or continuous assignment."		endmodule
Duplicate case value	"Duplicate case value" "The following case value is duplicated: <case item>, see line <line number>. Please make sure there are no repeated case values."		Second occurrence of case item
Wrong case item width	"Width of case value does not match the given case expression" "Width of case expression (<x> bits wide) does not match width of this case item (<y> bits wide)."	Replace to correct width	Affected case item

Undefined variable in always_comb	"Some ports or variables might not always be assigned to" "The following variables might be undefined as they are not assigned to in every branch of conditional statements or case statements in the always_comb block: <list of variables>"		always_comb keyword of affected block(s)
Undefined reference to 'clk'	"Variable 'clk' is not defined as an input port." "To use always_ff blocks, please make sure to include 'clk' in the port list."	Insert 'input logic clk;'	'clk' in sensitivity list
'clk' output port	"'clk' must be an input port" "'clk' represents the clock signal, which must be an input port"		Port name
'clk' variable	"Variable cannot be called 'clk'" "'clk' is reserved for the clock signal, please rename the variable"		Variable name
Wrong clock width	"'clk' must have width 1" "'clk' represents the clock signal, which must have width 1, here it has width <w>"		Port name
Using 'clk' in expressions	"Illegal use of 'clk'" "'clk' represents the clock signal, make sure to only use it as 'always_ff @(posedge clk)'"		'clk' name in expression
Cycle in combinational logic	"The following variables form a dependency cycle: <list of variables>" "The following variables form a cycle: <list of variables>"		endmodule
Reading then writing a variable in always_comb	"Variable written to after it is read" "The following variables are read and then updated: <variable name>. This creates undefined behaviour in an always_comb block, please make sure to not update a variable after it is read."		Assignment left-hand side
Undefined module type in module instantiation	"Component <name> does not exist" "Component <name> does not exist - there is no custom component or Verilog component with this name"	Replace to closest component name	Module type
Undefined port in module instantiation	"No such port for the given component" "The port <port name> does not exist for component <component name>"		Port identifier
Missing port in module instantiation	"Missing port(s) for component" "The port <port name> missing for component <component name>"		Module type

Invalid output port for module instantiation	"Output port already driven" "Output port <port name> in module <module name> is already driven by continuous or procedural assignments"		Connected primary
Incorrect port width for module instantiation	"Wrong port width" "Wrong port width for port <port name> in module <module name>: <x> bits wide but expected <y> bits"		Connected primary

Table 5.1: Summary of semantic error messages

Note that the above table only contains information on the error messages added for the new feature. The error handling functions of the base compiler are still in use, and they have been modified to accommodate for procedural blocks and module instantiation statements. For the details of the rest of the error messages, see Appendix C.

5.5 Hardware generation

The approach adopted for hardware generation involved the creation of a dedicated circuit for each variable and output port for every node of the AST. This design choice allowed for a modular and efficient process of circuit construction and updating.

At each node of the AST, the circuits associated with the variables and output ports were updated based on the properties and operations defined by the node. This updating process involved incorporating the necessary logic gates, registers, and interconnections to reflect the desired functionality and behaviour of the hardware circuit. By leveraging the existing circuits and selectively modifying them, the overall hardware generation process was streamlined.

Functional programming principles, such as immutability and higher-order functions, were employed to manipulate the circuits effectively, resulting in an elegant codebase.

5.5.1 Circuit manipulation

In the process of implementing the hardware generation, two different solutions were considered to handle the storage and updating of circuits for variables. The first solution involved storing circuits for slices (consecutive bits) of variables, utilizing a mapping structure that linked variables to their corresponding slices and circuits. This approach would have enabled updating only the relevant slice of a variable at each assignment, dynamically splitting or merging slices as needed, which would result in a precise representation of the Verilog input. The main advantage of this solution is that for some inputs it would generate more easily understandable hardware than the other solution, hence it would give a better opportunity for students to learn from the underlying hardware of their Verilog code.

However, upon further evaluation, the second option, which involved storing a single circuit for the entire variable, was chosen as the preferred solution. This decision was based on several factors, including ease of implementation. The slice-based approach would have introduced additional complexities and challenges, particularly when dealing with conditional statements and case statements. Implementing these constructs with the slice method would have required additional development hours and increased code complexity and reduced maintainability. It was also established that for most Verilog inputs, the two solutions would produce identical hardware, the difference only occurring for some unusual bit-select or part-select logic that is unlikely to occur.

The single circuit approach involved creating an initial circuit for every variable. This initial circuit is simply a constant zero of the width of the corresponding variable. Then, at every AST node, the previous circuit for the relevant variables are updated to create the next circuit; see Figure 5.2 for the high level design hardware generation flow.

By opting for a single circuit for each variable, the implementation process became streamlined and straightforward. The entire circuit associated with a variable is updated at each assignment, ensuring that the desired behaviour and functionality of the variable were consistently maintained.

This meant that when generating the Issie components and connections for a node in the AST, existing circuits associated with variables did not need to be a concern at each node. Although this approach required updating the entire circuit, the benefits in terms of simplicity and ease of implementation outweighed the potential overhead in circuit updates of the first option. By choosing this approach, the hardware generation became more manageable and less prone to errors, facilitating a more efficient and effective development process.

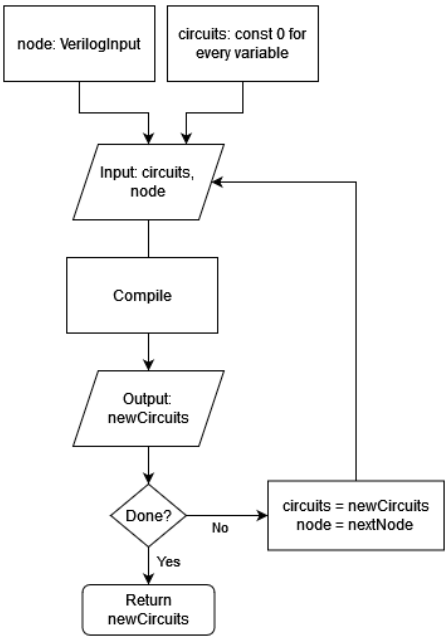


Figure 5.2: Hardware generation flow chart

5.5.2 Combinational logic

Hardware generation for combinational logic involved associating every combinational variable and output port with a wire label.

During the compilation process, when a variable appears on the right-hand side of an assignment, the output of the wire label associated with that variable is connected to the relevant circuit. This ensures that the final value of the variable will be connected to the circuit. The detailed logic used to traverse and compile the AST are shown in Figure 5.3

Once all the necessary circuits are established, the final step involves connecting each variable's circuit to the input of its corresponding wire label. This connection enables the input of the wire label to receive the value produced by the circuit associated with the variable.

For output ports, the process involves connecting the wire label associated with the output port to the corresponding output port component. This connection ensures that the value propagated through the wire label would be delivered to the appropriate output port, allowing it to be externally accessible.

By utilizing this approach, the implementation successfully represented SystemVerilog combinational logic, ensuring the accurate representation and functioning of the combinational logic in the generated hardware circuits.

5.5.3 Sequential logic

The hardware generation for sequential logic is identical to the combinational circuit generation described above. However, clocked variables are associated with a register instead of a wire label to ensure the desired sequential behaviour.

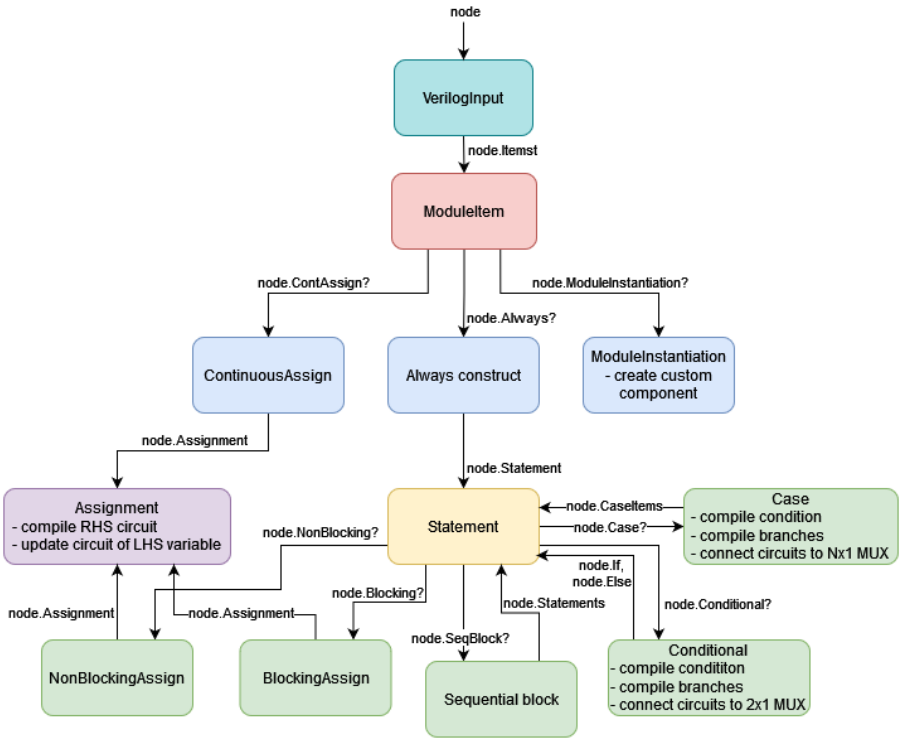


Figure 5.3: Traversing the AST for hardware generation

5.5.4 Expression widths

Following the expression and assignment evaluation rules discussed in subsection 2.2.2, before hardware generation, the expression widths are determined as follows:

1. Evaluate right-hand side expression by recursively traversing the expression
2. Save the calculated width at each node
3. Determine target width by taking the maximum of LHS width and the RHS width from before
4. Pass in the target width into the compile function
5. For context determined operators, pass in the target width from before to the recursive function call. For self-determined operators, pass in the width corresponding to the node of the operator.
6. After the circuits for context determined operators are created, the circuit is extended to match the target width if necessary.

Note that as the compiler only supports unsigned variables and integer literals, when a circuit is extended, it is always padded with zeros and is never sign extended.

