Bachelorarbeit Technische Informatik

Analyse der Yosys Prozesse, die Verilog in die interne Datenstruktur RTLIL konvertieren

vorgelegt von

Christopher Parnow

Erstgutachter: Prof. Dr. Tobias Krawutschke (Technische Hochschule Köln)

Zweitgutachter: Dr.-Ing. Michael Karagounis (Fachhochschule Dortmund)

August 2022

Bachelorarbeit

Titel: Analyse der Yosys Prozesse, die Verilog in die interne Datenstruktur RTLIL konvertieren

Gutachter:

Prof. Dr. Tobias Krawutschke (TH Köln)

Dr.-Ing. Michael Karagounis (FH Dortmund)

Zusammenfassung:

Das Ziel der vorliegenden Arbeit besteht darin, die Frage zu beantworten wie Yosys Verilog einliest und daraus RTLIL generiert. Mit der Beantwortung dieser Frage, soll die Datenstruktur RTLIL und die Verknüpfung zu einem Verilog Design besser verstanden werden. Dafür wurde das Frontend von Yosys untersucht und die Datenstruktur RTLIL näher beleuchtet. Als Ergebnis konnte festgehalten werden, dass die AstNode Datenstruktur eine wesentliche Rolle bei der Konvertierung von Verilog zu RTLIL spielt, und mit deren Hilfe beim Einlesen ein abstrakter Syntaxbaum gebildet wird. Allein der Typ des Knotens beeinflusst, wie der RTLIL Generator damit umgeht. Weiter ist die Generierung von RTLIL::Cell Objekten als erster Schritt zur Synthese zu verstehen, da sie durch Technologie Mapping reale Komponenten abbilden können.

Stichwörter: Yosys, Synthese, Frontend, RTLIL, AST

Datum: 15.08.2022

Bachelors Thesis

Title: Analysis of the Yosys processes that convert Verilog into the internal data structure of RTLIL

Reviewers:

Prof. Dr. Tobias Krawutschke (TH Köln)

Dr.-Ing. Michael Karagounis (FH Dortmund)

Abstract: The aim of this thesis is to answer the question of how Yosys reads Verilog and generates RTLIL. By answering this question, the data structure RTLIL and the link to a hardware design written in Verilog should be better understood. For this purpose, the front end of Yosys and the data structure RTLIL was examined more closely. The result of this analysis is that the AstNode structure plays an essential role in the conversion from Verilog to RTLIL and is used to form an abstract syntax tree during read in. Only the type of the node influences how the RTLIL generator handles it. Furthermore, the generation of RTLIL::Cell objects are to be understood as a first step towards synthesis, as they can introduce real components by means of technology mapping.

Keywords: Yosys, synthesis, frontend, RTLIL, AST

Date: 15.08.2022

Danksagung

An dieser Stelle möchte ich mich bei Herr Prof. Dr. Tobias Krawutschke sowie Herr Prof. Dr.-Ing. Michael Karagounis bedanken, die mich während der Bachelorarbeit unterstützt und betreut haben.

Danken möchte ich auch mein Kommilitone Patryk Janik für die konstruktive Kritik und das Lesen meiner Arbeit.

Abkürzungsverzeichnis

ASIC	*Application Specific Integrated Circuit*
AST	*Abstract Syntax Tree*
DEA	*Deterministischer endlicher Automat*
FPGA	*Field Programmable Gate Array*
HDL	*Hardware Description Language*
RTLIL	*Register Transfer Level Intermediate Language*
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*

Verwendete Begriffe

Syntaxbaum: Ein Syntaxbaum bildet die Syntax in Form eines Baumes ab. Ein Baum ist eine Datenstruktur, der aus einer Menge von Knoten und Verbindungen von Knoten besteht. Durch die Verbindungen wird eine hierarchische Struktur aufgebaut, die den Baum darstellen.

Children: Knoten in einem Baum können Unterknoten besitzen. Diese Knoten werden, aus Sicht des Knotens, der Unterknoten besitzt, als Children bezeichnet.

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel soll einen Überblick auf die Motivation, Zielsetzung, Rahmen sowie den Aufbau dieser Arbeit geben.

1.1 Motivation und Zielsetzung der Arbeit

Yosys ist ein Syntheseprogramm für Verilog. Die bekanntesten Hardwarebeschreibungssprachen sind VHDL und Verilog. Sie werden angewendet, um das Verhalten von digitalen Schaltungen in textuellen Modellen zu beschreiben. Mit der Synthese wird aus der Verhaltensbeschreibung einer digitalen Schaltung, auch Hardwaredesign genannt, eine Netzliste erstellt. Diese Netzliste enthält einzelne Komponenten wie z.B. Logikgatter, Flip-Flops und Speicher und deren Verbindungen, die in Verbindung mit Technologie-Mapping eine reelle Schaltung darstellen. Diese kann dann auf physikalischer Ebene in einem ASIC oder einem konfigurierbaren Logikbaustein implementiert werden.

An der Fachhochschule Dortmund wird ein Projekt durchgeführt, bei dem die interne Struktur und Funktion von Yosys untersucht wird und die Beantwortung der Forschungsfrage „Wie funktioniert Synthese“ zum Ziel hat. Für die Synthese nutzt Yosys eine eigene Darstellung, die sogenannte Register Transfer Level Intermediate Language (RTLIL). Diese kann durch eine Textdarstellung in der Konsole lesbar für den Nutzer dargestellt werden. Um zu begreifen, wie Yosys Synthese umsetzt, ist es erforderlich sich mit RTLIL und den Zusammenhang zwischen Verilog und RTLIL auseinanderzusetzen.

Yosys ist quelloffen und stellt auch ein Handbuch zu Verfügung. Bei näherer Betrachtung stellt sich aber heraus, dass der Quellcode zum großen Teil unkommentiert ist und das Handbuch nicht detailliert auf einzelne Prozesse eingeht.

In dieser Bachelorarbeit wird anhand von Untersuchungen der Prozesse im Frontend dargelegt, welche Schritte Yosys durchführt, um ein Hardwaredesign in Verilog in die Datenstruktur RTLIL zu konvertieren. Im weiteren Verlauf der Analyse wird RTLIL näher untersucht und der Zusammenhang zwischen den RTLIL Objekten und den dazugehörigen Konstrukten in Verilog erläutert.

Abschließend werden die Ergebnisse zusammengefasst und herausgestellt, welche neue Erkenntnisse die Analyse geschaffen hat.

1.2 Thematische Abgrenzung der Arbeit

Im Rahmen dieser Arbeit soll herausgearbeitet werden, wie das Einlesen eines Verilogdesign in Yosys funktioniert und wie daraus RTLIL generiert wird. Yosys bietet dem Nutzer verschiedene Möglichkeiten, den Einlese-Prozess anzupassen. Ein konkretes Beispiel ist die Unterstützung von SystemVerilog oder das Ignorieren von bestimmten Elementen im Design. Eine komplette Analyse, die jede unterstützte Option berücksichtigt, ist nicht Ziel dieser Arbeit. Darüber hinaus kann aufgrund der Menge an Elementen in Verilog nicht auf alle Typen explizit eingegangen werden. Der Schwerpunkt dieser Arbeit liegt darin, die prinzipielle Funktionsweise zu beschreiben. Die Datenstruktur RTLIL enthält neben den Typen, die für die Abbildung des Designs verwendet werden, auch Hilfsstrukturen, die vor allem von Passes in Yosys genutzt werden. Für diese Arbeit sind jedoch nur Typen relevant, die beim Einlesen generiert werden und das Design beschreiben.

1.3 Aufbau der Arbeit

Kapitel 1 „Einleitung“ verschafft dem Leser einen Überblick und geht auf Motivation, Zielsetzung und die thematische Abgrenzung ein.

Kapitel 2 „Grundlagen Lexer und Parser“ erklärt, wie ein Lexer und ein Parser funktionieren.

Kapitel 3 “Yosys Analyse“ erfolgt die Untersuchung des Frontends von Yosys. Dazu gehören das Verilog und das AST Frontend.

Kapitel 4 „Die Datenstruktur RTLIL“ wird auf die RTLIL Objekte eingegangen, die das Design abbilden und im Frontend generiert werden.

Kapitel 5 „Zusammenfassung“ schließt die Arbeit ab.

2 Grundlagen Lexer und Parser

In diesem Kapitel wird zunächst auf die Relevanz von Lexer und Parser im Yosys Frontend eingegangen. Anschließend wird die Vorgehensweise von Lexer und Parser beschrieben. Dabei stützt sich die Arbeit auf Grundlagen des Compilerbau und den Anwendungen Flex und Bison.

2.1 Einleitung

Damit ein Hardwaredesign in Verilog in die Datenstruktur RTLIL umgewandelt werden kann, wird das Design durch das Verilog Frontend in einen Syntaxbaum konvertiert. Diese Umwandlung von einer höheren Sprache in einen Syntaxbaum ist ein Konzept des Compilerdesigns mit Lexer und Parser [1, p. 25].

Grundlegend ist ein Compiler ein Programm, das als Eingabe Sourcecode in einer Hochsprache erhält und in eine ausführbare Datei umwandelt. Der Vorgang von der Eingabe bis zur Erzeugung der ausführbaren Datei kann in die Analyse- und Synthesephase eingeteilt werden. In der Analysephase wird der Sourcecode auf lexikalische, syntaktische und semantische Korrektheit überprüft und der Syntaxbaum erzeugt. Der Syntaxbaum ist eine hierarchische Darstellung der Elemente im Sourcecode. Die Synthesephase führt Optimierungen durch und erzeugt aus dem Syntaxbaum die ausführbare Datei [2, p. 4]. Der Lexer und der Parser kommen in der Analysephase zum Einsatz. Das Frontend von Yosys verwendet Lexer und Parser. Deshalb wird in dieser Arbeit auf die Funktionen von Lexer und Parser eingegangen, um für die spätere Analyse des Verilog Frontend das nötige Verständnis aufzubauen.

Abbildung 1 zeigt vereinfacht die zwei Phasen eines Compilers mit dem Syntaxbaum zwischen der Analyse- und Synthesephase.

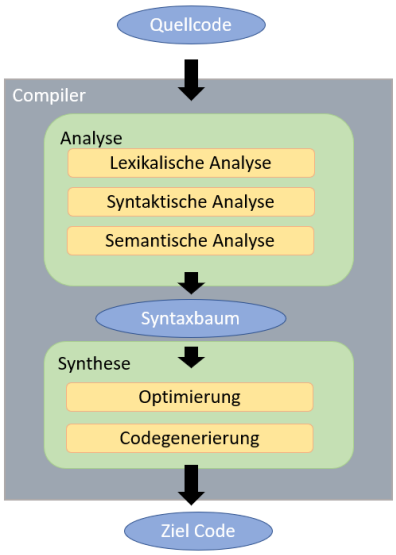

Abbildung 1: Vereinfachte Darstellung der Phasen eines Compilers

2.2 Der Lexer

Der Lexer ist die erste Komponente des Compilers und besitzt als Hauptaufgabe die lexikalische Analyse des Quellcodes [2, p. 5]. In der lexikalischen Analyse wird der Sourcecode Zeichen für Zeichen eingelesen und sogenannte Tokens erzeugt.

2.2.1 Token

Beim Einlesen der Zeichen wird überprüft, ob ein einzelnes Zeichen oder eine Sequenz von Zeichen einen Ausdruck in der Quellsprache bilden. Ist dies der Fall, wird ein Token erstellt. Ein Token besteht in der einfachsten Form aus einem Tupel aus Token-Typ und Wert. Der Token-Typ ist auch gleichzeitig die Bezeichnung des Tokens. Der Wert tritt als optionales Attribut im Token auf und ist immer dann vorhanden, wenn der Token nicht nur mit der Bezeichnung komplett beschrieben werden kann. In Abbildung 2 wird der Wert 10 einer Integervariable zugewiesen. Der Integerwert 10 wird in ein Token überführt mit der Bezeichnung „NUMBER“ und dem Wert „10“.

Abbildung 2: Beispielhafte Darstellung eines Tokens für den Ganzzahlwert 10

Damit der Lexer entscheiden kann, ob Zeichen in einer Zeichenfolge zusammengehören und einen gültigen Ausdruck bilden, sucht der Lexer nach Mustern, die mit Hilfe von regulären Ausdrücken beschrieben werden [3, p. 2]. Wenn Zeichen ein Muster bilden, welches mit einem regulären Ausdruck übereinstimmt, wird ein Token erzeugt.

2.2.2 Der Lexergenerator Flex

Flex ist ein Open Source Lexergenerator, der aus einem Flex-Programm den Lexer erzeugt. Ein Flex-Programm enthält die Beschreibung des Lexers und besitzt die Dateiendung „l“. Die Beschreibung besteht aus den drei Sektionen Definitionen, Regeln und Routinen, die mit “%%“ getrennt werden [3, p. 119].

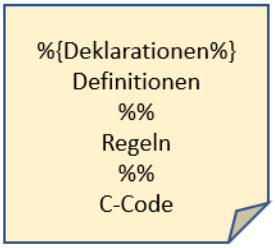

Abbildung 3: Aufbau eines Flex-Programms

Definition

Die Sektion Definition kann verschiedene Komponenten enthalten. Zum einen werden C-Code Deklarationen, die zwischen „%{“ und „%}“ angegeben werden, vom generierten Lexer übernommen. Zum anderen werden unter dieser Sektion Optionen, Startbedingungen und Namen-Definitionen angegeben [3, p. 119]. Namen-Definitionen können als Makros für reguläre Ausdrücke angesehen werden [3, p. 122].

Flex definiert eine Funktion yywrap(), die den Inputstream auf eine andere Datei setzt, wenn das Ende der eingelesen Datei erreicht wird. Mit der Nutzung der Option „%option noyywrap“ wird Flex gesagt, dass der Lexer keine weitere Datei lesen soll [3, p. 24].