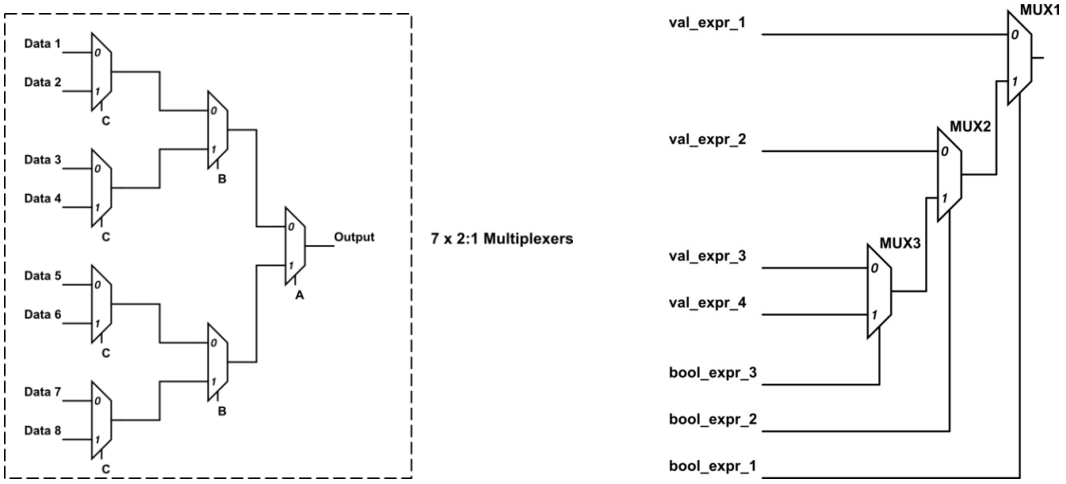
5.5.5 Case statements

Case statements in general synthesize to highly optimized decode logic [79]. However, Issie does not support custom decoders, hence different solutions were considered for the hardware generation of case statements.

An initial naive implementation translated the case statement into a single N-bit multiplexer, where N is the number of possible case items. As Issie does not support general Nx1 multiplexers, this was done have by composing 2×1 multiplexers (or ideally 8×1) multiplexers into an Nx1 multiplexer, as shown in Figure 5.4a. This solution creates $O(2^M)$ components where M is the bit width of the case expression, regardless of the number of case items present in the case statement. When this implementation was tested with a 16-bit wide case expression and 15 case items (which

is a realistic example), the whole application crashed during the hardware generation as $O(2^{16})$ components were being created. Hence, this solution is not feasible since the users could easily crash the app with a simple case statement.

The final hardware generation implementation of case statements treats case statements the same way as a set of if-else statements by chaining 2×1 multiplexers in a row, as shown in Figure 5.4b. This optimized implementation generates $O(N)$ components where N is the number of case statements. The number of case statements is always at most 2^{M} where M is the bit-width of the case expression, showing the improvement in complexity. Users can easily write a case statement with a 10 bit case expression but they will not write 2^{10} case statements, hence this solution is more optimal than the naive implementation.



(a) Naive implementation of case statements. Image credits [80]

(b) Optimized implementation of case statements. Image credits [81]

Figure 5.4: Hardware generation options for case statements

5.5.6 Variable bit select

Variable bit select consisted of two tasks:

1. Supporting variable bit select in expressions: `a=b[i];`

2. Variable bit select on the LHS of assignments: `a[i]=b;`

The hardware generation for both of these tasks can be implemented in various ways. A naive way to support 1. is by generating an Nx1 multiplexer for the right hand side variable, which would select bit `i`. As Issie only has 2×1, 4×1 and 8×1 multiplexers, this solution would require the generation of O(N) components where N is the bit width of the variable being indexed (about N components for the multiplexers and N more for bus selection components that split up the variable into bits). A more optimal solution can be achieved by manipulating the bits of the indexed circuit using a set of N-bit logical gates. Given an integer b, the i^{th} bit can be accessed (for reading) as follows: `(b & (1 « i)) » i` [82]. The same logic can be used for implementing variable bit selects in Issie, as seen in Figure 5.5. Note that the variable Shift components are internal components to Issie. This option always requires the generation of exactly 5 components (constant 1, two shift components, an AND gate and a bus select) regardless of the widths of the inputs. Hence, this solution was chosen over the multiplexer solution described above, making the generated hardware scalable, allowing for fast compilation and simulation of complex circuits.

For 2. similar approaches were considered, and for optimal complexity the solution including N bit logical gates was chosen in this case too. Given an integer a, the i^{th} bit can be updated to b as follows: `(a &  (1 « i)) | (b « i)` [83]. The equivalent circuit in Issie is shown in Figure 5.6.

Note that the same logic can be extended to M bit variable bus select by using the constant $2^{\mathrm{M}} - 1$ instead of the constant 1 in the above circuits.

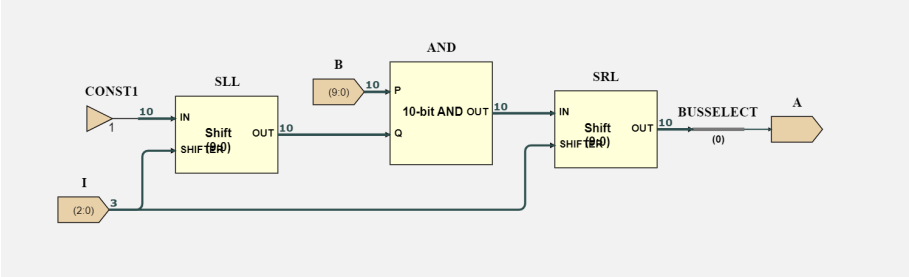


Figure 5.5: Variable bit indexing circuit

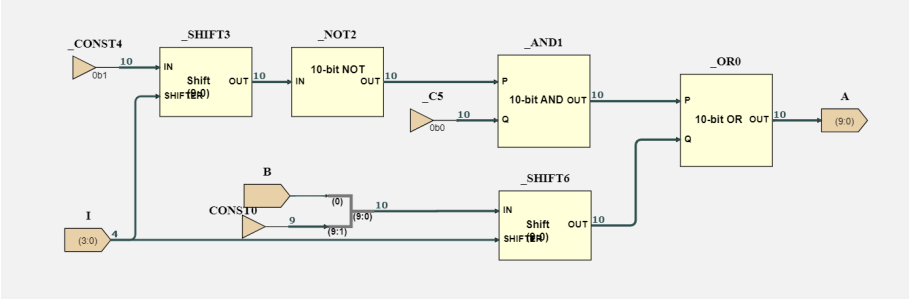
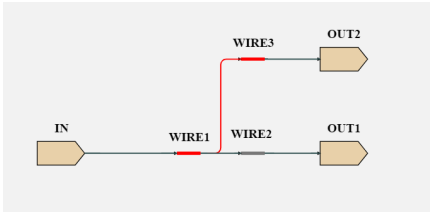


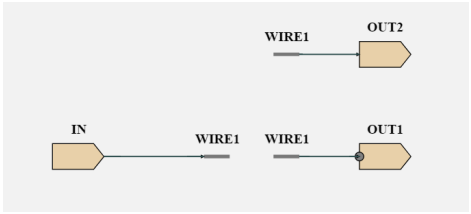
Figure 5.6: Variable bit indexing circuit

5.5.7 Consecutive wire labels

Issie does not allow two wire label components to be connected via a wire. To join the two wire labels, there must not be a wire between them, and they must have the same name to form a single net as shown in Figure 5.7.



(a) Wire labels connected by a wire gives an error



(b) Wire labels joined by giving them the same name

Figure 5.7: Wrong wire label connections and the corrected version

As the hardware generation method described in subsection 5.5.2 associates a wire label component with each variable and combinational output port declared in the input. Hence, the example in Listing 5.8 would result in the circuit shown in Figure 5.7a.

```
module m (
    input in;
    output out1, out2;
    );
    bit wire1 = in;
    assign out1 = wire1;
    assign out2 = wire2;
endmodule
```

Listing 5.8: Verilog input resulting in wire labels connected by wires

To make sure that wires labels are not connected in the generated Issie sheet the final step of hardware generation is executing graph search on the components and renaming all consecutive wire label components to share a single name and remove the wires between them. For ease of implementation, a DFS traversal was chosen to correct the generated components.

Chapter 6

Testing

6.1 Parser tests and language ambiguity

To verify the correctness of the parser, manual testing was done during the development process. The abstract syntax tree in JSON format was observed for various testcases. Manual testing in this case was sufficient, as the functionality of the error handling functions and hardware generation was extensively tested. If the AST was wrong, some of the unit tests described in section 6.2 and section 6.3 would fail.

As detailed in subsection 5.2.1, making sure the grammar had no ambiguities was crucial. Hence, for the hardware generation and semantic error handling unit tests, it was confirmed that the number of parses was exactly one.

6.2 Error handling

During the development, manual testing was done to verify that the error handling functions produced the expected error messages. Later extensive unit testing was done which consisted of:

1. Writing malformed inputs
2. Observing the error messages in the application
3. Confirming that the error locations were correct, and the error messages are as expected and legible
4. Saving the generated error information as reference output

Table 6.1 describes the number of test cases added for the different error messages[1]. For a more detailed description of the unit tests, see appendix D. By continuously testing every feature, it was easy to see if one of the already existing functionalities broke when developing new ones. Note that all 39 error handling tests pass.

Error tested	Number of test cases
Duplicate case item	3
Variable not always assigned to in always_comb	2
Case item width does not match condition width	2
Nonblocking assignment in always_comb	6
Blocking assignment in always_ff	6
Multi-driven variable	4

[1]The error handling test cases can be seen here

Using 'clk' in expression	1
'clk' output port	1
Undefined reference to 'clk'	1
Variable called 'clk'	1
Wrong clock width	1
Cycle in combinational logic	3
Integer literal does not fit in width	3
Out of bounds bit select	3
Reading then writing to a variable	2

Table 6.1: Summary of the semantic error unit tests

6.3 Hardware generation

It was also crucial to verify that in the case of well-formed inputs, the generated hardware achieves the desired behaviour. Extensive unit testing was done to test hardware generation as well as more complex integration tests were added. The unit tests in general were small Verilog modules that test a single feature, whereas the integration tests[2] include multiple features, and they represent common hardware logic, including counters, shift registers, flip-flops.