Regeln

Diese Sektion entspricht dem Hauptteilsektion des Flex-Programms. Hier werden die Regeln für reguläre Ausdrücke aufgelistet. Dabei geht der generierte Lexer die Regeln von oben nach unten durch, wenn keine Startbedingung genannt wird. Startbedingungen können mit „%start <symbol>“ definiert werden. Eine Regel beginnt immer mit einem regulären Ausdruck, gefolgt von C-Code in geschweiften Klammern. Beim Lexen wird die Eingabe mit dem regulären Ausdruck der Regel verglichen und bei Übereinstimmung der assoziierte C-Code ausgeführt [3, p. 120]. Im einfachsten Fall ist die Aktion die Rückgabe eines Tokens. Für die Speicherung von Werten wird die Variable „yytext“ genutzt, die den eingelesenen Ausdruck enthält und den Wert des aktuellen Tokens darstellt [3, p. 137]. Damit der Wert nicht verloren geht, wird er „yylval“ übergeben, die eine Variable des Parsers ist [3, p. 161].

Routinen

In der Sektion Routinen werden benutzerdefinierte Funktionen in C aufgelistet. Diese Funktionen werden vom generierten Lexer übernommen und genutzt. Im Normalfall steht hier die Routine, die den Lexer mit yylex() aufruft und das Lexen startet.

2.2.3 Reguläre Ausdrücke

Eine Zeichenkette entspricht einem regulären Ausdruck, der nur exakt diesen Muster entspricht. Ausdrücke können aber auch aus Mustern aufgebaut sein, die bestimmten Regeln folgen. In so einem Fall werden in Zeichenklassen Regeln definiert, die angeben welche Zeichen wie oft gültig in einem Ausdruck vorkommen dürfen. Im Folgenden ist eine Tabelle mit Zeichenklassen aufgelistet. Ein Beispiel für einen regulären Ausdruck stellt [4-9] dar, der sagt, dass alle Ganzzahlen zwischen 4 und 9 gültig sind. Wird der Ausdruck in [4-9]+ geändert, deutet das „+“, dass der Ausdruck mindestens einmal vorkommen muss und beliebig oft vorkommen kann. Gültige Zahlen wären z.B. 5, 44 oder 85.

a	Ein Zeichen ist gültig
abc	Zeichenkette ist gültig
(a|b)	Linker und rechter Ausdruck ist gültig
[abc]	Jedes Zeichen in [] ist gültig
[^abc]	Jedes Zeichen außer in [] ist gültig
a+	Ausdruck enthält min. ein bestimmtes Zeichen
a*	Zeichen kommt 0-n im Ausdruck vor
a{min,max}	Zeichen kommt min-max im Ausdruck vor
a?	Zeichen kommt 0-1 im Ausdruck vor
.	Alle Zeichen außer Zeilenumbruch sind gültig
\d	Zeichen ist Zahl
\D	Zeichen ist keine Zahl
\s	Zeichen ist Leerzeichen
\S	Zeichen ist kein Leerzeichen
\n	Zeichen ist Zeilenumbruch
\t	Zeichen ist Tabulator

Tabelle 1: Zeichenklassen für reguläre Ausdrücke [3, p. 19]

Aus den regulären Ausdrücken wird mit dem generierten Lexer von Flex ein deterministischer endlicher Automat (DEA) umgesetzt. Die Definition lautet für einen Automat A mit A= (Q, ∑, δ, $q0$, F) folgendermaßen [4, p. 53]:

- Q ist eine endliche Menge an Zuständen
- ∑ ist die Menge der gültigen Eingabesymbole (Eingabealphabet)
- δ: Q x ∑ -> Q sind die Übergangsfunktionen
- $q0$ ϵ von Q ist das Startsymbol oder Anfangszustand
- F ist die Menge an Endzuständen

Verdeutlicht wird das an einem Beispiel für den Ausdruck 00(0|1)*. Dieser Ausdruck sagt aus, dass die ersten zwei Zeichen 0 sein müssen und mit (0|1)* werden weitere Zeichen für den Ausdruck akzeptiert, solange es sich um 0 oder 1 handelt. Formell sieht das wie folgt aus:

- Q = $\{q0, q1, q2\}$
- ∑ = {0, 1}
- δ: ($q0$, 0, $q1$), ($q1$, 0, $q2$), ($q2$, 0, $q2$), ($q2$, 1, $q2$)
- $q0$ = Startzustand
- F = $\{q2\}$

In Abbildung 4 ist der dazugehörige DEA graphisch dargestellt. Am Anfang befindet man sich im Zustand $q0$. Die Übergangsfunktion ($q0$, 0, $q1$) sagt aus, dass mit 0 in den Zustand $q1$ gewechselt wird. Mit einer weiteren 0 wird von $q1$ in den Zustand $q2$ übergegangen. Der Zustand $q2$ ist ein Endzustand und akzeptiert als kleinsten Ausdruck 00. Mit den Übergangsfunktionen ($q2$, 0, $q2$) und ($q2$, 1, $q2$) erlaubt der DEA, dass der Ausdruck weitere Zeichen von 0 und 1 besitzen kann und im Endzustand bleibt.

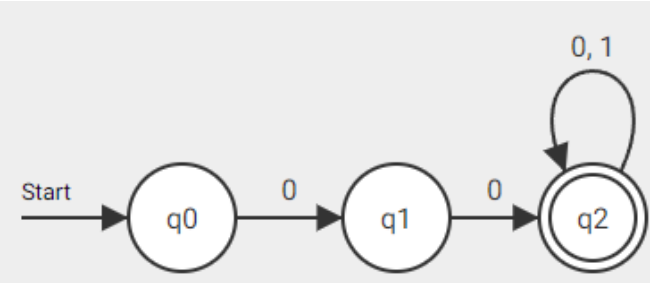

Abbildung 4: Beispiel für einen DEA der den Ausdruck 00(0|1)* beschreibt

2.3 Der Parser

Der Parser ist eine weitere Komponente eines Compilers. Er übernimmt die syntaktische Analyse und bekommt als Eingabe Token vom Lexer übergeben. Bei der syntaktischen Analyse wird die Beziehung der Token unter Anwendung einer definierten Grammatik überprüft. Mit einem Syntaxbaum wird die Beziehung unter den Token dargestellt [3, p. 9]. Folgt eine Kette von Token nicht der definierten Grammatik des Sourcecodes, wird vom Parser ein Syntaxerror ausgegeben.

Abbildung 5 zeigt anhand eines Beispiels die Zerlegung in Token und den dazugehörigen Syntaxbaum. Dabei ist zu erkennen, dass die Struktur des Baumes mit der Zuweisung im Beispiel übereinstimmt.

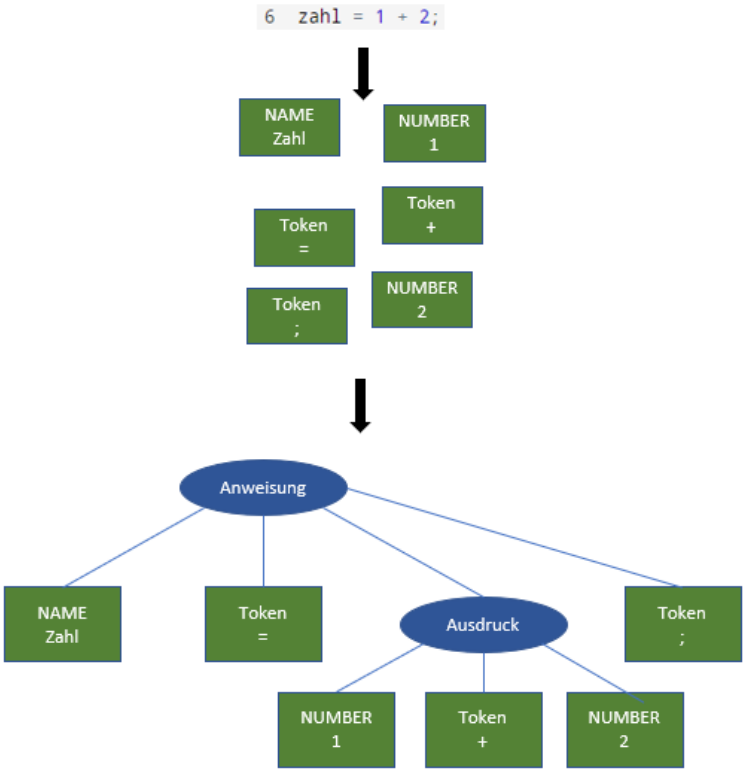

Abbildung 5: Zerlegung eines Quellcodes in Token und Aufbau des Syntaxbaum

2.3.1 Grammatik

Die Syntax einer Sprache in der Programmierung gibt vor, wie eine Anweisung gebildet werden kann. Eine einfache Zuweisung besteht im einfachsten Fall aus den vier Elementen <name>, "=", <ausdruck> und ";". Die Regeln, die entscheiden, ob eine Syntax korrekt ist, bilden die Grammatik. Die Regeln im Parser sind so aufgebaut, dass auf der linken Seite der Regel nur eine Variable steht. Dieses Merkmal sagt aus, dass die Grammatik im Parser eine kontextfreie Grammatik ist. Eine kontextfreie Grammatik G wird definiert mit G = (N, T, P, S) [4, p. 32] wobei:

- N eine endliche Menge von Nicht-Terminale ist
- T eine endliche Menge von Terminalen ist
- P eine Menge an Produktionsregeln ist
- S ein Startsymbol ist und zu den Nichtterminalsymbolen gehört

Die einzelnen Elemente, die ausgewertet werden, bezeichnet man als Symbole. Dabei teilen sich die Symbole in Terminal- und Nicht-Terminalsymbole auf. Terminalsymbole sind Symbole, die nicht weiter durch Produktionsregeln ausgetauscht werden können. Nicht-Terminalsymbole können durch Anwendungen von Produktionsregeln in andere Symbole ausgedrückt werden.

Abbildung 6 zeigt einen Ausdruck aus Terminal- und Nicht-Terminalsymbolen. Mit Anwendung der Produktionsregeln, kann der Ausdruck mit dem Nicht-Terminalsymbol „Modul" ersetzt werden. Dieses Symbol kann weiter mit „Design" ersetzt werden.

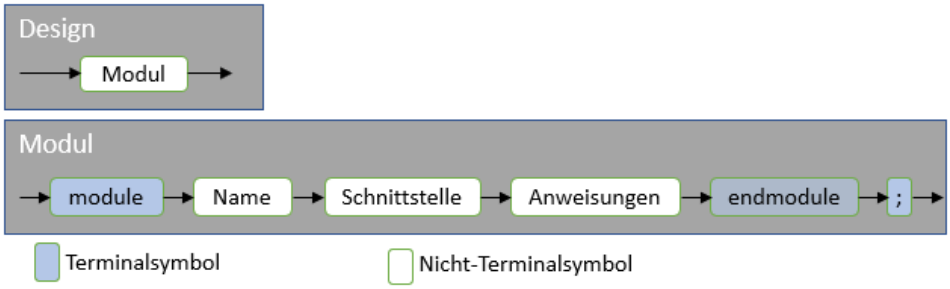

Abbildung 6: Beispielsyntax für ein Verilogmodul mit Regeln für Design und Modul

Backus-Naur Form

Die Produktionsregeln in der Grammatik sind in der Backus-Naur Form (BNF) definiert. Eine Produktionsregel besteht aus einer linken und rechten Seite. Auf der linken Seite ist immer ein Nicht-Terminalsymbol und auf der rechten Seite können Terminal- und Nicht-Terminalsymbolen stehen. Token stellen Terminalsymbole dar [3, p. 10].

```
<Nicht-Terminalsymbol> : <Nicht-Terminalsymbol> + < Terminalsymbol>
```

Betrachtet man das Beispiel in Abbildung 6, ergeben sich folgende Regeln in Backus-Naur-Form:

```
Design : Modul
Modul : „module" + Name + Schnittstelle + Anweisungen + „endmodule"
+ „;"
```

Bildet man die Regeln auf einen Syntaxbaum ab, ergibt sich eine hierarchische Struktur für die verwendeten Symbole und Token. Abbildung 7 zeigt, dass das Symbol „Design" in diesem Beispiel an höchster Stelle steht. An zweithöchster Stelle kommt das Symbol „Modul" und darunter liegen alle anderen Symbole.

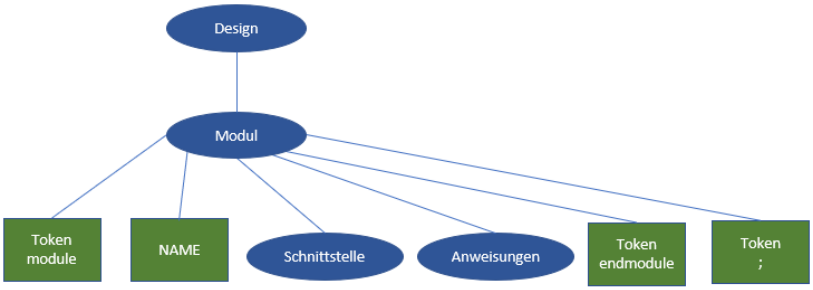

Abbildung 7: Beispiel für einen Syntaxbaum aus den Regeln für Design und Modul

2.3.2 Shiften und Reduzieren

Um zu entscheiden, welche Regeln für Token angewendet werden sollen, nutzt der Parser das Shiften und Reduzieren. Dazu wird ein Stack verwendet, auf dem Symbole gelegt werden. Nach dem Einlesen eines einzelnen Tokens wird dieser auf dem Stack gelegt. Dieser Vorgang wird als Shift bezeichnet. Anschließend wird überprüft, ob für die Symbole auf dem Stack eine Produktionsregel angewendet werden kann. Das Shiften erfolgt so lange, bis eine Produktionsregel anwendbar ist. Ist dies der Fall, erfolgt das Reduzieren. In diesem Schritt werden die Symbole auf dem Stack, die den rechten Ausdruck einer Produktionsregel entsprechen aus dem Stack entfernt und das Symbol auf der linken Seite der Regel auf dem Stack gelegt [3, p. 48]. Das Shiften und Reduzieren wird so lange ausgeführt, bis am Ende nur das Startsymbol im Stack liegt. Das zeigt, dass der eingelesene Code syntaktisch korrekt war. Andernfalls liegt ein Syntaxfehler vor.

Abbildung 8 zeigt vereinfacht das Reduzieren anhand einer Produktionsregel und dem Zustand des Stacks vor und nach dem Reduzieren auf. Zuerst liegen auf dem Stack Symbole, auf die eine Produktionsregel angewendet werden kann. Sie werden vom Stack entfernt und das Symbol „Addition" wird auf den Stack gelegt.

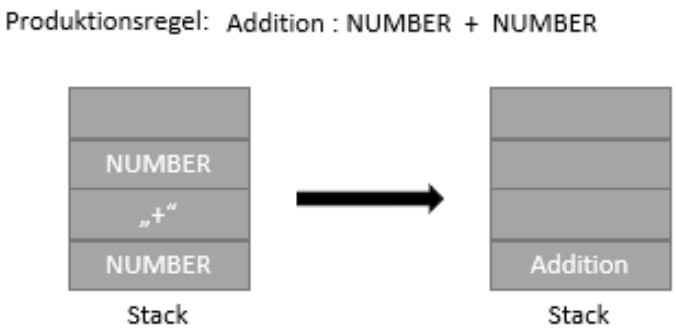

Abbildung 8: Reduzieren anhand einer Produktionsregel

2.3.3 Der AST

Ein Syntaxbaum, der aus den Produktionsregeln der definierten Grammatik erstellt wurde, enthält Informationen bzw. Symbole, die nur für Syntaxüberprüfung gebraucht werden. Sieht man sich z.B. vereinfacht eine If-Verzweigung in Abbildung 9 an, erkennt man, dass für eine gültige Syntax für das Symbol „if-Verzweigung“, die darunterliegenden Symbole bzw. Token in dieser Reihenfolge vorliegen müssen. Nachdem der Ausdruck ausgewertet wurde, werden die Token mit den Klammern und dem „if“ nicht mehr benötigt.

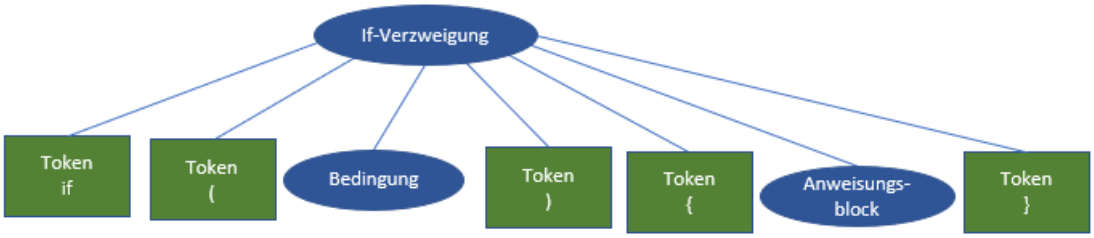

Abbildung 9: Syntaxbaum für eine if-Verzweigung

Entfernt man diese Elemente von dem Syntaxbaum enthält man den abstrakten Syntaxbaum (engl. Abstract Syntax Tree, AST). Ein AST ist also ein Syntaxbaum, der keine redundante Informationen der Syntaxanalyse mehr enthält. Wird das Beispiel von Abbildung 9 vereinfacht, entsteht der AST in Abbildung 10.

Abbildung 10: AST für eine if-Verzweigung

2.3.4 Der Parsergenerator Bison

Bison ist ein Open Source Parsergenerator, der aus einem Bison-Programm den Parser erzeugt. Ein Bison-Programm enthält die Beschreibung des Parsers und besitzt die Dateiendung „y“. Die Beschreibung besteht ähnlich wie bei einem Flex-Programm aus den Bereichen Definitionen, Regeln und Routinen [3, p. 141].

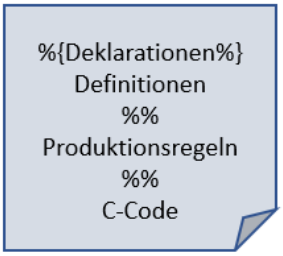

Abbildung 11: Aufbau eines Bison-Programms

Definitionen

Genau wie im Flex-Programm gibt es einen Block der C-Code Deklaration zwischen „%{„ und „%}“ beinhaltet. In dieser Sektion stehen vor allem Definitionen für Token, Symbole und mögliche Wertetypen, wie z.B. Integer, der ein Token oder Symbol enthalten kann. Token werden mit „%token <typ> <tokenname>“ definiert, können aber auch mit „%left“, „%right“ oder „%nonassoc“ angegeben werden. Jedes Symbol kann einen Wert enthalten. Alle möglichen Typen, werden in „%union {<Deklarationen>}“ deklariert. Die Assoziation von Typ und Symbol wird mit „%type <typ> <symbol>“ angegeben [3, pp. 162-163].

Regeln

Die Produktionsregeln der Grammatik sind in der Sektion Regeln zu finden. Dabei enthält eine Regel Aktionen, die in Form von C-Code in geschweiften Klammern angegeben sind. Zwei Produktionsregeln für das gleiche Nicht-Terminalsymbol auf der linken Seite werden mit „|“, als oder, zusammengefasst. Für den Zugriff auf den Wert für das Nicht-Terminalsymbol auf der linken Seite wird „$$“ genutzt. Werte für Symbole auf der rechten Seite werden in Reihenfolge mit „$1“, „$2“, usw. definiert[1] [3, pp. 142-143].

Routinen

Benutzerdefinierte Funktionen vom Nutzer sind in der Sektion Routinen aufgelistet und werden vom Parser übernommen. Im Normalfall steht hier das Hauptprogramm, das den Parser mit yyparse() aufruft, und eine Routine für Fehlerausgaben [3, pp. 142-143].

3 Yosys Analyse

Yosys ist ein Synthesewerkzeug für Designmodelle in Verilog. Verilog gehört mit VHDL zu den meistgenutzten Hardwarebeschreibungssprachen und wurde ursprünglich für die Modellierung, Simulation und Analyse digitaler Schaltungen entwickelt. Der Vorgang, der die Beschreibung einer Schaltung in eine Netzliste umwandelt, wird Synthese genannt. In Abbildung 12 ist die Synthese vereinfacht dargestellt.

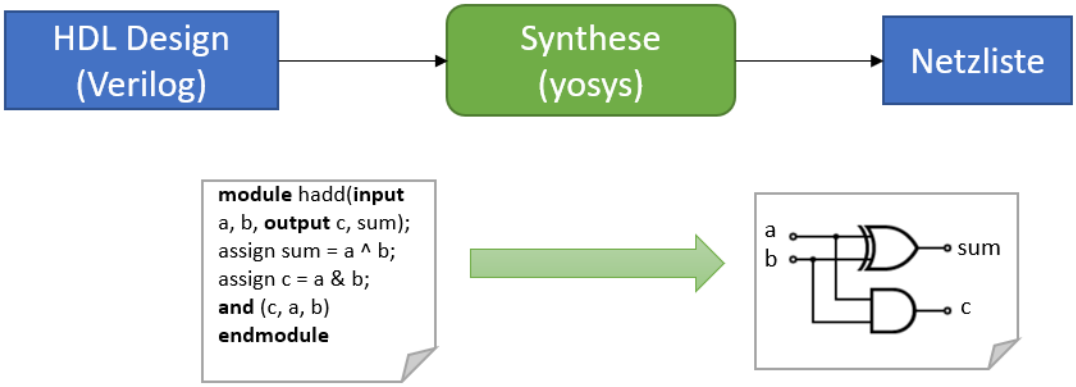

Abbildung 12: Vereinfachte Darstellung der Synthese

Der Kontrollfluss von Yosys kann, wie in Abbildung 13 dargestellt ist, in drei Abschnitte unterteilt werden. Der erste Abschnitt ist das Frontend. Die Aufgabe des Frontends ist das Einlesen von Eingaben, die der Nutzer tätigt. Dazu gehört die Aufbereitung von Eingabedateien in die Datenstruktur RTLIL. Auf RTLIL wird im Kapitel 4 näher eingegangen. Nach der Ausführung des Frontends wird die eigentliche Synthese durchführt. Die Prozesse, welche die Synthese durchführen, werden in Yosys als „Passes“ bezeichnet und werden auf RTLIL angewendet. Der letzte Abschnitt ist das Backend. Hier wird RTLIL in das gewünschte Ausgabeformat umgewandelt.

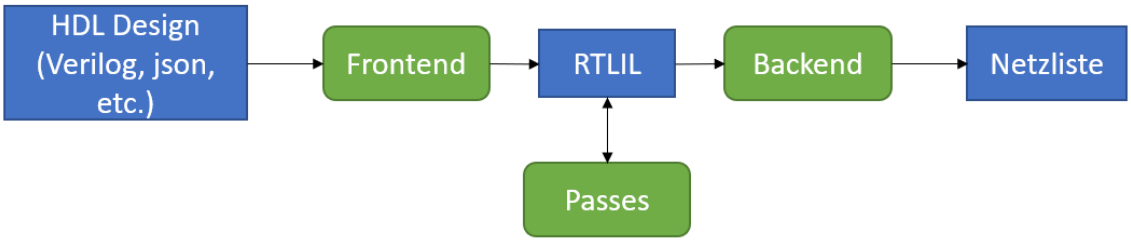

Abbildung 13: Kontroll- und Datenfluss vereinfacht

Diese Arbeit beschäftigt sich mit dem Frontend von Yosys. Der Fokus liegt auf den Prozessen, die dafür zuständig sind, Verilog in RTLIL zu konvertieren. Dabei werden die relevanten Teile der Prozesse untersucht, die für das Verständnis der Umwandlung erforderlich sind.

3.1 Das Verilog Frontend

Wenn eine Verilogdatei in Yosys eingelesen wird, führt Yosys das Verilog Frontend aus. Dies geschieht mit dem Befehl „read_verilog <filename>“ in der Kommandozeile von Yosys. Das Frontend wird durch die Funktion VerilogFrontend::execute() in der Datei „verilog_frontend.cc“ ausgeführt. Yosys unterstützt neben Verilog begrenzt auch SystemVerilog bei der Verwendung des Arguments „-sv“ [1, p. 156] und bietet zusätzliche Optionen an. Alle möglichen Optionen für den „read_verilog“ Befehl sind im Yosys Handbuch aufgelistet [1, p. 156]. Grundlegend besteht das Verilog Frontend aus den Komponenten Präprozessor, Lexer und Parser [1, p. 64]. Deshalb liegt der Fokus in diesem Kapitel auf den genannten Komponenten. Zunächst wird die Funktion VerilogFrontend::execute() untersucht. Yosys führt sie, wie in Abbildung 14 dargestellt, mit folgenden Parametern aus:

- Der Inputstream std::istream *&f enthält den Sourcecode
- Das String Objekt std::string filename enthält den Dateinamen
- Zusätzliche Argumente sind in std::vector<std::string> args enthalten
- Der Zeiger RTLIL::Design *design zeigt auf ein RTLIL::Design Objekt, das am Anfang leer ist und mit der Konvertierung mit RTLIL Objekten befüllt wird.

```
236    void execute(std::istream *&f, std::string filename, std::vector<std::string> args, RTLIL::Design *design) override
```

Abbildung 14: Kopf der Funktion VerilogFrontend::execute()

Der Programmablauf der Funktion ist vereinfacht und in mehrere Schritte eingeteilt in Abbildung 15 dargestellt:

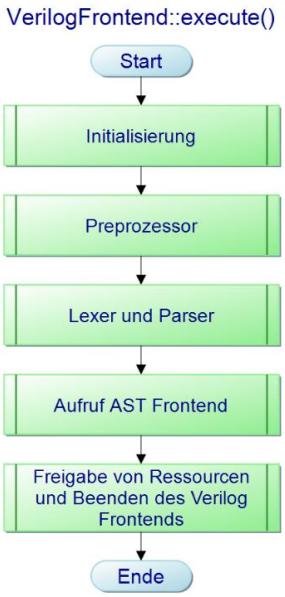

Abbildung 15: Abschnitte im Verilog Frontend

3.1.1 Initialisierung

Der erste Schritt im Verilog Frontend ist die Initialisierung von Objekten und Variablen, die während der Ausführung genutzt werden, und die Auswertung zusätzlicher Argumente, die Yosys unterstützt. Dazu gehört auch der Befehl „verilog_defaults“, der laut Yosys dem Nutzer ermöglicht, Default Optionen zu setzen [1, p. 158]. Im Folgenden sind die wichtigsten Variablen erläutert, die für die Konvertierung relevant sind.

- defines_map: Ist ein Key-Value Storage Objekt, das genutzt wird, um Makros zu speichern. Makros werden mit der Kommandozeile definiert oder sind auch im Verilogcode mit „`define“ eingepflegt. Der Präprozessor identifiziert diese und speichert sie ab.
- include_dirs: Ist eine Liste von String Objekten, in der Verzeichnisnamen gespeichert werden. Damit sind die Pfade gemeint, die mit der Compiler Direktive „`include“ im Code einbezogen werden.
- current_ast: Ist ein Zeiger auf ein leeres AST, das während des Parsens mit AST::AstNode Objekten ergänzt wird.
- code_after_preproc: Der durch den Präprozessor aufbereitete Sourcecode wird in diesem String Objekt gespeichert
- lexin: Stellt den Inputstream, also die Eingabe des Lexers dar.

3.1.2 Der Präprozessor

Der Präprozessor ist die erste grundlegende Komponente des Frontends und wird im Normalfall vor dem Lexen und Parsen ausgeführt. Yosys bietet aber dem Nutzer die Option, mit dem Argument „-noop“ den Schritt zu überspringen. Für den Fall, dass der Präprozessor nicht ausgeführt werden soll, wird der Inputstream mit dem Sourcecode „f“ dem Inputstream des Lexers „lexin“ zugewiesen und der If-Block für die Ausführung des Präprozessors übersprungen. Ansonsten wird der Präprozessor mit der Funktion „frontend_verilog_preproc()“ aufgerufen, welcher dann den bearbeiteten Sourcecode einem String Objekt „code_after_preproc“ als Rückgabe zuweist. Dieser wird dann dem Inputstream des Lexers als neues String Objekt zugewiesen. Das Beschriebene deckt sich mit dem in Abbildung 16 dargestellten Codeauszugs aus dem Verilog Frontend.

```
472         lexin = f;
473         std::string code_after_preproc;
474
475         if (!flag_nopp) {
476             code_after_preproc = frontend_verilog_preproc(*f, filename, defines_map, *design->verilog_defines, include_dirs);
477             if (flag_ppdump)
478                 log("-- Verilog code after preprocessor --\n%s-- END OF DUMP --\n", code_after_preproc.c_str());
479             lexin = new std::istringstream(code_after_preproc);
480         }
```

Abbildung 16: Aufruf des Präprozessors im Verilog Frontend

Der Präprozessor wird mit den folgenden Parametern aufgerufen:

- Der Sourcecode der Datei als Inputstream (f)
- Der Directory Name als String Objekt (filename)
- Ein Objekt des Typen defines_map_t für die Speicherung und Bearbeitung von Makros (defines_map)
- Ein Zeiger für ein Objekt vom Typ defines_t im RTLIL::Design Objekt für die Speicherung von Verilog Makros (*design->verilog_defines)
- Eine Liste für die Speicherung und Bearbeitung von Directory Namen in Form von String Objekten, die inkludiert wurden (include_dirs)

Die Aufgabe des Präprozessors besteht darin, Compiler Direktiven im Verilog Code zu identifizieren und auszuführen. Compiler Direktive sind Anweisungen, die für den Compiler bestimmt sind und im Verilogcode mit dem Zeichen „`“ beginnen und die Kompilierung beeinflussen [5].