Table 6.2 describes some of the integration tests added and the features they test. Note that the hardware generation test cases were based on existing Verilog examples available on the internet [84] [85] [86].

Test case	Behaviour	Features tested
Arbiter	Generate and repeat a sequence	- always_ff - always_comb - if-else construct - case statement - logical operators
Counter	Synchronous, resetable counter	- always_ff - if-else construct
Gray counter	Synchronous, resettable Gray counter	- always_ff - continuous assignment - if-else construct - concatenation - constant bitselect - wire declaration
Magnitude comparator	Compares two 4 bit numbers	- always_comb - if-else constructs - nested sequential blocks

[2]The hardware generation tests can be seen here

SR flip flop	Flip flop	- always_ff - if-else construct - case statement
Universal shift register	Based on the input shifts left, right or parallel in parallel out	- always_ff - if-else construct - case statement - part-select - concatenation
Mux with case	2x1 Mux	- always_comb - case statement
Mux with if	2x1 Mux	- always_comb - if-else construct
Parity generator	Generate parity bit	- continuous assignment

Table 6.2: Integration tests for synthesis

The testing workflow can be seen in Figure 6.1. Every unit test and integration test consisted of a single SystemVerilog module along with a set of input values for a number of clock cycles. The tests were run with the help of a programmatically generated top level Issie sheet that contained the Verilog component as well as a counter and a ROM for every input, where the ROM contained the inputs values parsed in, see Figure 6.2. The counter output was connected to the ROM address inputs and the ROM data outputs were connected to the corresponding input of the Verilog module under test. Finally, the simulator was run on the top level sheet and the output waveforms were captured and compared to the reference outputs.

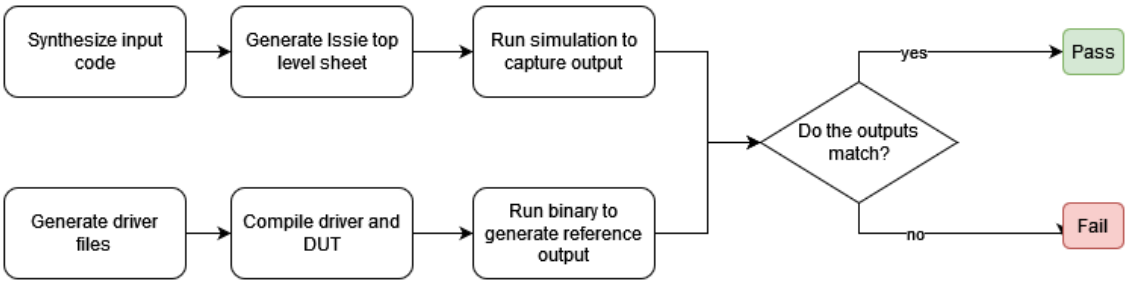


Figure 6.1: Steps of automated testing of synthesis

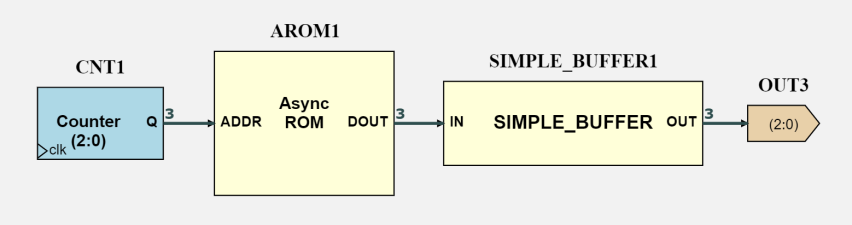


Figure 6.2: Top level sheet for testing the module 'simple_buffer'

The reference outputs were generated by Icarus Verilog [3] version 11 (the most recent stable release) with the flag `-g 2012` to use the 2012 revision of the language, enabling SystemVerilog features such as `always_comb` and `always_ff`. To drive the design under test (DUT) with the given inputs, a top level verilog testbench (or driver) was necessary for each test module. To simplify the testing, the top level sheets are automatically generated. The driver module also outputs the waveforms in a JSON format, matching the structure the Issie compiler testbench uses for capturing the simulation output. The driver and the DUT are then compiled and run by Icarus Verilog and the output waveforms are saved.

Listing 6.1 shows an example of the automatically generated top level sheet for the DUT `register`.

```
module top_module;
  // instantiating the neccessary variables
  bit [2:0] in;
  bit [2:0] in_array [9:0]; // for storing the input values at every clock cycle
  bit [2:0] out;
  bit [2:0] out_array [9:0]; // for storing the output values at every clock cycle
bit clk;
integer i_, j_;
  initial begin // this initial block generates the clock - needed for clocked
    modules
   clk=0;
    repeat(24) begin
      #1;
      clk=!clk;
    end
    // after the given clock cycles, print out the values of each output port in a
    json format:
    $display("[");
    $write("{\"Label\": \"out\", \"Values\": [");
    for(i_=0;i_<9; i_=i_+1) begin $write("%d, ", out_array[i_]); end
    $display("%d]}", out_array[9]);
    $write("]");
    $finish(0);
  end
  initial begin
      // set the input array with the given input values
      in_array[0] = 3'd7;
      in_array[1] = 3'd1;
      ...
      in_array[9] = 3'd5;
      for(j_=0; j_<10; j_=j_+1) begin
        in=in_array[j_]; // drive the DUT input port with the jth input value
        #0.5; // wait for combinational logic in the DUT to update
        out_array[j_]=out; // save the output port value
        @(negedge clk);
      end
  end
  register dut (.in(in), .out(out), .clk(clk); // instantiating the DUT
endmodule
```

Listing 6.1: Example Verilog driver for module register

Note that one of the initial blocks drives the clock and at the end of the simulation it prints out the output values; the other initial block drives the inputs - ensuring that the signals are updated in the correct order [87]. As Issie abstracts away the complications of timing in combinational and clocked blocks the timing of driving the inputs of the DUT and sampling the outputs were different to usual Verilog testbenches. In the DUT the clocked logic always updates at the positive edge of the clock, so in the testbench, sampling the output is done at the negative edge. However, to match exactly with the Issie simulation outputs, the inputs must be driven first, then a delay is needed so that the combinational logic in the DUT can update and then the output values are sampled. The `@(negedge clk)` is done after these assignments so that the very first clock cycle is captured to match the Issie simulation logic.

Module instantiation statements were also tested by the test bench by grouping the modules of a design in a single directory. However, as described in section 5.1, clocked module instantiation statements do not take in the clock signal, hence it was not possible to simulate these such tests with Icarus Verilog. Therefore, clocked module instantiation statements were tested manually.

6.4 Test results

Table 6.3 summarises the different hardware generation test cases grouped by functionality. The test cases include both combiational and clocked logic, covering all supported features. Out of the 63 test cases 61 pass.

Functionality	Tests passing	Number of tests
Counter	6	6
Adder/subtractor	8	8
Comparator	2	2
Decoder/encoder	4	4
Multiplexer	2	2
Demultiplexer	1	1
Flipflops/registers	6	6
Finite State Machine	5	5
Basic logic gates	5	7
Multiplier	5	5
ROM	2	2
Module instantiations	3	3
Other	12	12
Total	61	63

Table 6.3: Test result summary

After observing the generated hardware for the failing tests, it was observed that the generated hardware was as expected and that the two test cases failing are due to a small bug in the Issie simulator[3]. One of the test cases is a simple 1 bit inverter:

```
module jinverter(y,a);
  output bit y;
  input bit a;
  assign y=~a;
endmodule
```

This test case is failing because the width inference for an N bit inverter in the Issie simulator is wrong, and it outputs a number that is wider than it should be, resulting in different values compared to Icarus Verilog.

The other test case that does not pass is the following:

```
module three_st (t, i, o);
    input  bit t, i;
    output bit o;
    always_comb begin
        if (~t)
            o = i;
        else
            o = 1'b0;
    end
endmodule
```

In the Issie compiler, the value of `o` is always identical to the input `i`, regardless of the value of `t`. The hardware generated contains a similar circuit in Figure 6.3. This sheet inverts a 1 bit constant 1 – that should result in 0, then this is piped into a comparator, comparing to 0. As 0=0, the output should be 1, however the output of this circuit is 0. This is likely to be also caused by the N bit inverter width inference problem described above.

[3]For the GitHub issue regarding the bug, see https://github.com/tomcl/issie/issues/283

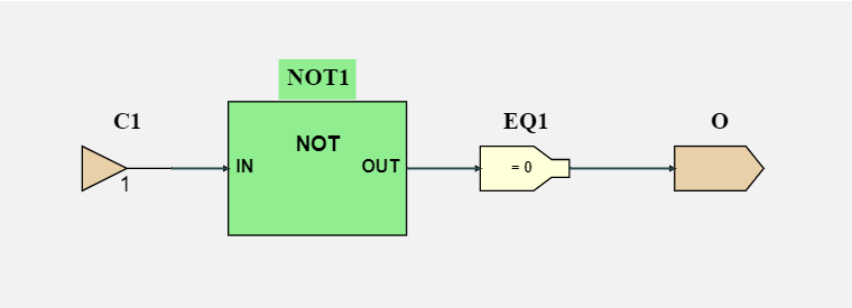


Figure 6.3: Circuit showing simulator bug

Chapter 7

Evaluation

This chapter evaluates the final product, focusing on the quality of the error messages, the compilation performance as well as the code quality.

7.1 Error messages

To evaluate the quality of the semantic or syntactic error messages, the error messages were observed for various Verilog code snippets. Some of these examples include modules containing common mistakes [88], some of them are based on the synthesis tests with some manual changes – removing or adding tokens. Finally, some of these code snippets were generated by ChatGPT [89] ensuring that the error message evaluation was independent of the feature set implemented. Section 7.1.16 creates a numerical score system and assigns a score to each example.

Each subsection below contains a code snippet with a syntax or semantic error, followed by the error message appearing at the location of the error on hover (as described in section 5.3), followed by the more detailed extra error message that appears in the error table next to the editor. Where only a single error is given, the extra error message is identical to the in-text error message.