Abbildung 17 zeigt die Beeinflussung anhand eines einfachen Beispiels mit den Compileranweisungen `define, `ifdef, `else und `endif. Ist ein Makro „behave“ definiert, wird die Zuweisung „assign a = b“ kompiliert. Ansonsten wird die Zuweisung im else-Block kompiliert.

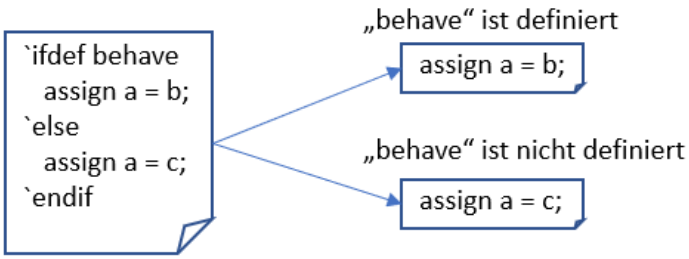

Abbildung 17: Beispiel für das Kompilierverhalten mit Compiler Direktiven

Der Präprozessor in Yosys untersucht jedes Zeichen im Verilogcode und bildet aus zusammenhängenden Zeichen String Objekte, die er als Token bezeichnet. Diese sind nicht zu verwechseln mit den Token des Lexers. Im Präprozessor sind Token Compileranweisungen, Makronamen, Directory Namen oder auch normaler Verilogcode. Dabei wird eine Zeile eingelesen und der Token von der Funktion next_token() zurückgegeben. Der Token wird dann mit Hilfe von If-Else-Bedingungen untersucht, ob und um welche Compileranweisung es sich handelt. Für jede gefundene Direktive werden Aktionen im zugehörigen If-Block ausgeführt. Was unter den Aktionen gemeint ist, wird im anschließenden Abschnitt näher erläutert. Gehört der Token zu keiner Compiler Direktive, wird dieser in einer String Liste „output_code“ hinzugefügt. Dies geschieht so lange in einer Schleife, bis die String Liste „input_buffer“, die den Verilogcode enthält, leer ist und somit jedes Zeichen im Sourcecode untersucht wurde. Am Ende werden alle String Elemente im „output_code“ in ein String Objekt „output“ gespeichert, die Buffer zurückgesetzt und der bearbeitete Verilogcode in „output“ dem Verilog Frontend zurückgegeben.

Abbildung 18 zeigt das Ende des Präprozessors mit der Rückgabe von „output“.

```
977         output_code.push_back(tok);
978     }
979
980     if (ifdef_fail_level > 0 || ifdef_pass_level > 0) {
981         log_error("Unterminated preprocessor conditional!\n");
982     }
983
984     std::string output;
985     for (auto &str : output_code)
986         output += str;
987
988     output_code.clear();
989     input_buffer.clear();
990     input_buffer_charp = 0;
991
992     return output;
993 }
```

Abbildung 18: Erzeugung der Rückgabe des String Objekts „output“ im Präprozessor

Die Aktionen, die der Präprozessor bei Erkennung einer Compiler Direktive durchführt, wird anhand von „`endif“ in Abbildung 19 verdeutlicht. Der Verilogcode ist im input_buffer gespeichert. Solange der Buffer nicht leer ist, extrahiert die Funktion next_token() den Token und speichert diesen im String Objekt „tok“. Der Token wird untersucht, ob und um welche Compiler Direktive es sich handelt. Für den Fall von „`endif“ muss auch überprüft werden, ob im Code davor auch ein „`ifdef“ existiert, damit es gültig ist. Da es sich um eine Verzweigung handelt, muss zusätzlich festgehalten werden, ob das in der Bedingung angegebene Makro definiert ist. Dies wird mit zwei Integervariablen „ifdef_fail_level“ und „ifdef_pass_level“ umgesetzt. Für ein „`ifdef‘ mit definiertem Makro wird „ifdef_pass_level“ inkrementiert, ansonsten „ifdef_fail_level“. Die Compiler Direktive „`endif“ ist gültig, wenn einer der Integervariablen größer 0 ist. Die jeweilige Variable wird dekrementiert. Bei Ungültigkeit, gibt Yosys eine Fehlermeldung aus und bricht den Vorgang ab.

```
778     while (!input_buffer.empty())
779     {
780         std::string tok = next_token();
781         // printf("token: >>%s<<\n", tok != "\n" ? tok.c_str() : "NEWLINE");
782
783         if (tok == "`endif") {
784             if (ifdef_fail_level > 0)
785                 ifdef_fail_level--;
786             else if (ifdef_pass_level > 0)
787                 ifdef_pass_level--;
788             else
789                 log_error("Found %s outside of macro conditional branch!\n", tok.c_str());
790             continue;
791         }
```

Abbildung 19: Codeausschnitt aus dem Preprozessor für die Compileranweisung „endif“

Der Präprozessor erkennt folgende Compiler Direktive:

Compiler Direktive	*Funktion*
include	Einbinden einer externen Verilogdatei
define	Speicher das Define in define_map
undef	Entferne Define in define_map
ifndef	Gibt den Preprozessor an, welcher Teil im Verilog Code kompiliert werden soll und welcher Teil übersprungen werden soll.
ifdef	
else	
elsif	
endif	
timescale	Nicht synthetisierbar, wird entfernt
resetall	Löscht alle Defines in define_map

Tabelle 2: Auflistung unterstützter Compiler Direktiven im Präprozessor

Zusätzlich zu den unterstützen Compileranweisungen, wird der Anfang und das Ende des Sourcecodes mit „`file_push „<filelocation>““ und „`file_pop“ markiert. Dies deckt sich mit der Ausgabe des Verilogcodes nach Ausführung des Präprozessors in Abbildung 20.

```
1. Executing Verilog-2005 frontend: /home/user/Desktop/test/testlp.v
Parsing Verilog input from `/home/user/Desktop/test/testlp.v' to AST representation.
-- Verilog code after preprocessor --
`file_push "/home/user/Desktop/test/testlp.v"
module test(input A, B, output reg C);
assign C = A + B;
endmodule

`file_pop
-- END OF DUMP --
```

Abbildung 20: Dump des Codes nach Präprozessor

3.1.3 Der Verilog Lexer und Parser

Für die weitere Verarbeitung des Sourcecodes wird das Lexen und Parsen angewendet. Die grundlegenden Vorgänge wurden in Kapitel 2 erläutert und werden im Verilog Lexer und Verilog Parser in Yosys angewendet. Der Verilog Lexer und Parser werden jeweils vom Lexer-Generator Flex und Parser-Generator Bison erzeugt. Dieses Unterkapitel beschreibt den Ablauf von Lexen und Parsen im Verilog Frontend. Im Verilog Frontend werden der Verilog Lexer und Parser mit den Funktionsaufrufen gestartet, die in Abbildung 21 dargestellt sind.

```
498         frontend_verilog_yyset_lineno(1);
499         frontend_verilog_yyrestart(NULL);
500         frontend_verilog_yyparse();
501         frontend_verilog_yylex_destroy();
```

Abbildung 21: Funktionsaufruf von Lexer und Parser im Verilog Frontend

- frontend_verilog_yyset_lineno(1): Die Angabe der Zeilennummer wird auf 1 gesetzt.
- frontend_verilog_yyrestart(NULL): Im Normalfall nutzt man yyrestart(file), um die einzulesende Datei zu ändern [3, p. 123]. Weil für den Input „lexin“ genutzt wird, erfolgt als Übergabe NULL. Ein Argument führt dazu, dass der Input Buffer geleert wird.
- frontend_verilog_yyparse(): Diese Funktion ruft den Parser auf und startet das Parsen.
- frontend_verilog_yylex_destroy(): Mit yylex_destroy() werden die Ressourcen des Lexers wieder freigegeben.

Mit dem Aufruf von frontend_verilog_yyparse() wird die Funktion yyparse() in „verilog_parser.tab.cc“ aufgerufen. Der Parser speichert ein eingelesenes Token des Lexers in der Variable „yychar“ ab. Solange dieses Token leer ist, fordert der Parser mit der Funktion „yylex()“ ein Token vom Lexer an. Abbildung 22 zeigt den Aufruf des Lexers im Verilog Parser.

```
3714    /* YYCHAR is either empty, or end-of-input, or a valid lookahead.  */
3715    if (yychar == FRONTEND_VERILOG_YYEMPTY)
3716      {
3717        YYDPRINTF ((stderr, "Reading a token\n"));
3718        yychar = yylex (&yylval, &yylloc);
3719      }
```

Abbildung 22: Aufruf des Lexers im Parser mit yylex

Im Switch-Case-Block der Funktion „yylex()“ ist, wie in Abbildung 23 dargestellt, zu erkennen, wie die Lexer Regeln im Flex-Programm in den generierten Lexer übernommen werden.

```
332 {FIXED_POINT_NUMBER_NO_DEC} {
333         yylval->string = new std::string(yytext);
334         return TOK_REALVAL;
335 }
```

→

```
3366 case 102:
3367 YY_RULE_SETUP
3368 #line 333 "frontends/verilog/verilog_lexer.l"
3369 {
3370         yylval->string = new std::string(yytext);
3371         return TOK_REALVAL;
3372 }
3373         YY_BREAK
```

Abbildung 23. Lexerregel (links) und Case Block im generierten Lexer (rechts)

Nach dem Einlesen ist der Token in „yychar“ und der Wert des Tokens in „yylval“ gespeichert. Das Speichern der Position in der Datei erfolgt, wie in Abbildung 24 dargestellt, mit dem Makro YY_USER_ACTION. Es wird mit Hilfe der Variablen „old_location“ und „real_location“ die Position festgehalten und in die Variable „yylloc“ des Parsers überführt.

```
#define YY_USER_ACTION \
        old_location = real_location; \
        real_location.first_line = real_location.last_line; \
        real_location.first_column = real_location.last_column; \
        for(int i = 0; yytext[i] != '\0'; ++i){ \
                if(yytext[i] == '\n') { \
                        real_location.last_line++; \
                        real_location.last_column = 1; \
                } \
                else { \
                        real_location.last_column++; \
                } \
        } \
    (*yylloc) = real_location;
```

Abbildung 24: Makro YY_USER_ACTION

Der Lexer gibt eine Ganzzahl zurück, die ein Token repräsentiert. Die Zuordnung zwischen Zahlen und Tokens wird in der Headerdatei des Parsers definiert. Abbildung 25 zeigt einen Ausschnitt der Definitionen.

```
66 # define FRONTEND_VERILOG_YYTOKENTYPE
67   enum frontend_verilog_yytokentype
68   {
69     FRONTEND_VERILOG_YYEMPTY = -2,
70     FRONTEND_VERILOG_YYEOF = 0,     /* "end of file"  */
71     FRONTEND_VERILOG_YYerror = 256, /* error  */
72     FRONTEND_VERILOG_YYUNDEF = 257, /* "invalid token"
73     TOK_STRING = 258,               /* TOK_STRING  */
74     TOK_ID = 259,                   /* TOK_ID  */
75     TOK_CONSTVAL = 260,             /* TOK_CONSTVAL  */
76     TOK_REALVAL = 261,              /* TOK_REALVAL  */
77     TOK_PRIMITIVE = 262,            /* TOK_PRIMITIVE  */
```

Abbildung 25: Ausschnitt von definierten Tokens im Header vom Parser

Bei der Formulierung der Produktionsregeln verwendet der Parser Terminal- und Nicht-Terminalsymbole, die in einer Symbolliste „yysymbol_kind_t“ aufgelistet sind. Mit der Funktion „YYTRANSLATE“ werden Token des Lexers in Symbole umgewandelt und in „yytoken“ gespeichert. Der Codeabschnitt in Abbildung 26 zeigt den Aufruf von „YYTRANSLATE“ mit dem übergebenen Token in „yychar“.

```
3740    yytoken = YYTRANSLATE (yychar);
3741    YY_SYMBOL_PRINT ("Next token is", yytoken, &yylval, &yylloc);
```

Abbildung 26: Umwandlung von yychar zu yytoken

Für jedes Reduzieren werden die Aktionen, die in den Produktionsregeln angegeben sind, ausgeführt. Für den Verilog Parser gehört zu den Aktionen, die Erzeugung von AstNode Objekten, wie in Abbildung 27 zu erkennen ist.

```
2359 always_event:
2360         TOK_POSEDGE expr {
2361                 AstNode *node = new AstNode(AST_POSEDGE);
2362                 SET_AST_NODE_LOC(node, @1, @1);
2363                 ast_stack.back()->children.push_back(node);
2364                 node->children.push_back($2);
2365         } |
2366         TOK_NEGEDGE expr {
2367                 AstNode *node = new AstNode(AST_NEGEDGE);
2368                 SET_AST_NODE_LOC(node, @1, @1);
2369                 ast_stack.back()->children.push_back(node);
2370                 node->children.push_back($2);
2371         } |
```

Abbildung 27: Auszug der Produktionsregel für das Nichtterminalsymbol "always_event"

Eine anschauliche Darstellung, wie das Reduzieren angewandt wird und ein AstNode Objekt erstellt wird, ist in Abbildung 28 anhand der Zuweisung „assign C = A + B“ zu sehen. Dabei wurden die Ausgaben von Yosys genutzt. Die Aktionen werden unter dem Punkt „Der Verilog Parser“ näher beleuchtet.