7.1.1 `Assign` in always blocks

```
module m (input bit a, b, output bit out);
    always_comb begin
        assign out = a ^ b;
    end
endmodule
```

```
Unexpected token 'assign' at line 4 column 5. Expected: 'if', 'begin', 'case', {
    IDENTIFIER}
```

This syntax error message is very clear, the expected tokens are useful. The compiler could give more information about how to correct the error. This would require the parser to accept this input as valid, and an error checking function could provide more information.

7.1.2 Driving the same variable in multiple always blocks

```
module m(input bit a,b, output bit x);
  always_comb begin
    x = a ^ b;
  end
  always_comb begin
    x = 1'b0;
end
endmodule
```

```
Some ports of variables are driven by multiple always blocks or continuous
    assignments
```

```
The following variables are driven by multiple always blocks or continuous
    assignments: x. Please make sure every port is driven by at most one always
    block or continuous assignment.
```

The error message is clear, the extra error message is detailed. However, the error message and its location does not tell the user where the multi driven variables are.

7.1.3 Not setting a variable in all paths of an always block

```
module m(input bit xyz,abc, output bit freq_c);
  always_comb begin
    if (xyz == 4'b0010) begin
      freq_c = abc;
    end
  end
endmodule
```

```
Some ports or variables might not always be assigned to
```

```
The following variables might be undefined as they are not assigned to in every
    branch of conditional statements of case statements of the always_comb block:
    freq_c
```

The error message is clear, the explanation of the extra error message is useful. However, the compiler could suggest a solution to correct the mistake – by giving the option to insert a statement assigning a default value to the variable in question.

7.1.4 Omitting the 'bit' specifier

```
module accum (clk, clr, d, q);
  input      clk, clr;
  input  bit [3:0] d;
  output bit [3:0] q;
  bit    [3:0] tmp;
  always_ff @(posedge clk)
  begin
     if (clr)
        tmp <= 4'b0000;
     else
        tmp <= tmp + d;
  end
     assign q = tmp;
endmodule
```

```
Unexpected token "clk" at line 2 col 14. Expected: 'bit'
```

This syntax error message makes it easy for the user to understand how to correct the problem. There is no need for improvements.

7.1.5 Blocking assignment in sequential logic

```
module accum (clk, clr, d, q);
  input   bit   clk, clr;
  input  bit [3:0] d;
  output bit [3:0] q;
  bit    [3:0] tmp;
  always_ff @(posedge clk)
  begin
     if (clr)
        tmp <= 4'b0000;
     else
        tmp = tmp + d;
  end
     assign q = tmp;
endmodule
```

```
Blocking assignment in always_ff block
```

```
Blocking assignment in a clocked always block is not supported. Please use
    nonblocking assignments in clocked always blocks.
Did you mean: <=
```

The error message is very clear, and the user can solve the problem by simply clicking on the "<=" token to fix the assignment, making this error message excellent.

7.1.6 Missing semicolon

```
module counter (clk, load, d, q);
    input bit       clk, load;
    input bit [3:0] d;
    output bit [3:0] q;
    bit    [3:0] tmp;
    always_ff @(posedge clk)
    begin
       if (load)
          tmp <= d
       else
          tmp <= tmp + 1'b1;
    end
       assign q = tmp;
endmodule
```

```
Unexpected token "else" at line 10 col 8. Expected: ';', {OPERATOR}, '['
Is the previous line missing a semicolon?
```

The error message clearly suggests that a semicolon might be missing, hence the mistake is easy to correct.

7.1.7 Missing clock input

```
module counter (load, d, q);
    input bit  load;
    input bit [3:0] d;
    output bit [3:0] q;
    bit    [3:0] tmp;
    always_ff @(posedge clk)
    begin
       if (load)
          tmp <= d;
       else
          tmp <= tmp + 1'b1;
    end
       assign q = tmp;
endmodule
```

```
Variable 'clk' is not defined as an input port
```

```
To use always_ff blocks please make sure to include 'clk' in the port list.
Did you mean: input bit clk;
```

The error message is detailed enough to understand what the problem is. When the suggestion is clicked, 'clk' is added as an input port, making the error easy to correct.

7.1.8 Missing 'begin' ... 'else'

```
module jmagnitudeComparator(AEQB, AGTB, ALTB, A, B);
  output bit AEQB, AGTB, ALTB;
  input bit [3:0] A, B;
  always_comb
  begin
    if( A == B )
        AEQB = 1'b1;
        AGTB = 1'b0;
        ALTB = 1'b0;
    else if ( A > B )
      begin
        AEQB = 1'b0;
        AGTB = 1'b1;
        ALTB = 1'b0;
      end
    else
      begin
        AEQB = 1'b0;
        AGTB = 1'b0;
        ALTB = 1'b1;
```

```
      end
  end
endmodule
```

```
Unexpected token "else" at line 10 col 5. Expected: 'end', 'if', 'begin', 'case', {
    IDENTIFIER}
```

The syntax error is clear and concise. Note that it is not possible for the parser to give a more accurate error.

7.1.9 Unrecognised module in module instantiation statement

```
module m(a,b);
  input bit a;
  output bit b;
  bufer u1(.a(a), .b(b)); // Undeclared module
endmodule
```

```
Component 'bufer' does not exist
```

```
Component 'bufer' does not exist - there is no custom component or Verilog
    component with this name
Did you mean: buffer
```

The error message describes the problem clearly. If there is a module with a similar name, a suggestion appears with the possible module name. The error is user-friendly as it helps correct typos in module names.

7.1.10 Incorrect bit width

```
module IncorrectBitwidth(a,b);
  input bit [3:0] a;
  output bit [2:0] b;

  assign b = a; // Mismatched bitwidths for assignment
endmodule
```

```
The RHS expression (4 bits wide) doesn't fit in the variable on the LHS (3 bits
    wide)
```

This error message prevents users from losing data by accidentally slicing variables, reducing the probability of a bug existing in the module. The error message is clear.

7.1.11 Cycles in combinational logic

```
module InfiniteLoop();
  bit a;
  always_comb begin
    a = ~a; // Infinite loop
  end
endmodule
```

```
The following variables form a dependency cycle: [a[0];a[0]]
```

This error message appears at the end of the module and not at the affected variables, hence this error message could be improved. It might not be obvious for a beginner what the problem is and how to best fix it from the error message.

7.1.12 Redefined signal

```
module RedefinedSignal();
  bit a;
  bit a; // Signal 'a' is redefined
  assign a = 1'b0;
endmodule
```

```
Identifier 'a' is already used by another variable.
Please use a different name for this variable.
```

Helpful error message that describes the problem in an easy-to-understand manner.

7.1.13 Missing 'endmodule'

```
module ModuleWithoutEndmodule;
  bit a;
  always_ff @(posedge clk) begin
    a = ~a;
  end
  // Missing 'endmodule'
```

```
Unexpected end of input. Missing endmodule?
```

Note that in the legacy compiler, this threw an uncaught exception. The error message clearly states that the input is incomplete and tells the user that the endmodule token might be missing, making the mistake easy to correct.

7.1.14 Invalid sensitivity list in `always_ff`

```
module ModuleWithIncorrectSensitivityList();
  bit clk, reset;
  bit [3:0] count;
  always_ff @(reset) begin // Error: Incorrect sensitivity list, should be posedge clk
    if (reset)
      count <= 0;
    else
      count <= count + 1;
  end
endmodule
```

```
Unexpected token "reset" at line 4 col 15. Expected: 'posedge'
```

This syntax error gives a clear indication of what token the user should write next in the sensitivity list.

7.1.15 Duplicate port in module instantiation

```
module TopModule(output bit [1:0] out);
  bit [1:0] a, b;
  buffer inst (.in(a), .in(b), .out(out)); // Error: Non-unique port names
endmodule
```

```
Duplicate port
```

```
Duplicate port name 'in' for module buffer
```

The error contains all the details necessary for correcting the mistake.

7.1.16 Overall comments on error messages

To evaluate the overall quality of the error messages, a score system from 1-5 was created and each example above was given a score based on the helpfulness of the error message. Table 7.1 describes the score system and Table 7.2 contains the scores assigned to each error message discussed above. Note that an error message assigned a score of 5+ means that the quality of the error message was outstanding.

The mean score of the errors was 4.67, suggesting that based on the examples above, most syntactic and semantic error messages provide useful information to the users, enabling beginner and advanced users to work in Verilog efficiently. The compiler occasionally gives suggestions that allow the user to correct the error with a single button click. Some of the error messages shown could be improved but overall, the error messages are all of high quality and are explained in a way that makes them easy to correct, meeting the Issie principles, see chapter 1. User testing was not done as part of the evaluation, as the user interface of the compiler did not change during the project.

Score	Meaning
5	Easy to understand and correct and/or there is no way to give a better error message
4	The error message was useful, but there are some minor improvements possible
3	Gives some detail about the error but difficult to resolve
2	Gives no information about the error
1	Misinforms the user

Table 7.1: Error score rating system

Example	Score
Assign in always block	4
Driving the same variable in multiple always blocks	4
Not setting a variable in all paths of an always block	4
Omitting the 'bit' specifier	5
Blocking assignment in sequential logic	5+
Missing semicolon	5
Missing clock input	5+
Missing 'begin' ... 'else'	5
Unrecognised module in module instantiation statement	5+
Incorrect bit width	5
Cycles in combinational logic	3
Redefined signal	5
Missing 'endmodule'	5
Invalid sensitivity list in always_ff	5
Duplicate port in module instantiation	5
Mean	4.67

Table 7.2: Error message scores

7.2 Compiler performance

Evaluating the compiler's performance was also of great importance: fast compilation is necessary for a seamless user experience. Figure 7.1 describes the time needed for parsing, error detection and synthesis for the tests detailed in section 6.3. The performance tests were executed on a 12th Gen Intel® Core™ i7-12700H 2.30 GHz CPU. The graphs show a roughly linear trend between the time taken and the number of lines in the input (but this might not be the case for more complex modules, parsing with Nearley can have quadratic worst case time complexity as seen in subsection 2.3.6). For all the test cases the total compilation time is below 35ms which is sufficiently small. Parsing and error checking is done per character written in the input, in total these take less than 25ms. Productive typing speed is 300 characters per minute [90], equivalent to one character per 200ms, which is much greater than the parse and error detection time, which also means that the parsing speed will be fast enough for much larger inputs. Based on the linear relationship between both the parse time and the error check time relative to the input length, it

is estimated that blocks up to 1000 lines can be parsed within 200ms. Synthesis takes up to 12ms for programs of size 0–120 lines, which is much below the 250ms average human reaction time to visual stimulus [91]. As the synthesis is only done once the user clicks the update button, the synthesis seems nearly immediate for the users.

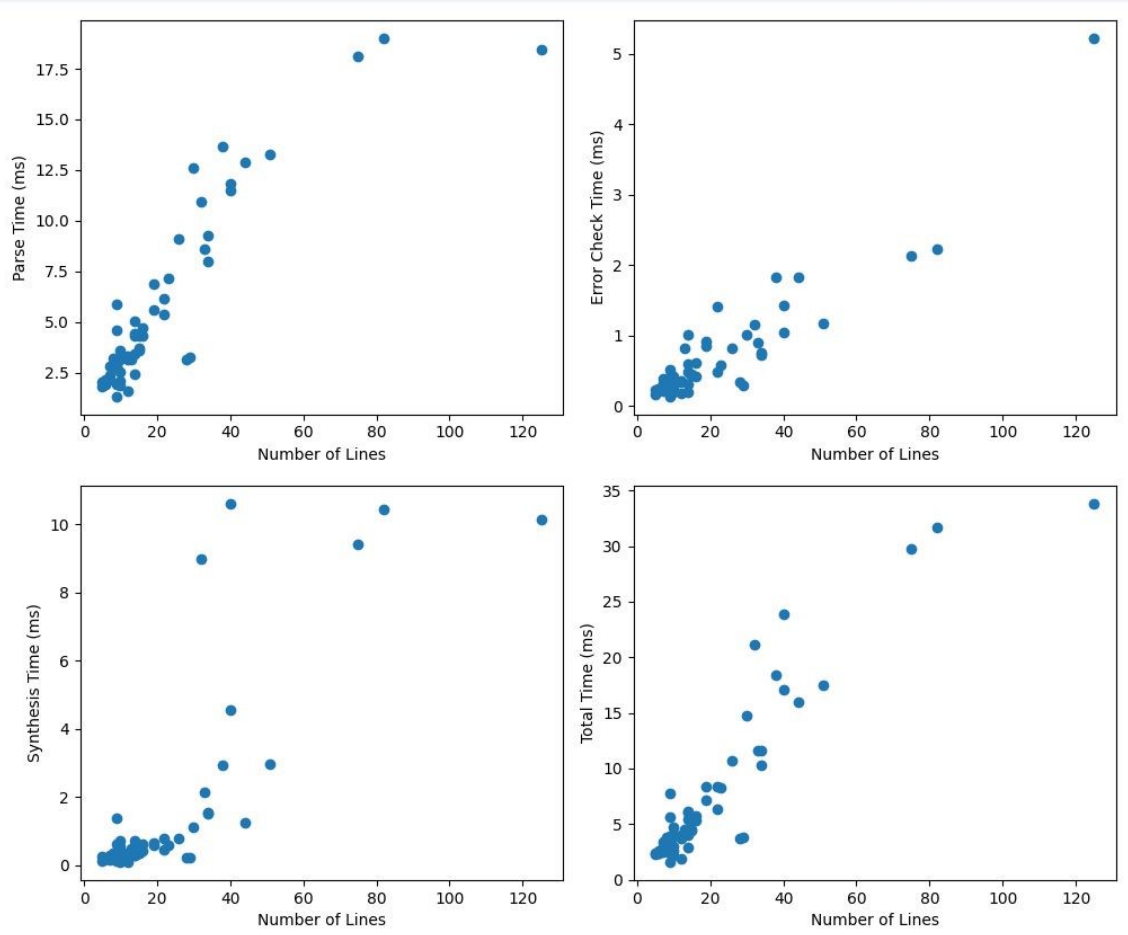


Figure 7.1: Time needed for parsing, error detection and synthesis with respect to the number of lines in the input

Note that based on Figure 7.1 the synthesis time exhibits irregular patterns. This behaviour can be attributed to the fact that the module's complexity is not solely determined by the number of lines it contains. Figure 7.2 displays the relationship between the synthesis time and the number of components generated, showing a linear correlation.

7.3 Code quality

As the Issie compiler is expected to be further developed by other students, ensuring the compiler's maintainability and extensibility was a crucial aspect of the project. This section evaluates the quality of the delivered code by comparing them to the Issie coding guidelines [92].

7.3.1 Code quality and the Issie guidelines

Types

Using appropriate types in the compiler is important as it affects the entire code base. Records were used to model the nodes of the abstract syntax tree, encapsulating the pieces of data that belong together: for example the condition, the if statement and the else statement in a conditional statement. DUs were used to represent the different cases of the general AST node, which allowed for easy to read `match` statements in the error detection and hardware generation functions. Optional

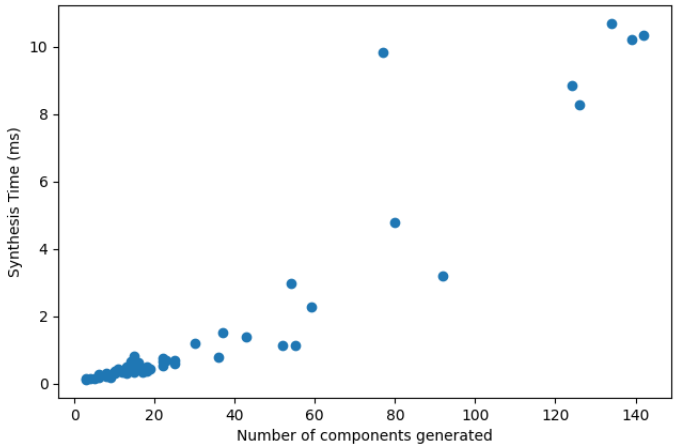


Figure 7.2: Time needed for parsing with respect to the number of components generated

fields were used in the AST nodes where it made sense to use them, such as representing the default statement in a case statement. The new types added during the project do not introduce more mutually recursive types. In the entire compiler, no mutable data was used, preventing potential bugs. The built-in List and Array types were used in the AST and in the functions operating on the AST, whereas the Map type was used in some cases for better lookup performance, for example to represent the dependency graph for checking for cycles in combinational logic.

Names

Most functions and variables in the compiler were given meaningful names, that are descriptive but not too long, making the code concise and readable. The code base uses pipes extensively to break up operations into smaller steps for legibility.

Functions

Library functions, such as the List and Map functions, were used in the compiler as much as possible and anonymous functions were used when the function is small and just reading it gives enough information to understand it. Higher order functions were used across error handling and hardware generation. One of the most notable functions used frequently was the `foldAST` function that recursively iterates through the given AST node, calling the function passed in as a parameter on each child node in the tree. The use of this and many other functions abstracted away a lot of common logic, making the code in the main error handling and hardware generation functions easy to read. Using recursion was necessary to handle the AST, but the library `map`, `collect` and `fold` functions were used instead of recursion where possible. As users can input malformed data, it was crucial to use `tryFind` instead of `find` and match statements on Options instead of `Option.get`, making the compiler robust.

Documentation

According to the Issie guidelines, the best way to document F# code is by adding XML comments, which is why the most important functions in error handling and synthesis have XML comments describing their functionality in a couple of lines. Furthermore, the Issie wiki has been extended with necessary information about the final compiler[1].

7.3.2 Extensibility

This section evaluates the extensibility of the compiler by rating the difficulty of adding some of the desirable features mentioned in section 4.1 that have not been implemented yet. For detailed steps on adding new features, see the compiler documentation.

[1]See: https://github.com/tomcl/issie/wiki/Verilog-Component

Adding for loops

To add for loops to the compiler, the grammar must be modified. This is straightforward, as the entire SystemVerilog grammar is available in the SystemVerilog language standard [22]. To support for loops inside an always block, a new type of statement node must be added to the AST records. Before the error handling functions are called, the for loops must be checked for semantic correctness. If they are correct, the for loop must be unrolled into a list of statements and then the error handling and the hardware generation can be called without needing any changes in the code.

Supporting multiple modules per block

Updating the grammar and the AST to support multiple modules in a single file is very simple: using the Nearley `:+` EBNF operator [93] in the grammar would allow the description of 1 or modules in a single code block. The AST would have to be modified to contain a list of modules instead of a single module. Before the start of the error handling, the hierarchy of the modules must be established, determining the root node and the dependencies using graph search as well as detecting any errors in the module hierarchy, such as cycles or multiple root nodes. This should also be straightforward to do, as the general DFS graph search function used for detecting cycles in combinational logic can be used to detect cycles in the module dependencies. After the hierarchy is established, the error check functions can be called on each module in the block, requiring no further update to the error handling. The sheet creation needs a similar update, where the hardware for each module must be generated and the logic for handling module instantiation statements must be modified as the modules in the open editor are not loaded component, so the sheet creator must generate a flat structure out of the AST.

Module parameters – multiple modules per block

After multiple modules per block are implemented, adding module parameters is also straightforward. The main error handling function and the sheet creator functions must take in the list of module parameters and their values, then the error check and sheet create functions must use the values of these constants where necessary.

7.3.3 Bug fixing and adding tests

During the testing phase, a few minor bugs were identified which were fixed without difficulty, showing the maintainability of the code. Furthermore, adding testcases is also straightforward: the developer can simply add a Verilog file into the test case input directory, along with a JSON file that contains the values of the input ports of the module at each clock tick. Then, the driver file is automatically generated, and the reference outputs are created by Icarus Verilog.

7.3.4 Overall thoughts on maintainability and extensibility

The compiler code base uses a lot of good functional programming practices and it clearly meets the Issie coding guidelines, allowing for a legible codebase. By adding a lexer to the compiler as well as a fully automated test-bench, the maintainability of the compiler was greatly improved. The numerous test cases will ensure in the future that any changes to the code do not inadvertently break the existing functionality. The compiler's extensibility is demonstrated by the fact that potential future features described above can be implemented with minimal development effort, thanks to the extensive reusability of the existing code.

7.4 Reflection on the requirements

This section revisits the project requirements laid out in chapter 3, referring to the identifiers (Essential and Desired) of the initial objectives.