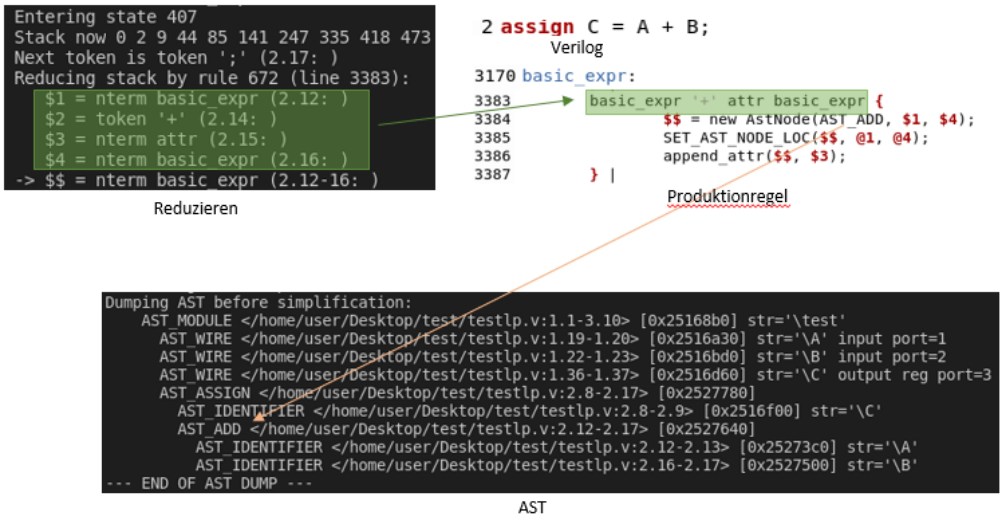

Abbildung 28: Erzeugung eines AstNode Objektes anhand einer Zuweisung in Verilog

Das Verilog Flex-Programm

Der Aufbau eines Flex-Programms wurde in Kapitel 2 beschrieben. Dieses Kapitel betrachtet Abschnitte aus dem Flex-Programm in Yosys, um das Speichern der Werte zu verstehen. Dazu werden Ausschnitte näher betrachtet. Der Lexer unterstützt neben Verilog einen Teil von SystemVerilog. Für Schlüsselwörter in Verilog wird der dazugehörige Token zurückgegeben. Wenn es sich um ein Schlüsselwort von SystemVerilog handelt, erfolgt die Rückgabe des Tokens mit „SV_KEYWORD()“. Abbildung 29 zeigt die Regeln für einen Ausdruck in Verilog und einen in SystemVerilog.

```
241    "automatic"     { return TOK_AUTOMATIC; }
242
243    "unique"        { SV_KEYWORD(TOK_UNIQUE); }
```

Abbildung 29: Ausschnitt der Regeln für Ausdrücke von Verilog und SystemVerilog

Die Funktion SV_KEYWORD überprüft, ob der Lexer mit Unterstützung für SystemVerilog ausgeführt werden wird oder nicht. Wenn die Unterstützung nicht gesetzt wurde, wird eine Ausgabe geloggt und ein Token „TOK_ID“ zurückgegeben, wie auch in Abbildung 30 zu sehen ist.

```
65  #define SV_KEYWORD(_tok) \
66      if (sv_mode) return _tok; \
67      log("Lexer warning: The SystemVerilog keyword `%s' (at %s:%d) is not "\
68              "recognized unless read_verilog is called with -sv!\n", yytext, \
69              AST::current_filename.c_str(), frontend_verilog_yyget_lineno()); \
70      yylval->string = new std::string(std::string("\\") + yytext); \
71      return TOK_ID;
```

Abbildung 30: Funktion SV_KEYWORD im Verilog Lexer

Wenn für einen Ausdruck, der Wert gespeichert werden soll, wird ein String Objekt mit dem Inhalt von „yytext“ erzeugt und in „yylval“ überführt. Abbildung 31 zeigt das anhand der Regel für „UNSIGNED_NUMBER“.

```
305  {UNSIGNED_NUMBER} {
306      yylval->string = new std::string(yytext);
307      return TOK_CONSTVAL;
308  }
```

Abbildung 31: Regeln für UNSIGNED_NUMBER im Verilog Lexer

Compiler Direktiven, wie „file_push“, „file_pop“, die vom Präprozessor zum Code hinzugefügt wurden, werden mit den Stapeln „fn_stack“ und „ln_stack“ verarbeitet. Bei „file_push wird zuerst der Name der aktuellen Datei und die aktuelle Zeilennummer auf die zwei Stapeln gelegt. Anschließend wird der Dateiname auf den von „file_push“ und die Zeilennummer auf 0 gesetzt.

Für das Speichern der Zeilennummer werden die Variablen „yylloc“ und „real_location“ verwendet. Beim „Verlassen“ der Datei mit „file_pop“, werden Dateiname und Zeilennummer aus dem Stapel für die verwendetet Variablen übernommen und aus dem Stapel entfernt.

Abbildung 32 zeigt die Regeln für „file_push“ und „file_pop“. Dabei ist zu erkennen, dass vor dem jeweiligen Ausdruck „INITIAL“ und „SYNOPSYS_TRANSLATE_OFF“ vorkommen. Laut Yosys Handbuch, wird dies zur Handhabung von Kommentaren, wie „// synopsys translate_off“ verwendet, die als „`ifdef“ fungieren [1, p. 56].

```
141  <INITIAL,SYNOPSYS_TRANSLATE_OFF>"`file_push "[^\n]* {
142      fn_stack.push_back(current_filename);
143      ln_stack.push_back(frontend_verilog_yyget_lineno());
144      current_filename = yytext+11;
145      if (!current_filename.empty() && current_filename.front() == '"')
146          current_filename = current_filename.substr(1);
147      if (!current_filename.empty() && current_filename.back() == '"')
148          current_filename = current_filename.substr(0, current_filename.size()-1);
149      frontend_verilog_yyset_lineno(0);
150      yylloc->first_line = yylloc->last_line = 0;
151      real_location.first_line = real_location.last_line = 0;
152  }
153
154  <INITIAL,SYNOPSYS_TRANSLATE_OFF>"`file_pop"[^\n]*\n {
155      current_filename = fn_stack.back();
156      fn_stack.pop_back();
157      frontend_verilog_yyset_lineno(ln_stack.back());
158      yylloc->first_line = yylloc->last_line = ln_stack.back();
159      real_location.first_line = real_location.last_line = ln_stack.back();
160      ln_stack.pop_back();
161  }
```

Abbildung 32: Regeln für file_push und file_pop im Verilog Lexer

Weil der Verilog Lexer den Inputstream „lexin“ nutzt und nicht „yyinput“ bzw. „yyin“ verwendet, definiert der Verilog Lexer eine Funktion in Abbildung 33, die Fehlermeldungen unterdrückt.

```
600  // this is a hack to avoid the 'yyinput defined but not used' error msgs
601  void *frontend_verilog_avoid_input_warnings() {
602      return (void*)&yyinput;
603  }
```

Abbildung 33: Routine frontend_verilog_avoid_input_warnings()

Der Verilog Parser

Der Aufbau eines Bison-Programms wurde in Kapitel 2 beschrieben. Dieses Kapitel erläutert Abschnitte aus dem Bison-Programm in Yosys, um die Erzeugung des Syntaxbaumes zu verstehen. Dazu wird zunächst betrachtet welche Aktionen in den Produktionsregeln durchgeführt werden anhand der Produktionsregel für „basic_expr : basic_expr ‚&‘ attr basic_expr“ in Abbildung 34. Der Ausdruck ‘&‘ stellt ein Token dar und „basic_expr“ und „attr“ Nicht-Terminalsymbole.

```
3260  basic_expr '&' attr basic_expr {
3261      $$ = new AstNode(AST_BIT_AND, $1, $4);
3262      SET_AST_NODE_LOC($$, @1, @4);
3263      append_attr($$, $3);
3264  } |
```

Abbildung 34: Aktionsteil für die Produktionsregel mit dem Knotentyp AST_BIT_AND

Es wird ein AstNode Objekt vom Typ „AST_BIT_AND“ erstellt und die Werte der zwei „basic_expr“ Symbole, auf denen mit $1 und $4 zugegriffen wird, werden als Parameter übergeben. Bei der Untersuchung von AstNode später in diesem Kapitel erkennt man, dass die Objekte von $1 und $4 als Children im erzeugten AstNode Objekt gesetzt werden. Mit $$ wird auf den Wert des Symbols auf der linken Seite der Produktionsregel zugegriffen, was in diesem Fall ein Symbol basic_expr ist und das erzeugte AstNode Objekt zugewiesen bekommt. Symbole können also AstNode Objekte repräsentieren. Anschließend wird die Funktion SET_AST_NODE_LOC() angewendet, die die Position im Code für den erstellten Knoten setzt. Zum Schluss fügt die Funktion append_attr() die Attribute in „attr“ zu den erstellten Knoten hinzu. Mit der Erkenntnis, dass Symbole AstNode Objekte darstellen können, lässt sich ein Teil der Produktionsregeln zusammenfassen, indem man untersucht, welche AstNode Objekte sie repräsentieren. In Tabelle 3 sind alle Symbole zusammengefasst die AstNode Objekte darstellen und welchen Typ diese besitzen. Dabei übernehmen einige Symbole, Symbole mit AstNode Objekten.

Symbole, die AstNode Objekte darstellen	*AstNodeType von den AstNode Objekten oder welches Symbol ersetzt wird*
non_opt_range	AST_RANGE
non_opt_multirange	AST_MULTIRANGE
range	Ersetzt non_opt_range
range_or_multirange	Ersetzt non_opt_range und non_opt_multirange
non_io_wire_type	AST_WIRE
io_wire_type	AST_WIRE
typedef_base_type	AST_WIRE
wire_type	Ersetzt io_wire_type und non_io_wire_type
expr	Ersetzt basic_expr oder erzeugt AstNode Objekt mit AST_TERNARY
basic_expr	Ersetzt rvalue, expr oder stellt AstNode Objekt dar. Siehe Abschnitt unter Tabelle.
concat_list	AST_CONCAT
rvalue	AST_PREFIX, AST_IDENTIFIER
lvalue	Ersetzt rvalue
lvalue_concat_list	AST_CONCAT
struct_type	Reduziert struct_union
struct_union	AST_STRUCT, AST UNION
opt_enum_init	Ersetzt basic_expr
enum_type	AST_WIRE
enum_struct_type	Ersetzt enum_type, struct_type
func_return_type	AST_WIRETYPE
genvar_identifier	AST_IDENTIFIER

Tabelle 3: Liste mit Symbolen, die AstNode Objekte darstellen

Das Symbol „basic_expr“ ist in diesem Fall speziell, da abhängig der Regel, unterschiedliche AstNode Objekte darstellt werden. Dazu gehören AstNode Objekte für Operationen wie AST_BIT_AND, AST_REDUCE_OR, AST_SHIFT_LEFT, AST_ADD und weitere Typen. Eine Liste aller AstNode Typen ist im Yosys Handbuch aufgelistet [1, pp. 66-67].

Die Symbole oben in der der Tabelle sind nicht alle, die AstNode Objekte mit den Produktionsregeln erzeugen. Andere Symbole erzeugen AstNode Objekte, die zuerst in Buffer Objekten oder mit dem Stapel „ast_stack“ gespeichert werden. Das dazugehörige Symbol selbst besitzt als Wert kein AstNode Objekt. Diese Symbole können als eine weitere Gruppe zusammengefasst werden und sind in Tabelle 4 mit den Typen, der AstNode Objekte aufgelistet. Sie zeigt alle möglichen Typen an, die abhängig der Produktionsregel erzeugt werden.

Symbole in der AstNode Objekte erzeugt werden	*AstNodeType der Objekte*
module	AST_MODULE
single_module_para	AST_PARAMATER, AST_LOCALPARAM
module_arg_opt_assignment	AST_INITIAL, AST_BLOCK, AST_ASSIGN_LE, AST_ASSIGN
module_arg	AST INTERFACEPORT, AST_INTERFACEPORT-TYPE
interface	AST_INTERFACE
bind_directive	AST_BIND, AST_CELLTYPE
bind_target_instance	AST_IDENTIFIER
checker_decl	AST_GENBLOCK
task_func_decl	AST_DPI_FUNCTION, AST_TASK, AST_FUNCTION
task_func_port	AST WIRE
specify_item	AST_CELL, AST_CELLTYPE, AST_PARASET, AST_ARGUMENT
param_real	AST_REALVALUE
param_type	AST_WIRETYPE
param_decl	AST_PARAMETER
localparam_decl	AST_LOCALPARAM
single_param_decl_ident	AST_PARAMETER
single_defparam_dec	AST_DEFPARAM
enum_type	AST_ENUM, AST_ENUM_ITEM
enum_name_decl	AST_NONE
struct_union	AST_STRUCT, AST_UNION
wire_decl	AST_WIRE, AST_ASSIGN, AST_IDENTIFIER
wire_name_and_opt_assign	AST_IDENTIFIER, AST_FCALL, AST_ASSIGN, AST_ASSIGN_LE, AST_BLOCK, AST_INITIAL
ssign_expr	AST_ASSIGN
typedef_base_type	AST_WIRE
cell_stmt	AST_CELL, AST_CELLTYPE, AST_PRIMITIVE
single_cell_arraylist	AST_CELLARRAY

cell_parameter	AST_PARASET
cell_port	AST_ARGUMENT, AST_IDENTIFIER
always_stmt	AST_ALWAYS, AST_BLOCK, AST_INITIAL
always_event	AST_POSEDGE, AST_NEGEDGE, AST_EDGE
modport_stmt	AST_MODPORT
modport_member	AST_MODPORTMEMBER
assert und assert_property	AST_ASSUME, AST_ASSERT, AST_FAIR, AST_LIVE, AST_COVER
assert_property	AST_ASSUME, AST_ASSERT, AST_FAIR, AST_LIVE
simple_behavioral_stmt	AST_ASSIGN_EQ, AST_ASSIGN_LE, AST_ADD AST_SUB, AST_TO_UNSIGNED
for_initialization	AST_IDENTIFIER, AST_ASSIGN_EQ, AST_BLOCK
behavioral_stmt	AST_TCALL, AST_BLOCK, AST_FOR, AST_WHILE, AST_REPEAT, AST_CASE, AST_COND ,AST_RE-DUCE_BOOL
optional_else	AST_BLOCK, AST_COND, AST_DEFAULT
case_item	AST_CONDX, AST_CONDZ, AST_COND ,AST_BLOCK
gen_case_item	AST_CONDX, AST_CONDZ, AST_COND
case_expr_list	AST_DEFAULT, AST_IDENTIFIER
genvar_initialization	AST_GENVAR, AST_ASSIGN_EQ
gen_stmt	AST_GENFOR, AST_GENIF, AST_GENCASE, AST_TECALL
gen_block, gen_stmt_block	AST_GENBLOCK

Tabelle 4: Liste mit Symbolen, die AstNode Objekte erzeugen, aber nicht darstellen

Die restlichen Symbole, bzw. Produktionsregeln erstellen keine AstNode Objekte, sondern stellen für diese Daten bereit, die in dem AstNode Objekten gespeichert werden. Als Nächstes wird untersucht, wie der Syntaxbaum erstellt wird. Im VerilogFrontend wird ein AstNode Objekt vom Typ AST_DESIGN erzeugt und zu „current_ast“ zugewiesen, wie in Abbildung 35 dargestellt ist.

```
470        current_ast = new AST::AstNode(AST::AST_DESIGN);
```

Abbildung 35: current_ast in verilog_frontend.cc

Betrachtet man die erste Produktionsregel im Verilog Parser in Abbildung 36, zeigt sich für „input“, das Objekt „current_ast“ im Stapel „ast_stack“ abgelegt wird. Der Verilog Parser besitzt keine Start-Produktionsregel, die mit „%start“ definiert werden kann [3, pp. 159-160]. Das bedeutet es werden die Produktionsregeln von oben nach unten durchgegangen.

```
419   input: {
420       ast_stack.clear();
421       ast_stack.push_back(current_ast);
422   } design {
423       ast_stack.pop_back();
424       log_assert(GetSize(ast_stack) == 0);
425       for (auto &it : default_attr_list)
426           delete it.second;
427       default_attr_list.clear();
428   };
```

Abbildung 36: Regel für "input" im Verilog Parser

Die Aktion, bei der „current_ast“ auf den Stapel gelegt wird, steht vor dem Symbol „design“. Das würde bedeuten, dass der Verilog Parser diese Produktionsregel zuerst anwendet. Weil der Verilog Lexer am Anfang noch keine Tokens eingelesen hat, läuft der Parser weiter. Erst wenn alle Token eingelesen sind und die Produktionsregel für „input“ angewendet wird, ist das Parsen abgeschlossen. Für ein Symbol „design“ auf der rechten Seite, wird ein Element aus dem Stapel entfernt und die Hilfsvariable „default_attr_list“ geleert.

In Abbildung 37 ist ein Ausschnitt für die Ausgabe beim Parsen dargestellt und belegt, dass die erste Regel beim Starten angewendet wird. Der Stack enthält keine Elemente und der Parser reduziert anhand der ersten Regel.

```
Starting parse
Entering state 0
Stack now 0
Reducing stack by rule 1 (line 419):
-> $$ = nterm $@1 (1.1: )
```

Abbildung 37: Ausgabe beim Parsen

Sieht man sich die Regel für das Symbol „module“ in Abbildung 38 und für „assign_expr“ in Abbildung 39 an, erkennt man, wie der Syntaxbaum aufgebaut wird. Der Stapel „ast_stack“, der ein AstNode AST_DESIGN enthält, dient beim Parsen als Hilfsobjekt. In der Regel für „module“ wird ein AstNode Objekt vom Typ AST_MODULE erstellt und als Children zum AstNode Objekt AST_DESIGN hinzugefügt. Anschließend wird das Objekt auf dem Stack gelegt.

```
530   module:
531       attr TOK_MODULE {
532           enterTypeScope();
533       } TOK_ID {
534           do_not_require_port_stubs = false;
535           AstNode *mod = new AstNode(AST_MODULE);
536           ast_stack.back()->children.push_back(mod);
537           ast_stack.push_back(mod);
538           current_ast_mod = mod;
```

Abbildung 38: Ausschnitt für Regel "module" im Verilog Parser

Für die Produktionsregel „assign_expr“ wird ein AstNode Objekt vom Typ AST_ASSIGN erzeugt und als Children zum letzten AstNode Objekt im Stapel hinzugefügt.

```
2051 assign_expr_list:
2052     assign_expr | assign_expr_list ',' assign_expr;
2053
2054 assign_expr:
2055     lvalue '=' expr {
2056         AstNode *node = new AstNode(AST_ASSIGN, $1, $3);
2057         SET_AST_NODE_LOC(node, @$, @$);
2058         ast_stack.back()->children.push_back(node);
2059     };
```

Abbildung 39: Regel für "assign_expr" im Verilog Parser

Die AstNode Datenstruktur

Aus der Deklaration AstNode in Abbildung 41 ist zu erkennen, dass AstNode universal aufgebaut ist, sodass alle Knotentypen, die in AstNodeType definiert sind, unterstützt werden. Wesentlich speichert ein AstNode Objekt folgende Daten:

- Der Knotentyp „type“ vom Typ AstNodeType
- Ein Vektor „children“ vom Typ AstNode* für Unterknoten
- Ein Container „attributes“ für Attribute als Tupel aus RTLI::IdString und AstNode*
- Ein String Objekt „str“ für die Bezeichnung
- Ein Vektor „bits“ vom Typ RTLIL::State, eine double Variable „realvalue“ und ein uint32_t variable „integer“ für Werte
- Integer Variablen „port_id“, „range_left“ und „range_right“ für Angaben der Breite eines Signals und eine Port ID.
- Flags für Eigenschaften wie Input, Output, Register, Signed, etc.
- Ein String Objekt „filename“ für Angabe der Quelldatei
- Ein Objekt „location“ vom Typ AstSrcLocType für die Position im Quellcode

Zusätzlich enthält die Datenstruktur Hilfsvariablen und Member Funktionen. Unter den Member Funktionen gehören die Funktion simplify() und genRTLIL(). In Abbildung 40 ist die Deklaration von genRTLIL() in AstNode abgebildet, Abbildung 41 stellt einen Ausschnitt mit den Member Variablen dar.

```
299     RTLIL::SigSpec genRTLIL(int width_hint = -1, bool sign_hint = false);
300     RTLIL::SigSpec genWidthRTLIL(int width, bool sgn, const dict<RTLIL::SigBit, RTLIL::SigBit> *new_subst_ptr = NULL);
```

Abbildung 40: Deklaration von genRTLIL in AstNode in ast.h

```
178    struct AstNode
179    {
180        // for dict<> and pool<>
181        unsigned int hashidx_;
182        unsigned int hash() const { return hashidx_; }
183
184        // this nodes type
185        AstNodeType type;
186
187        // the list of child nodes for this node
188        std::vector<AstNode*> children;
189
190        // the list of attributes assigned to this node
191        std::map<RTLIL::IdString, AstNode*> attributes;
192        bool get_bool_attribute(RTLIL::IdString id);
193
194        // node content - most of it is unused in most node types
195        std::string str;
196        std::vector<RTLIL::State> bits;
197        bool is_input, is_output, is_reg, is_logic, is_signed, is_string, is_wand, is_wor, range_valid,
198        int port_id, range_left, range_right;
199        uint32_t integer;
200        double realvalue;
201        // set for IDs typed to an enumeration, not used
202        bool is_enum;
```

Abbildung 41: Deklaration von AstNode in ast.h

Der Typ AstNodeType definiert den Typ eines AstNode Objekts. Schaut man sich die Deklaration von AstNodeType in Abbildung 42 an, erkennt man, dass es ein Enum ist und keine anderen Daten außer den Typ angibt. Im Handbuch von Yosys ist eine Tabelle mit allen Knotentypen und Angabe, welches Element in Verilog es repräsentiert aufgelistet [1, pp. 66-67].

```
38    namespace AST
39    {
40        // all node types, type2str() must be extended
41        // whenever a new node type is added here
42        enum AstNodeType
43        {
44            AST_NONE,
45            AST_DESIGN,
46            AST_MODULE,
47            AST_TASK,
48            AST_FUNCTION,
49            AST_DPI_FUNCTION,
50
51            AST_WIRE,
52            AST_MEMORY,
53            AST_AUTOWIRE,
```

Abbildung 42: Ausschnitt von AstNodeType in ast.h

3.1.4 AST Frontend Aufruf und Beenden des Verilog Frontends

Dieser Abschnitt zeigt, dass das AST Frontend im Verilog Frontend mit der Funktion process() in Abbildung 43 aufgerufen wird. Neben den Flags für Optionen von Yosys wird das RTLIL::Design Objekt „design“ und der Syntaxbaum in „current_ast“ an den AST Frontend übergeben.

```
513        AST::process(design, current_ast, flag_dump_ast1, flag_dump_ast2, flag_no_dump_ptr, flag_dump_vlog1,
514         flag_dump_vlog2, flag_dump_rtlil, flag_nolatches, flag_nomeminit, flag_nomem2reg, flag_mem2reg, flag_noblackbox,
515          lib_mode, flag_nowb, flag_noopt, flag_icells, flag_pwires, flag_nooverwrite, flag_overwrite, flag_defer, default_nettype_wire);
```

Abbildung 43: Aufruf des AST Frontends in VerilogFrontend::execute()

Die Hauptaufgabe des AST Frontends ist die Generierung von RTLIL Objekten aus dem Syntaxbaum. Nach der Ausführung des AST Frontends ist das Einlesen und die Überführung in RTLIL abgeschlossen und Ressourcen werden wieder freigegeben, wie in Abbildung 44 zu sehen ist. Das AST Frontend selbst wird in Kapitel 3.2 behandelt.

```
517         if (!flag_nopp)
518             delete lexin;
519
520         // only the previous and new global type maps remain
521         log_assert(user_type_stack.size() == 2);
522         user_type_stack.clear();
523
524         delete current_ast;
525         current_ast = NULL;
526
527         log("Successfully finished Verilog frontend.\n");
528     }
529 } VerilogFrontend;
```

Abbildung 44: Beenden des Verilog Frontends

3.2 Das AST Frontend

Die Analyse des Verilog Frontends hat gezeigt, dass durch den Aufruf des AST Frontends die Konvertierung von AST zu RTLIL umgesetzt wird. Laut dem Yosys Handbuch besteht die Umwandlung aus den Schritten „simplification“ und „RTLIL generation“ [1, p. 66]. In diesem Kapitel werden diese Prozesse im AST Frontend untersucht und die Schritte für die Generierung von RTLIL erläutert. Dazu wurde der Quelltext untersucht und das Yosys Handbuch studiert. Zusätzlich wurden verschiedene Verilog-Beispiele erstellt und die Ausgabefunktionen in Yosys eingesetzt, um eingelesene Verilog-Designs in AST und RTLIL anzuzeigen.

3.2.1 Beschreibung der Aufgaben

Ein RTLIL::Design Objekt dient als Container für RTLIL::Module Objekte [1, p. 33]. Aus dem AST werden RTLIL::Module Objekte erzeugt und dem RTLIL::Design Objekt hinzugefügt. Als Zweites besitzt das Frontend die Aufgabe den Syntaxbaum für den RTLIL Generator aufzubereiten. Laut dem Yosys Handbuch ist dieser Vorgang erforderlich, bevor der RTLIL Generator sie einliest und in RTLIL Objekte überführt [1, p. 67].

Im Verilog Frontend wurde erkannt, dass dieser den AST Frontend, wie in Abbildung 43 dargestellt wird, aufruft. Die zwei wichtigsten Parameter sind die Zeiger „design“ und „current_ast“. Der Zeiger „design“ zeigt auf das aktuelle RTLIL::Design Objekt, das während der Konvertierung mit Objekten befüllt wird. Der Zeiger „current_ast“ zeigt auf den abstrakten Syntaxbaum mit dem eingelesenen Design.

In der Funktion process() befindet sich eine Schleife, in der für jedes Modul als Knoten vom Typ AST_MODULE die Funktion process_module() aufgerufen wird. Dort erfolgt für das Modul die Vereinfachung und die Generierung von RTLIL mit dem Aufruf von simplify() und genRTLIL().

3.2.2 Vereinfachung

Die Vereinfachung eines Moduls erfolgt mit dem Aufruf von AST::simplify(). Der Aufruf ist in Abbildung 45 abgebildet und zeigt, dass so lange vereinfacht wird, bis die Funktion „false“ zurückgibt.

```
1058        while (ast->simplify(!flag_noopt, false, false, 0, -1, false, false)) { }
```

Abbildung 45: Aufruf der Vereinfachung in AST::process_module()

Laut Handbuch führt die Funktion folgender Vereinfachungen durch [1, pp. 67-68]:

1. Führt für alle Task- and Funktionsaufrufe „Inlining“ aus,
2. Wertet generate-Anweisungen aus und rollt alle for-Schleifen ab.
3. Führt „const folding“ aus, wo nötig.
4. Ersetzt AST_PRIMITIVE Knoten mit geeigneten AST_ASSIGN Knoten.
5. Ersetzt dynamischen Bitbereiche auf der linken Seite von Zuweisungen durch AST_CASE Knoten mit AST_COND Children für jeden möglichen Fall.
6. Erkennt Array-Zugriffsmuster, die für die RTLIL::Memory-Abstraktion zu kompliziert sind, und ersetzt Sie sie durch einen Satz von Signalen und Fällen für alle Lese- und/oder Schreibvorgänge.
7. Ansonsten werden Array-Zugriffe durch AST_MEMRD und AST_MEMWR Knoten ersetzt.
8. Auf alle Bereiche (Breite von Signalen und Bit-Auswahl) wird nicht nur „const folding“ ausgeführt, sondern (wenn ein konstanter Wert gefunden wird) auch in Member-Variablen in AST_RANGE Knoten geschrieben.
9. Alle Bezeichner werden aufgelöst und alle AST_IDENTIFIER-Knoten werden mit einem Zeiger auf den AST-Knoten versehen, der die Deklaration des Bezeichners enthält. Wurde keine Deklaration gefunden, wird ein AST_AUTOWIRE Knoten erstellt und für die Annotation verwendet.