E1 The exact subset of Verilog the compiler supports was chosen as described in section 4.1, considering the Issie principles

E2-E3 The final compiler supports both clocked and combinational always blocks. Combinational always blocks give users more freedom when describing combinational logic compared to using simple continuous assignment statements. Sequential always blocks enable users to describe clocked logic, which is crucial as most real world hardware projects are clocked. Always blocks allow for easy implementation of various hardware blocks including counters, registers, flip-flops.

E4 Blocking assignments in combinational always blocks and non-blocking assignments in clocked always blocks are supported, ensuring that the generated hardware is a mix of combinational and D flip-flop type logic, not allowing the description of unexpected pieces of hardware, such as latches.

E5 Sequential blocks have been added, as these are necessary for describing complex circuits in SystemVerilog.

E6-E7 If statements and case statements were implemented, making it possible for users to programmatically describe encoders, decoders, multiplexers, demultiplexers, resettable logic as well as comparators. Case statements in combinational always blocks allow users to implement read only memories and finite state machines.

E8 The error messages added are informative and conform to the Issie principles, with occasional suggestions for automatic semantic error correction.

E9 The parser is implemented with the correct operator associativity, the bug from the legacy compiler has been fixed, and the new operators also follow the correct associativity.

E10 The compiler is mostly compatible with SystemVerilog which is proven by the comparison with Icarus Verilog simulation outputs. Clocked module instantiation statements are not compatible with the language standard as seen in section 5.1, but every other supported feature is.

D1 Modules can be instantiated within Verilog blocks, allowing for hierarchical designs. The module instantiation statements allow users to include both Issie and Verilog modules within Verilog blocks. This means that every feature that is available in Issie schematic editor is also available to use in the Issie Verilog compiler. For example, arrays are not supported by the compiler, but by instantiating a RAM component in Verilog, the users can have a RAM in Verilog modules as well.

D2 A fully automated test bench was implemented which compares the Issie simulation outputs to Icarus Verilog outputs. Extensive testing was done to ensure the functional correctness and maintainability of the compiler.

D3 The syntax error messages were greatly improved by having added a tokenizer to the compiler

D4 Variable single bit select is supported. Variable bus select has not been implemented.

D5 The compiler is extensible as seen in the examples in subsection 7.3.2 and maintainability is ensured by adhering to the Issie coding guidelines as well as by developing reusable, higher order functions such as `foldAST`. The maintainability of the parser was improved by the addition of the lexer.

D6 The grammar ambiguities of the legacy compiler were removed, and the dangling if-else problem is avoided by the parser. The syntactically valid test cases were inspected for ambiguity, and none of them had multiple possible parses.

D7 The `==`, `!=`, `<`, `<=`, `>`, `>=` to allow the end users to write meaningful if and case statements.

D8 Parsing, error detection and hardware generation are significantly faster than the threshold of human perception (see section 7.2), enabling a frustration-free user experience

In conclusion, the compiler successfully meets all the core requirements, E1-E10. The desirable features D2-D8 have also been implemented. Desirable feature D1 has been partially met, with the implementation of module instantiation statements. The future implementation of for loops and module parameters are discussed in section 8.1. The implemented features conform to the Issie

principles: the error messages are easy to understand and correct, see section 7.1 and the final feature set provides a SystemVerilog IDE and synthesizer that is easy to use for both beginner and advanced users. The compiler was found to be sufficiently fast and the correctness of the compiler was thoroughly tested, hence the compiler can be merged into the main branch of Issie and used for teaching SystemVerilog for the first year students.

Chapter 8

Conclusions and further work

In conclusion, throughout this project the Issie SystemVerilog compiler was enhanced by providing support for behavioural Verilog – both clocked and combinational always blocks, as well as structural Verilog. The main focus of the project was adding meaningful error messages in an effort of making Issie easier to use for students than third party synthesizers and simulators such as Icarus Verilog or Quartus. This was ensured through the evaluation of the error messages against the Issie principles, whereas the functional correctness of the entire compiler was guaranteed by extensive testing.

One of the most remarkable actions undertaken in this project was adding a lexer before the parser (see section 4.4) which significantly improved the syntax errors and ameliorated maintainability of the code-base. The improved syntax errors facilitated the compiler's adherence to the Issie principles.

Another significant part of the project was implementing the hardware generation test-bench. By automating these tests, the development and testing of new features became extremely simple. Comparing the Issie simulation outputs to Icarus Verilog outputs did not only check the correctness of the SystemVerilog compiler but also of a large part of the Issie simulator. The extensive testing led to the discovery of a few bugs in the simulator, as seen in section 6.4. More importantly, the testing confirmed that the Issie simulator's behaviour is correct in most cases as it aligns with the third party simulator.

Finally, another notable development detail of the compiler was the implementation of graph algorithms, described in subsection 5.4.2 and section about consecutive wire labels. Implementing depth first search for detecting cycles in combinational logic was a challenging but interesting task to do in a functional programming language.

As every software project has its limitations, the Issie Verilog compiler is no exception. One of the main limitations of the compiler is a bug in the editor: when the lines in the input are too long to fit in the box, the red underlines of the error messages get misaligned. As the focus of the project was mostly on parsing and error handling, making significant changes to the user interface was outside the scope of the project. Furthermore, the hardware generated by the compiler is not at all optimised, resulting in circuits of over 100 components in some cases for a single Verilog module. Although this did not cause any performance or other issues in the simulator for the test cases analysed, in the future when more complex Verilog structures are supported, optimisation of the generated hardware might be necessary for responsive simulation of Verilog components.

All the core requirements in chapter 3 have been met, enabling users to implement various hardware components in SystemVerilog including multiplexers, counters, finite state machines, registers and arithmetic-logic units using `always_comb` and `always_ff` constructs, conditional and case statements. Using module instantiation statements the students can use any Issie custom component or other Verilog component within a Verilog module facilitating the (mostly) programmatic implementation of complex digital circuits such as CPUs. section 8.1 details the features future students can implement to further enhance the Issie SystemVerilog compiler.

8.1 Future work

One very powerful feature to be implemented later is for loops with module parameters. By allowing multiple module parameters in a single Verilog block, the development of module parameters

in a single block is straightforward and the high level steps of implementation are described in subsection 7.3.2. These features would enable users to create templated modules as well as effectively minimize code repetition within Verilog designs. Another important feature the compiler should support is variable bus select as well as arrays so that users can easily implement general ROM and RAM components. To add arrays and variable bus select, the grammar must be slightly modified, as well as the AST. Furthermore, certain error handling functions have to be modified. Hardware generation requires a minor change, which is straightforward to do.

Parallel to this project, a fast Issie simulator has been implemented by final year student Yujie Wang at Imperial College London (A high performance digital circuit simulator for ISSIE [1]). To improve the simulation speed of Verilog components, merging the two projects is essential.

As described in section 4.6, it could be beneficial for students to look at the underlying hardware of their Verilog designs. Currently, the Issie sheets generated by the compiler are very hard to understand, as the components are placed arbitrarily and wires are not routed intelligently. Consequently, future work should encompass the implementation of an Issie sheet beautifier, as well as the addition of a UI option to view the generated sheet.

Furthermore, to allow the Issie compiler to be used by a wide range of students or other individuals, implementing SystemVerilog assertions is of great importance. By incorporating assertions, Issie could become the primary design and test benching software for various modules within the Electrical Engineering department at Imperial College London. These modules, including the year two Instruction Set Architectures, would greatly benefit from utilizing Issie in their coursework. A team of students taking the 2023 High Level Programming module added assertions to Issie[2]. Merging this functionality into Issie would allow for an easy implementation of SystemVerilog assertions by modifying the sheet creator to instantiate assertion components in the generated sheet in the case of assertions in the Verilog test bench.

8.2 Summary

After considering the accomplishments, evaluation, and constraints of the undertaken work, it can be concluded that this project was successful, as evidenced by the enhanced functionality of Issie resulting from the compiler implementation.

The implementation of the project can be found here, with detailed documentation here.

[1] See: https://github.com/tomcl/issie/tree/yw2919-simulator/version2

[2] The repository is found here: https://github.com/Jpnock/hlp23-team3/blob/team3/README-FEATURES-TEAM3.md

Appendix A

Expression bit lengths

Table A.1 describes the bit lengths resulting from different types of expressions. Note that *L(i)* represents the bit length of operand *i*.

Expression	Bit length	Comments
Unsized constant number	Same as integer	
Sized constant number	As given	
i op j, where op is: `+ - * / % & | ^ ^~ ~^`	max(L(i),L(j))	
op i, where op is: `+ - ~`	L(i)	
i op j, where op is: `=== !== == != > >= < <=`	1 bit	Operands are sized to max(L(i),L(j))
i op j, where op is: `&& || -> <->`	1 bit	All operands are self-determined
op i, where op is: `& ~& | ~| ^ ~^ ^~ !`	1 bit	All operands are self-determined
i op j, where op is: `>> << ** >>> <<<`	L(i)	j is self-determined
`i ? j : k`	max(L(j),L(k))	i is self-determined
`{i,...,j}`	L(i)+..+L(j)	All operands are self-determined
`{i{j,..,k}}`	i × (L(j)+..+L(k))	All operands are self-determined

Figure A.1: Bit lengths resulting from self-determined expressions [22]