Die Vereinfachung soll anhand der AST_RANGE Vereinfachung aus Punkt 8 verdeutlicht werden. In Abbildung 46 ist in Verilog ein Vektor A definiert der als Zuweisung den Wert 4’b0000 erhält. Der darunterliegende AST zeigt den Stand vor der Vereinfachung. Es ist zu sehen, dass AST_RANGE keine Angabe für die Breite besitzt, aber zwei Children mit AST_CONSTANT. Die Breite ist mit „int“ in den AST_CONSTANT Children angegeben, einmal mit dem Wert 3 und einmal mit dem Wert 0 wobei im letzteren Fall nur die Binärdarstellung und keine Integer bzw. Dezimaldarstellung der Konstante 0 erfolgt.

```
1 module range;
2 reg [3:0]A;
3 assign A = 4'b0000;
4 endmodule
Dumping AST before simplification:
    AST_MODULE </home/user/Desktop/test/range.v:1.1-4.10> [0x27ee770] str='\range'
      AST_WIRE </home/user/Desktop/test/range.v:2.10-2.11> [0x27ff070] str='\A' reg
        AST_RANGE </home/user/Desktop/test/range.v:0.0-0.0> [0x27ff1b0]
          AST_CONSTANT </home/user/Desktop/test/range.v:2.6-2.7> [0x27ff2f0] bits='00000000000000000000000000000011'(32) signed range=[31:0] int=3
          AST_CONSTANT </home/user/Desktop/test/range.v:2.8-2.9> [0x27ff490] bits='00000000000000000000000000000000'(32) signed range=[31:0]
      AST_ASSIGN </home/user/Desktop/test/range.v:3.8-3.19> [0x27eeb00]
        AST_IDENTIFIER </home/user/Desktop/test/range.v:3.8-3.9> [0x27fef30] str='\A'
        AST_CONSTANT </home/user/Desktop/test/range.v:3.12-3.19> [0x27fedf0] bits='0000'(4) range=[3:0]
--- END OF AST DUMP ---
```

Abbildung 46: AST_RANGE vor Vereinfachung

Schaut man sich in der Funktion simplify() den Code für die Behandlung von AST_RANGE in Abbildung 47 an, erkennt man, dass für die Speicherung der Breite die Variablen „range_left“ und „range_right“ genutzt werden. Im behandelten Beispiel besitzt AST_RANGE zwei Unterknoten vom Typ AST_CONSTANT. Für den ersten Unterknoten setzt die Vereinfachung „range_valid“ auf true und holt sich den linken Wert mit „children[0]->integer“, speichert ihn in „range_left“ ab. Für die rechte Angabe der Breite wird mit „children[1]->integer“ der Wert in „range_right“ gespeichert.

```
1877    // annotate constant ranges
1878    if (type == AST_RANGE) {
1879        bool old_range_valid = range_valid;
1880        range_valid = false;
1881        range_swapped = false;
1882        range_left = -1;
1883        range_right = 0;
1884        log_assert(children.size() >= 1);
1885        if (children[0]->type == AST_CONSTANT) {
1886            range_valid = true;
1887            range_left = children[0]->integer;
1888            if (children.size() == 1)
1889                range_right = range_left;
1890        }
1891        if (children.size() >= 2) {
1892            if (children[1]->type == AST_CONSTANT)
1893                range_right = children[1]->integer;
1894            else
1895                range_valid = false;
1896        }
1897        if (old_range_valid != range_valid)
1898            did_something = true;
1899        if (range_valid && range_right > range_left) {
1900            int tmp = range_right;
1901            range_right = range_left;
1902            range_left = tmp;
1903            range_swapped = true;
1904        }
1905    }
```

Abbildung 47: Vereinfachung für AST_RANGE

Somit wird für AST_RANGE die Breite aus den Children bestimmt und in AST_RANGE in Member Variablen gespeichert. Das deckt sich mit der Ausgabe nach der Vereinfachung in Abbildung 48, wo AST_RANGE in „range“ eine Angabe besitzt.

```
AST_RANGE </home/user/Desktop/test/range.v:0.0-0.0> [0x27ff1b0] basic_prep range=[3:0]
```

Abbildung 48: AST_RANGE nach Vereinfachung

3.2.3 Der RTLIL Generator

Die RTLIL Generierung erfolgt mit der Ausführung der Funktion genRTLIL(), die eine Member Funktion von AstNode ist. Der Aufruf erfolgt in drei Schleifen in der Funktion process_module(), welche in Abbildung 49 dargestellt sind. In diesen Schleifen werden die Children des Knoten AST_MODULE durchlaufen. Zuerst werden Unterknoten vom Typ AST_WIRE und AST_MEMORY in RTLIL überführt, anschließend alle anderen Typen bis auf AST_INITIAL, welche als Letzte umgewandelt werden.

```
1158        for (size_t i = 0; i < ast->children.size(); i++) {
1159            AstNode *node = ast->children[i];
1160            if (node->type == AST_WIRE || node->type == AST_MEMORY)
1161                node->genRTLIL();
1162        }
1163        for (size_t i = 0; i < ast->children.size(); i++) {
1164            AstNode *node = ast->children[i];
1165            if (node->type != AST_WIRE && node->type != AST_MEMORY && node->type != AST_INITIAL)
1166                node->genRTLIL();
1167        }
1168
1169        ignoreThisSignalsInInitial.sort_and_unify();
1170
1171        for (size_t i = 0; i < ast->children.size(); i++) {
1172            AstNode *node = ast->children[i];
1173            if (node->type == AST_INITIAL)
1174                node->genRTLIL();
1175        }
```

Abbildung 49: Aufruf von genRTLIL() in process_module() in ast.cc

Die Funktion genRTLIL

Schaut man sich die Funktion genRTLIL() in Abbildung 50 an, stellt sich heraus, dass die Funktion hauptsächlich aus einem großen Switch-Case Block besteht, der Aktionen für jeden Knotentyp ausführt.

```
1287    // create an RTLIL::Wire for an AST_WIRE node
1288 >  case AST_WIRE: {···
1315        break;
1316
1317    // create an RTLIL::Memory for an AST_MEMORY node
1318 >  case AST_MEMORY: {···
1346        break;
1347
1348    // simply return the corresponding RTLIL::SigSpec for an AST_CONSTANT node
1349    case AST_CONSTANT:
1350    case AST_REALVALUE:
1351 >      {···
1368
1369 >  // simply return the corresponding RTLIL::SigSpec for an AST_IDENTIFIER node·
1372    case AST_IDENTIFIER:
```

Abbildung 50: Ausschnitt aus genRTLIL() in genrtlil.cc

Anhand eines Verilogbeispiels und des Switch-Case Blocks wird für einen Knoten vom Typ AST_WIRE und vom Typ AST_ASSIGN nachvollzogen, wie der AST in RTLIL umgewandelt wird. Abbildung 51 zeigt eine einfache Zuweisung mit assign in einem Modul und den dazugehörigen AST.

```
1 module test(input A, output B);
2 assign A = B;
3 endmodule
```

```
AST_MODULE </home/user/Desktop/test/assign2.v:1.1-3.10> [0x2f81650] str='\test' basic_prep
  AST_WIRE </home/user/Desktop/test/assign2.v:1.19-1.20> [0x2f817d0] str='\A' input basic_prep port=1 range=[0:0]
  AST_WIRE </home/user/Desktop/test/assign2.v:1.29-1.30> [0x2f81970] str='\B' output basic_prep port=2 range=[0:0]
  AST_ASSIGN </home/user/Desktop/test/assign2.v:2.8-2.13> [0x2f92050] basic_prep
    AST_IDENTIFIER </home/user/Desktop/test/assign2.v:2.8-2.9> [0x2f91dd0 -> 0x2f817d0] str='\A' basic_prep
    AST_IDENTIFIER </home/user/Desktop/test/assign2.v:2.12-2.13> [0x2f91f10 -> 0x2f81970] str='\B' basic_prep
```

Abbildung 51: Einfache assign Zuweisung mit AST

Die ersten zwei Children, die vom RTLIL Generator umgewandelt werden, sind AstNode Objekte vom Typ AST_WIRE. Der Abschnitt für AST_WIRE in Abbildung 52 zeigt das für die Bezeichnung in „str“ ein RTLIL::IdString Objekt erzeugt wird und mit check_unique_id() überprüft wird, ob diese Bezeichnung im Modul schon vergeben wurde. Anschließend wird mit addWire() ein neues RTLIL::Wire Objekt erzeugt, dass zum RTLIL::Module hinzugefügt wird. Zuletzt fehlen noch die Attribute, die mit der Funktion set_src_attr() im Wire übernommen werden und Eigenschaften des Wires, die mit Zugriff auf dem jeweiligen Variablen im Wire Objekt gesetzt werden.

```
1288        case AST_WIRE: {
1289                if (!range_valid)
1290                    log_file_error(filename, location.first_line, "Signal `%s' with non-constant wid
1291
1292                if (!(range_left + 1 >= range_right))
1293                    log_file_error(filename, location.first_line, "Signal `%s' with invalid width ra
1294
1295                RTLIL::IdString id = str;
1296                check_unique_id(current_module, id, this, "signal");
1297                RTLIL::Wire *wire = current_module->addWire(id, range_left - range_right + 1);
1298                set_src_attr(wire, this);
1299                wire->start_offset = range_right;
1300                wire->port_id = port_id;
1301                wire->port_input = is_input;
1302                wire->port_output = is_output;
1303                wire->upto = range_swapped;
1304                wire->is_signed = is_signed;
```

Abbildung 52: AST_WIRE im Switch-Case Block in genRTLIL ()

Nach den zwei AST_WIRE Knoten wird AST_ASSIGN ausgewertet. Es werden zwei RTLIL::SigSpec Objekte erstellt. Für den ersten Unterknoten wird genRTLIL() rekursiv angewendet und den zweiten genWidthRTLIL(). Im Yosys Handbuch wird über die Funktion genRTLIL() erwähnt, dass Sie ein SigSpec Objekt zurückgibt [1, p. 68]. Schaut man sich im RTLIL Generator die Behandlung des Knotentyps AST_IDENTIFIER und die Funktion genWidthRTLIL() an, kann man erkennen, dass der linke Unterknoten in ein SigSpec Objekt überführt wird. Der rechte Unterknoten wird nach Überführung in ein SigSpec Objekt auf die Größe der linken Seite angepasst. Abbildung 53 zeigt die Behandlung der Fälle AST_ASSIGN und AST_IDENTIFIER in der Funktion gen RTLIL() und der Hilfsfunktion genWidthRTLIL().

```
1369    // simply return the corresponding RTLIL::SigSpec for an AST_IDENTIFIER node
1370    // for identifiers with dynamic bit ranges (e.g. "foo[bar]" or "foo[bar+3:bar]") a
1371    // shifter cell is created and the output signal of this cell is returned
1372    case AST_IDENTIFIER:
1373        {
1374            RTLIL::Wire *wire = NULL;
1375            RTLIL::SigChunk chunk;
```

```
1892    case AST_ASSIGN:
1893        {
1894            RTLIL::SigSpec left = children[0]->genRTLIL();
1895            RTLIL::SigSpec right = children[1]->genWidthRTLIL(left.size(), true);
1896 >          if (left.has_const()) {···
1910            current_module->connect(RTLIL::SigSig(left, right));
1911        }
1912        break;
```

```
RTLIL::SigSpec AstNode::genWidthRTLIL(int width, bool sgn, const dict<RTLIL::SigBit, RTLIL::SigBit> *new_subst_ptr)
{
    const dict<RTLIL::SigBit, RTLIL::SigBit> *backup_subst_ptr = genRTLIL_subst_ptr;

    if (new_subst_ptr)
        genRTLIL_subst_ptr = new_subst_ptr;

    bool sign_hint = sgn;
    int width_hint = width;
    detectSignWidthWorker(width_hint, sign_hint);
    RTLIL::SigSpec sig = genRTLIL(width_hint, sign_hint);

    genRTLIL_subst_ptr = backup_subst_ptr;

    if (width >= 0)
        sig.extend_u0(width, is_signed);

    return sig;
}
```

Abbildung 53: genWidthRTLIL(), AST_ASSIGN, AST_IDENTIFIER in genRTLIL()

Arithmetische und logische Operationen erscheinen im AST als Knoten mit jeweiligem Typen. Diese werden in entsprechende Zellen, also RTLIL::Cell Objekte überführt. Für Operationen mit einem Operanden, wird die Hilfsfunktion uniop2rtlil() und für zwei Operanden binop2rtlil() genutzt. Grundlegend stellen RTLIL::Cell Objekte Komponenten dar, die nach dem Technologie Mapping als generische Zellen in der Netzliste auftauchen können. Das heißt, dass mit der Generierung von RTLIL::Cell Objekten, die ersten Schritte zur Synthese im Frontend durchgeführt werden. Die nachfolgende Tabelle gibt eine Übersicht darauf, in welche RTLIL Objekte AST-Kontentypen durch genRTLIL() übersetzt werden.