Appendix B

FoldAST

```
/// Recursively folds over an ASTNode, calling folder at every level. Only explores parts where there are multiple possibilities within a Node
let rec foldAST folder state (node:ASTNode) =
    let state' = folder state node
    match node with
    | VerilogInput input ->
        foldAST folder state'  (Module(input.Module))
    | Module m ->
        foldAST folder state' (ModuleItems(m.ModuleItems))
    | ModuleItems items ->
        items.ItemList
        |> Array.map (fun item -> Item(item))
        |> Array.fold (foldAST folder) state'
    | Item item ->
        foldAST folder state' (getItem item)
    | AlwaysConstruct always ->
        foldAST folder state' (Statement(always.Statement))
    | Statement statement ->
        statement
        |> getAlwaysStatement
        |> statementToNode
        |> foldAST folder state'
    | SeqBlock seqBlock ->
        seqBlock.Statements
        |> Array.map (fun s -> Statement(s))
        |> Array.fold (foldAST folder) state'
    | Case case ->
        let newState = foldAST folder state' (Expression(case.Expression))
        let newState' =
            case.CaseItems
            |> Array.map (fun item -> CaseItem(item))
            |> Array.fold (foldAST folder) newState
        match case.Default with
        | Some stmt -> foldAST folder newState' (Statement(stmt))
        | _ -> newState'
    | CaseItem caseItem ->
        let newstate =
            caseItem.Expressions
            |> Array.map (fun expr -> Number expr)
            |> Array.fold (foldAST folder) state'
        foldAST folder newstate (Statement(caseItem.Statement))
    | Conditional cond ->
        let tmpState =
            IfStatement(cond.IfStatement)
            |> foldAST folder state'
        match cond.ElseStatement with
        | Some elseStmt -> List.fold (foldAST folder) tmpState [Statement(elseStmt)]
        | _ -> tmpState
    | ContinuousAssign assign ->
        foldAST folder state' (Assignment(assign.Assignment))
    | Assignment assign ->
        (foldAST folder state' (AssignmentLHS(assign.LHS)), (Expression(assign.RHS)))
```

```
52          ||> foldAST folder
53      | NonBlockingAssign nonblocking ->
54          foldAST folder state' (Assignment(nonblocking.Assignment))
55      | BlockingAssign blocking ->
56          foldAST folder state' (Assignment(blocking.Assignment))
57      | IfStatement ifstmt ->
58          (foldAST folder state' (Expression(ifstmt.Condition)), (Statement(ifstmt
        .Statement)))
59          ||> foldAST folder
60      | AssignmentLHS lhs ->
61          match lhs.VariableBitSelect with
62          | Some expr ->
63              foldAST folder state' (Expression(expr))
64          | _ -> state'
65      | _ ->
66          state'
```

Listing B.1: The implementation of the foldAST higher order function

Appendix C

Semantic error messages

Table C.1 contains a list of the error messages implemented by the legacy compiler.

Error reason	Error and extra error message	Suggestion	Location
Duplicate port	"Ports must have different names" "Name x has already been used for a port. Please use a different name"		
No direction given for port	"Port x is not declared either as input or output" "Port x must be declared as input or output"	Insert 'input bit x;' or 'output bit x;'"	
Unused input port	"Variable x is defined as an input port but is not used" "Variable x is defined as an input port but is not used. Please delete it if it is not needed"		Location of x
Port missing from module header	"Port x is not defined as a port in the module declaration" "Port x is not defined as a port. Please define it in the module declaration"		Location of x
Duplicate port definition	"Port x is already defined" "Port x is already defined"		Location of x
Incorrect range in declaration	"Wrong width declaration" "A port's width can't be '[a:b]'. Correct form: [X:0]"		Location of the range
Module name already exists	"A sheet/component with that name already exists" "Module Name must be different from existing Sheets/Components"		Module name
Verilog component's name modified	"Verilog component's name cannot be changed " "Module Name of Verilog component cannot be changed"	Replace to old name	Module name
Unassigned output ports	"All output ports must be assigned" "The following ports are declared but not assigned: <list of ports>"	Insert assign x = 1'b0;	endmodule

Variable already defined	"Variable x is already used by a port" "Variable x is declared as an input/output port. Please use a different name for this wire"		x
Wrong width in declaration	"Wrong width declaration" "A port's width can't be '[a:b]'. Correct form: [X:0]"		Range
Out of bounds index in expr.	"Out of bounds index/range for variable x" "Variablex is defined as <range>". Therefore, <wrong range> is invalid."		range
Undefined variable in expr	"Variable x is not declared as an output port" "Variable x is not declared as an output port"	output bit x;	x
Variable declared after it is used	"Variable x is defined after this assignment" "Move the definition of variable x above this line"		x

Table C.1: Summary of semantic error messages, mainly from the legacy compiler

Appendix D

Semantic error message unit tests

Test case	Error tested
always_comb_case_repeated	Duplicate case item
always_comb_case_vars_undef	Variable not always assigned to in always_comb block
always_comb_case_wrong_width	Case item width does not match condition width
always_comb_if_vars_undef	Variable not always assigned to in always_comb block
always_comb_nonblocking_case_default	Nonblocking assignment in always_comb
always_comb_nonblocking_case	Nonblocking assignment in always_comb
always_comb_nonblocking_cond_else	Nonblocking assignment in always_comb
always_comb_nonblocking_cond	Nonblocking assignment in always_comb
always_comb_nonblocking_seqblock	Nonblocking assignment in always_comb
always_comb_nonblocking	Nonblocking assignment in always_comb
always_comb_repeated_case	Duplicate case item
always_ff_blocking_case_default	Blocking assignment in always_ff block
always_ff_blocking_case	Blocking assignment in always_ff block
always_ff_blocking_cond_else	Blocking assignment in always_ff block
always_ff_blocking_cond	Blocking assignment in always_ff block
always_ff_blocking_seqblock	Blocking assignment in always_ff block
always_ff_blocking	Blocking assignment in always_ff block
always_ff_case_wrong_width	Case item width does not match condition width
always_ff_repeated_case	Duplicate case item
always_ff_vars_driven_sim	Multi-driven variable
always_ff_vars_driven_sim2	Multi-driven variable

always_ff_vars_driven_sim3	Multi-driven variable
always_ff_vars_driven_sim4	Multi-driven variable
clk_expression	Using 'clk' in expression
clk_output	'clk' output port
clk_undefined	Undefined reference to 'clk'
clk_variable	Variable called 'clk'
clk_wrong_width	Wrong clock width
cycle_always_comb	Cycle in combinational logic
cycle_cont_assign_always	Cycle in combinational logic
cycle_cont_assign	Cycle in combinational logic
expression_0_number_width	Integer literal with width 0 in expression
expression_out_of_bounds	Out of bounds bit select
expression_wrong_number_width	Integer literal does not fit in width
expression_wrong_number_width2	Integer literal does not fit in width
expression_wrong_number_width_rhs	Out of bounds bit select
expression_wrong_number_width_rhs2	Out of bounds bit select
variable_written_after_read	Reading then writing to a variable
variable_written_after_read2	Reading then writing to a variable
variable_written_twice	Variable written twice in always_ff

Table D.1: Unit tests for semantic errors

Bibliography

[1] T. Clarke, "Home | ISSIE." [Online]. Available: https://tomcl.github.io/issie/

[2] "Home - nearley.js - JS Parsing Toolkit." [Online]. Available: https://nearley.js.org/#features

[3] S. Williams, "The ICARUS Verilog Compilation System," Jun. 2023, original-date: 2008-05-12T16:57:52Z. [Online]. Available: https://github.com/steveicarus/iverilog

[4] "GTKWave." [Online]. Available: https://gtkwave.sourceforge.net/

[5] "Top 4 HDL Simulators for Beginners | HackerNoon." [Online]. Available: https://hackernoon.com/top-4-hdl-simulators-for-beginners

[6] "FPGA Design Software - Intel® Quartus® Prime." [Online]. Available: https://www.intel.com/content/www/uk/en/products/details/fpga/development-tools/quartus-prime.html

[7] "F# Software Foundation." [Online]. Available: https://fsharp.org/

[8] ElectronJs, "Build cross-platform desktop apps with JavaScript, HTML, and CSS | Electron." [Online]. Available: https://electronjs.org/

[9] "About." [Online]. Available: https://nodejs.org/en/about

[10] "Chromium." [Online]. Available: https://www.chromium.org/Home/

[11] "Fable - Write Front-End Apps For The Web In F#." [Online]. Available: https://www.i-programmer.info/news/87-web-development/14969-fable-write-front-end-apps-for-the-web-in-f.html

[12] ".NET | Build. Test. Deploy." [Online]. Available: https://dotnet.microsoft.com/en-us/

[13] "npm About." [Online]. Available: https://www.npmjs.com/about

[14] cartermp, "What is F#," Oct. 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp/what-is-fsharp

[15] "Introduction to the 'Why use F#' series | F# for fun and profit." [Online]. Available: https://fsharpforfunandprofit.com/posts/why-use-fsharp-intro/

[16] "Elmish · Elmish." [Online]. Available: https://elmish.github.io/elmish/

[17] S. Forkmann, "UI programming with Elmish in F# | Compositional IT," 2017. [Online]. Available: https://www.compositional-it.com/news-blog/ui-programming-with-elmish-in-f/

[18] Z. Ajad, "The Elm Architecture - The Elmish Book." [Online]. Available: https://zaid-ajaj.github.io/the-elmish-book/#/chapters/elm/the-architecture

[19] "About Scaling Model-View-Update." [Online]. Available: https://thomasbandt.com/scaling-model-view-update

[20] ChipVerify, "SystemVerilog Tutorial." [Online]. Available: https://www.chipverify.com/systemverilog/systemverilog-tutorial

[21] Duolos, "What is Verilog?" [Online]. Available: https://www.doulos.com/knowhow/verilog/what-is-verilog/

[22] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb. 2018, conference Name: IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012).

[23] "Verilog Data Types." [Online]. Available: https://www.chipverify.com/verilog/verilog-data-types

[24] "SystemVerilog logic and bit." [Online]. Available: https://www.chipverify.com/systemverilog/systemverilog-data-types-logic-bit

[25] Morgan, "Answer to "Difference of SystemVerilog data types (reg, logic, bit)"," Nov. 2012. [Online]. Available: https://stackoverflow.com/a/13285242

[26] "Verilog Predefined Types." [Online]. Available: https://redirect.cs.umbc.edu/portal/help/VHDL/verilog/types.html

[27] "Expression Bit Widths." [Online]. Available: http://yangchangwoo.com/podongii_X2/html/technote/TOOL/MANUAL/21i_doc/data/fndtn/ver/ver4_4.htm

[28] ChipVerify, "Verilog Assignments." [Online]. Available: https://www.chipverify.com/verilog/verilog-assignments

[29] ——, "Verilog assign statement." [Online]. Available: https://www.chipverify.com/verilog/verilog-assign-statement

[30] S. Chavan, "So it's the initial or the final block in SystemVerilog?" Oct. 2022. [Online]. Available: https://medium.com/@_suyash/so-its-the-initial-or-the-final-block-in-systemverilog-bcf06688d0b6

[31] ChipVerify, "Verilog always block." [Online]. Available: https://www.chipverify.com/verilog/verilog-always-block

[32] S. Arar, "Describing Combinational Circuits in Verilog - Technical Articles," 2019. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/describing-combinational-circuits-in-verilog/

[33] Russell, "Blocking and Nonblocking Assignments in Verilog," Jun. 2022. [Online]. Available: https://nandland.com/blocking-vs-nonblocking-in-verilog/

[34] J. Yu, "SystemVerilog always_comb, always_ff," Oct. 2015. [Online]. Available: https://www.verilogpro.com/systemverilog-always_comb-always_ff/

[35] Xilinx, "Variable Part Selects Verilog Coding Example • Vivado Design Suite User Guide: Synthesis (UG901) • Reader • Documentation Portal," 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis/Variable-Part-Selects-Verilog-Coding-Example

[36] ——, "Structural Verilog Features." [Online]. Available: http://www.csit-sun.pub.ro/courses/Masterat/Materiale_Suplimentare/Xilinx%20Synthesis%20Technology/toolbox.xilinx.com/docsan/xilinx4/data/docs/xst/verilog4.html

[37] ChipVerify, "Verilog Module Instantiations." [Online]. Available: https://www.chipverify.com/verilog/verilog-module-instantiations

[38] Sidhartha, "Port Mapping for Module Instantiation in Verilog," Feb. 2016. [Online]. Available: https://www.vlsifacts.com/port-mapping-for-module-instantiation-in-verilog/

[39] S. Sutherland and S. Hdl, "Standard Gotchas," 2006.

[40] J. Yu, "Verilog reg, Verilog wire, SystemVerilog logic. What's the difference?" May 2017. [Online]. Available: https://www.verilogpro.com/verilog-reg-verilog-wire-systemverilog-logic/

[41] "SystemVerilog Coding Guidelines — Open SoC Debug 0.1 documentation." [Online]. Available: https://opensocdebug.readthedocs.io/en/latest/04_implementer/styleguides/systemverilog.html

[42] user3624, "Why are inferred latches bad?" tex.eprint: https://electronics.stackexchange.com/q/38650 tex.howpublished: Electrical Engineering Stack Exchange. [Online]. Available: https://electronics.stackexchange.com/q/38650

[43] N. Chomsky, "On Certain Formal Properties of Grammars," 1959.

[44] M. Johnson and J. Zelenski, "Formal Grammars," 2012. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/080%20Formal%20Grammars.pdf

[45] "Introduction To Grammar in Theory of Computation," Jan. 2021, section: GATE CS. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-grammar-in-theory-of-computation/

[46] "Chomsky Classification of Grammars." [Online]. Available: https://www.tutorialspoint.com/automata_theory/chomsky_classification_of_grammars.htm

[47] R. Sheldon, "What is a compiler?" 2022. [Online]. Available: https://www.techtarget.com/whatis/definition/compiler

[48] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson Education, Inc, 2006.

[49] J. Wickerson, "Lecture 1: Introduction to Compilers," Imperial College London, 2021.

[50] P. Lefebvre, "Compilers 101 - Overview and Lexer," 2018. [Online]. Available: https://dev.to/lefebvre/compilers-101---overview-and-lexer-3i0m

[51] Microsoft, "Production Rule," 2018. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/windows/desktop/indexsrv/production-rule

[52] "Difference between Top down parsing and Bottom up parsing," May 2019, section: Compiler Design. [Online]. Available: https://www.geeksforgeeks.org/difference-between-top-down-parsing-and-bottom-up-parsing/

[53] "Types of Parsers in Compiler Design," Jul. 2019, section: Compiler Design. [Online]. Available: https://www.geeksforgeeks.org/types-of-parsers-in-compiler-design/

[54] "Working of top down parser," Aug. 2020, section: Compiler Design. [Online]. Available: https://www.geeksforgeeks.org/working-of-top-down-parser/

[55] "Context-Free Grammar Introduction." [Online]. Available: https://www.tutorialspoint.com/automata_theory/context_free_grammar_introduction.htm

[56] T. Neha, "What is Top-Down Parsing? Definition, Problems, Backtracking, & Types," Jan. 2022. [Online]. Available: https://binaryterms.com/top-down-parsing-in-compiler-design.html

[57] D. Grune and C. J. H. Jacobs, *Parsing techniques: a practical guide*, 2nd ed., ser. Monographs in computer science. New York: Springer, 2008, oCLC: ocn191726482.

[58] "Operator grammar and precedence parser in TOC," May 2018, section: Compiler Design. [Online]. Available: https://www.geeksforgeeks.org/operator-grammar-and-precedence-parser-in-toc/

[59] "Operator precedence parser - ANTLR 3 - Confluence." [Online]. Available: https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687077/Operator+precedence+parser

[60] A. K. Bansal, *Introduction to Programming Languages*, 1st ed. New York: CRC Press, 2013.

[61] "3.3.5 Practical Considerations for LALR(1) Grammars." [Online]. Available: https://lambda.uta.edu/cse5317/notes/node21.html

[62] D. E. Knuth, "On the translation of languages from left to right," *Information and Control*, vol. 8, no. 6, pp. 607–639, Dec. 1965. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0019995865904262

[63] M. Johnson, "SLR and LR(1) Parsing," 2012. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf

[64] D. Thakur, "LR Parsers - Compiler Design," Jul. 2016. [Online]. Available: https://ecomputernotes.com/compiler-design/lr-parsers

[65] M. Johnson, "LALR Parsing," 2012. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf

[66] G. Tomassetti, "A Guide To Parsing: Algorithms And Terminology," Sep. 2017. [Online]. Available: https://tomassetti.me/guide-parsing-algorithms-terminology/

[67] L. TRatt, "Which Parsing Approach," 2020. [Online]. Available: https://tratt.net/laurie/blog/2020/which_parsing_approach.html

[68] L. Vaillant, "What is Earley parsing, and why you should care?" [Online]. Available: https://loup-vaillant.fr/tutorials/earley-parsing/what-and-why

[69] "Tokenizers - nearley.js - JS Parsing Toolkit." [Online]. Available: https://nearley.js.org/docs/tokenizers

[70] "JSON." [Online]. Available: https://www.json.org/json-en.html

[71] Z. Ajaj, "Fable.SimpleJson," Jan. 2023, original-date: 2017-12-30T17:23:12Z. [Online]. Available: https://github.com/Zaid-Ajaj/Fable.SimpleJson

[72] baeldung, "Type Safety in Programming Languages | Baeldung on Computer Science," Jul. 2022. [Online]. Available: https://www.baeldung.com/cs/type-safety-programming

[73] "Moo!" May 2023, original-date: 2017-03-10T17:28:28Z. [Online]. Available: https://github.com/no-context/moo

[74] cartermp, "Discriminated Unions - F#," Sep. 2021. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions

[75] "SystemVerilog," May 2023, page Version ID: 1153086258. [Online]. Available: https://en.wikipedia.org/w/index.php?title=SystemVerilog&oldid=1153086258

[76] "Dangling-else Ambiguity," Dec. 2021, section: Compiler Design. [Online]. Available: https://www.geeksforgeeks.org/dangling-else-ambiguity/

[77] P. Yadav, "Detect Cycle in Directed Graph," Aug. 2022. [Online]. Available: https://www.scaler.com/topics/detect-cycle-in-directed-graph/

[78] "Detect Cycle in a Directed Graph," Apr. 2012, section: DSA. [Online]. Available: https://www.geeksforgeeks.org/detect-cycle-in-a-graph/

[79] Russell, "Case Statement," Jun. 2022. [Online]. Available: https://nandland.com/case-statement-2/

[80] user103993, "8:1 mux from minimum 2:1 and 4:1 mux," tex.eprint: https://electronics.stackexchange.com/q/251429 tex.howpublished: Electrical Engineering Stack Exchange. [Online]. Available: https://electronics.stackexchange.com/q/251429

[81] u. (https://electronics.stackexchange.com/users/110971/user110971), "VHDL: Are if-else and case statements supposed to synthesize the same hardware?" tex.eprint: https://electronics.stackexchange.com/q/335899 tex.howpublished: Electrical Engineering Stack Exchange. [Online]. Available: https://electronics.stackexchange.com/q/335899

[82] "Find value of k-th bit in binary representation," May 2018, section: Bit Magic. [Online]. Available: https://www.geeksforgeeks.org/find-value-k-th-bit-binary-representation/

[83] "Modify a bit at a given position," Nov. 2017, section: Bit Magic. [Online]. Available: https://www.geeksforgeeks.org/modify-bit-given-position/

[84] "Verilog Example Codes." [Online]. Available: https://verificationguide.com/verilog-examples/

[85] "Verilog Examples." [Online]. Available: https://www.asic-world.com/examples/verilog/index.html

[86] "Behavioral Verilog Module Instantiation Example • Vivado Design Suite User Guide: Synthesis (UG901) • Reader • AMD Adaptive Computing Documentation Portal." [Online]. Available: https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis/Behavioral-Verilog-Module-Instantiation-Example

[87] m8pple, "Instruction Architectures and Compilers Lab : Autumn," Sep. 2022, original-date: 2020-10-22T11:55:08Z. [Online]. Available: https://github.com/m8pple/elec50010-2020-verilog-lab

[88] B. Baas, "VERILOG 4: COMMON MISTAKES," University of California, Davis. [Online]. Available: https://www.ece.ucdavis.edu/~bbaas/281/notes/Handout14.verilog4.pdf

[89] "Introducing ChatGPT." [Online]. Available: https://openai.com/blog/chatgpt

[90] "What Is a Good Typing Speed?" [Online]. Available: https://www.typingpal.com/en/blog/good-typing-speed

[91] "Experiment: How Fast Your Brain Reacts To Stimuli." [Online]. Available: https://backyardbrains.com/experiments/reactiontime

[92] "1 Coding guidelines for ISSIE · tomcl/issie Wiki." [Online]. Available: https://github.com/tomcl/issie/wiki/1---Coding-guidelines-for-ISSIE

[93] "Writing a parser - nearley.js - JS Parsing Toolkit." [Online]. Available: https://nearley.js.org/docs/grammar#postprocessors