AST	**RTLIL**
AST_LOCALPARAM	RTLIL::Wire
AST_WIRE	RTLIL::Wire
AST_MEMORY	RTLIL::Memory
AST_CONSTANT AST_REALVALUE	RTLIL::SigSpec
AST_IDENTIFIER	RTLIL::SigSpec
AST_TO_SIGNED AST_TO_UNSIGNED AST_SELFSZ	RTLIL::SigSpec
AST_CAST_SIZE	RTLIL::SigSpec
AST_CONCAT	RTLIL::SigSpec

AST	RTLIL
AST_REPLICATE	RTLIL::SigSpec
AST_BIT_NOT	RTLIL::Cell, $not
AST_POS	RTLIL::Cell, $pos
AST_NEG	RTLIL::Cell, $neg
AST_BIT_AND	RTLIL::Cell, $and
AST_BIT_OR	RTLIL::Cell, $or
AST_BIT_XOR	RTLIL::Cell, $xor
AST_BIT_XNOR	RTLIL::Cell, $xnor
AST_REDUCE_AND	RTLIL::Cell, $reduce_and
AST_REDUCE_OR	RTLIL::Cell, $reduce_or
AST_REDUCE_XOR	RTLIL::Cell, $reduce_xor
AST_REDUCE_XNOR	RTLIL::Cell, $reduce_xnor
AST_REDUCE_BOOL	RTLIL::Cell, $reduce_bool
AST_SHIFT_LEFT	RTLIL::Cell, $shl
AST_SHIFT_RIGHT	RTLIL::Cell, $shr
AST_SHIFT_SLEFT	RTLIL::Cell, $sshl
AST_SHIFT_SRIGHT	RTLIL::Cell, $sshr
AST_SHIFTX	RTLIL::Cell, $shiftx
AST_SHIFT	RTLIL::Cell, $shift
AST_POW	RTLIL::Cell, $pw
AST_LT	RTLIL::Cell, $lt
AST_LE	RTLIL::Cell, $le
AST_EQ	RTLIL::Cell, $eq
AST_NE	RTLIL::Cell, $ne
AST_EQX	RTLIL::Cell, $eqx
AST_NEX	RTLIL::Cell, $nex
AST_GE	RTLIL::Cell, $ge
AST_GT	RTLIL::Cell, $gt

AST	RTLIL
AST_ADD	RTLIL::Cell, $add
AST_SUB	RTLIL::Cell, $not
AST_MUL	RTLIL::Cell, $not
AST_DIV	RTLIL::Cell, $not
AST_MOD	RTLIL::Cell, $not
AST_LOGIC_AND	RTLIL::Cell, $logic_and
AST_LOGIC_OR	RTLIL::Cell, $logic_or
AST_LOGIC_NOT	RTLIL::Cell, $logic_not
AST_TERNARY	RTLIL::Cell, $mux
AST_MEMRD	RTLIL::Cell, $memrd
AST_MEMINIT	RTLIL::Cell, $meminit
AST_ASSERT	RTLIL::Cell, $assert
AST_ASSUME	RTLIL::Cell, $assume
AST_LIVE	RTLIL::Cell, $live
AST_FAIR	RTLIL::Cell, $fair
AST_COVER	RTLIL::Cell, $cover
AST_ASSIGN	RTLIL::SigSig
AST_CELL	RTLIL::Cell
AST_ALWAYS	RTLIL::Process
AST_INITIAL	RTLIL::Process
AST_BIND	RTLIL::Binding
AST_FCALL	RTLIL::Cell

Tabelle 5: RTLIL Objekte mit dazugehörigen AstNode Typ

Der Prozessgenerator

Always- und initial-Blöcke tauchen im AST als Knoten vom Typ AST_ALWAYS und AST_INITIAL auf. Bei der Überführung in RTLIL wird auf diesen Knoten der Prozessgenerator angewendet. Relevant ist hierbei der Ablauf, mit dem die RTLIL Objekte aus den Unterknoten im AST erzeugt werden. Erläuterungen für die einzelnen RTLIL Objekte, sind im Kapitel „Die Datenstruktur RTLIL“ vorhanden. Abbildung 54 zeigt den Aufruf des Prozessgenerators mit dem jeweiligen Knoten.

```
2019    case AST_ALWAYS: {
2020            AstNode *always = this->clone();
2021            ProcessGenerator generator(always);
2022            ignoreThisSignalsInInitial.append(generator.outputSignals);
2023            delete always;
2024        } break;
2025
2026    case AST_INITIAL: {
2027            AstNode *always = this->clone();
2028            ProcessGenerator generator(always, ignoreThisSignalsInInitial);
2029            delete always;
2030        } break;
```

Abbildung 54: Aufruf des Prozessgenerators in genRTLIL

Zunächst wird anhand von Verilogbeispielen und den Hilfsausgaben von Yosys untersucht, welche RTLIL Objekte aus dem Syntaxbaum erstellt werden. Anschließend wird der Ablauf des Prozessgenerators mithilfe des Codes näher erläutert.

Als Beispiel wird die Verilog-Beschreibung eines D-FlipFlops aus Abbildung 55 untersucht. Das Modul dffp besitzt den Dateneingang D, den Ausgang Q, das Taktsignal CLK und einen always Block, der das FlipFlop beschreibt. Für jede positive Taktflanke wird das Signal an D auf Ausgang Q übernommen. Der dazugehörige AST Block besteht aus den zwei Unterknoten AST_POSEDGE, für das Signal in der Sensitivityliste, und AST_BLOCK. In RTLIL wird für den Block ein RTLIL::Process Objekt erzeugt, das zwei Zuweisungen und ein Objekt RTLIL::SyncRule beinhaltet.

Verilog

```
1 module dffp (D, CLK, Q);
2 input D;
3 input CLK;
4 output Q;
5 always @ (posedge CLK)
6 begin
7    Q <= D;
8 end
9 endmodule
```

AST

```
AST_ALWAYS </home/user/Desktop/test/dffp.v:5.1-8.4> [0x1910930] basic_prep
  AST_POSEDGE </home/user/Desktop/test/dffp.v:5.11-5.18> [0x1906820] basic_prep
    AST_IDENTIFIER </home/user/Desktop/test/dffp.v:5.19-5.22> [0x1914cb0 -> 0x1910d50] str='\CLK' basic_prep
  AST_BLOCK </home/user/Desktop/test/dffp.v:5.23-8.4> [0x1907080] basic_prep
    AST_BLOCK </home/user/Desktop/test/dffp.v:6.1-8.4> [0x1913c20] basic_prep
      AST_ASSIGN_LE </home/user/Desktop/test/dffp.v:7.4-7.10> [0x19132e0] basic_prep
        AST_IDENTIFIER </home/user/Desktop/test/dffp.v:7.4-7.5> [0x1913db0 -> 0x1910be0] str='\Q' basic_prep
        AST_IDENTIFIER </home/user/Desktop/test/dffp.v:7.9-7.10> [0x1913a90 -> 0x1910a70] str='\D' basic_prep
```

RTLIL

```
process $proc$/home/user/Desktop/test/dffp.v:5$1
  assign { } { }
  assign $0\Q[0:0] \D
  sync posedge \CLK
    update \Q $0\Q[0:0]
end
```

Abbildung 55: D-FlipFlop als Verilogcode, Ausschnitt in AST und RTLIL

Um nachzuvollziehen, warum und wie diese Konvertierung abläuft, wird der Code näher untersucht. Dieser zeigt, dass der Prozessgenerator ein neues RTLIL::Process Objekt mit einem leeren RTLIL::CaseRule Objekt „root_case“ erzeugt. Anschließend werden für die Ausgangsregister, also die linke Seite der Zuweisungen, temporäre Signale in Form von RTLIL::SigSpec Objekten erzeugt. Abbildung 56 zeigt die Stelle im Code.

```
323        // generate process and simple root case
324        proc = current_module->addProcess(stringf("$proc$%s:%d$%d", always->filename.c_str(), always->location.first_line, autoidx++));
325        set_src_attr(proc, always);
326 >      for (auto &attr : always->attributes) {···
332        current_case = &proc->root_case;
333
334        // create initial temporary signal for all output registers
335        RTLIL::SigSpec subst_lvalue_from, subst_lvalue_to;
336        collect_lvalues(subst_lvalue_from, always, true, true);
337        subst_lvalue_to = new_temp_signal(subst_lvalue_from);
338        subst_lvalue_map = subst_lvalue_from.to_sigbit_map(subst_lvalue_to);
```

Abbildung 56: Erzeugung von RTLILL::Process mit dem root_case

Für ein Ausgangssignal mit der Bezeichnung \<name> gibt es ein temporäres Signal $<number>\<name> [1, p. 68]. Dies führt dazu, dass Zuweisungen mit dem temporären Signal angepasst werden müssen. Dies geschieht mit den Hilfsobjekten „subst_lvalue_map“, „subst_lvalue_from“ und „subst_lvalue_to“. „subst_lvalue_from“ enthält das ursprüngliche Signal, „subst_lvalue_to“ enthält das temporäre Signal und „subst_lvalue_map“ dient als Container für die Signale, die als Tupel gespeichert werden. Wird nun eine Zuweisung untersucht, wird mit Hilfe des Containers das eigentliche Signal durch das temporäre Signal ersetzt. Dies geschieht durch das Hinzufügen einer Zuweisung mit dem temporären Signal und entfernen der ursprünglichen Zuweisung. Ähnlich werden auch die Signale der rechten Zuweisungen behandelt. Wird das temporäre Signal auf der rechten Seite einer Zuweisung genutzt, muss es mit dem ursprünglichen Signal ersetzt werden. Die genutzten Hilfsobjekte lauten „subst_rvalue_map“, „subst_rvalue_from“ und „subst_rvalue_to“.

Nach der Erstellung der temporären Signale, bearbeitet der Prozessgenerator die Signale der Sensitivityliste, die in der AST Struktur als AST_POSEDGE oder AST_NEGEDGE Knoten vorkommen. Für jedes Signal wird ein SyncRule Objekt erzeugt. Das Objekt besitzt einen Typen, ein Signal und Aktionen. Der Typ des Signals wird anhand des Typs des Knoten festgelegt. Ein SyncRule Objekt bekommt für einen AST_POSEDGE Knoten den SyncType-Typ „STp“ und bei einem AST_NEGEDGE Knoten eine SyncRule vom Typ „STn“. Das Signal selbst wird mit der Anwendung der Funktion genRTLIL() in ein SigSpec Objekt umgewandelt, das von der Funktion zurückgegeben wird. Anstatt dass die Zuweisungen des Prozesses direkt in das SyncRule Objekt hinzugefügt werden, enthält das SyncRule Objekt Zuweisungen der temporären Signale zu den ursprünglichen Signalen in Form von Aktionen. Aktionen sind SigSig Objekte, die Tupel aus zwei SigSpec Objekten darstellen.

Abbildung 57 zeigt den Codeabschnitt im Prozessgenerator für die Generierung von flankengesteuerten SyncRule Objekten und Abbildung 58 zeigt die Umwandlung zu einem SyncRule Objekt anhand des Beispiels in Abbildung 55.

```
374            RTLIL::SyncRule *syncrule = new RTLIL::SyncRule;
375            syncrule->type = child->type == AST_POSEDGE ? RTLIL::STp : RTLIL::STn;
376            syncrule->signal = child->children[0]->genRTLIL();
377            if (GetSize(syncrule->signal) != 1)
378                log_file_error(always->filename, always->location.first_line, "Found posedge/negedge event on a signal that is not 1 bit wide!\n");
379            addChunkActions(syncrule->actions, subst_lvalue_from, subst_lvalue_to, true);
380            proc->syncs.push_back(syncrule);
```

Abbildung 57: Erstellen von SyncRule Objekten im Prozessgenerator

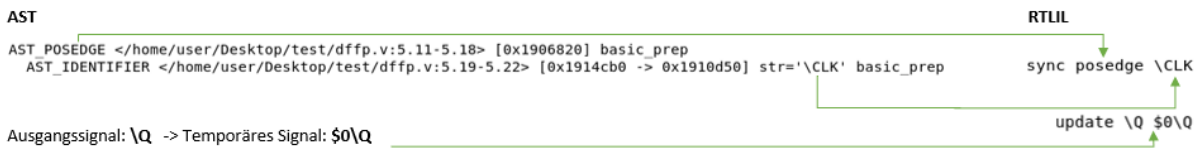

Abbildung 58:Umwandlung in SyncRule Objekt anhand des Beispiels mit dem Modul dffp

Wenn der Prozessgenerator kein Flankensignal findet bzw. keine flankengesteuerte SyncRule Objekte erstellt hat, sondern einen globalen Takt in Form eines AST-Knotens vom Typ AST_EDGE vorfindet, wird ein SyncRule Objekt des Typs „STg“ für eine globale Clock erstellt. Trifft keiner der Fälle zu, also weder flankengesteuert noch globale Clock, wird das SyncRule Objekt auf „STa“ gesetzt. Damit wird in RTLIL signalisiert, dass die Logik immer aktiv ist.

Als nächsten Schritt werden für die temporären Signale Zuweisungen mit den eigentlichen Signalen erstellt, die als Aktionen im root_case liegen. Damit kann mit dem Knoten AST_BLOCK fortgefahren werden, der mit der Funktion processAST() bearbeitet wird. Zuerst wird auf die Children des AST_BLOCK Knotens die Funktion processAST() rekursiv angewendet. Non-Blocking Zuweisungen für sequenzielle Logik[1] entsprechen AST_ASSIGN_LE Knoten und Blocking Zuweisungen für kombinatorische Logik[2] entsprechen Knoten vom Typ AST_ASSIGN_EQ. Für beide Typen wird das linke und rechte Signal der Zuweisung evaluiert. Das bedeutet, dass für das linke Signal die Funktion genRTLIL() angewendet wird. Das zurückgegebene SigSpec Objekt entspricht dem temporären Signal, das im Prozessgenerator erstellt wurde. Bei einer Zuweisung ist es wichtig, dass beide Signale die gleiche Bitbreite verwenden. Deshalb wird für die rechte Seite die Funktion genWidthRTLIL() angewendet. Das Signal wird mit Null-Bits erweitert oder es werden Bits entfernt, bis es die gleiche Größe wie das linke Signal der Zuweisung besitzt. Anschließend wird in der Funktion genWidthRTLIL() die Funktion genRTLIL() auf das Signal angewendet und ein SigSpec Objekt zurückgegeben. Damit existieren für jede Zuweisung zwei SigSpec Objekte, die als SigSig Objekt zusammengefasst werden und als Aktion zum aktuellen CaseRule Objekt hinzugefügt werden. Vorherige Zuweisungen für das gleiche Signal werden überschrieben, indem diese entfernt werden. In Verilog werden Blocking Zuweisungen unmittelbar ausgeführt und unterbrechen (blocken) den prozeduralen Ablauf. Deshalb wird für den Fall für Blocking Zuweisungen das linke Signal angepasst, was einer unmittelbaren Ausführung gleichkommt. Das linke Signal wird von „subst_rvalue_from“ und „subst_rvalue_to“ entfernt. Anschließend wird das linke Signal „subst_rvalue_from“ zugewiesen und das rechte Signal „subst_rvalue_to“.

Ein Knoten vom Typ AST_CASE sagt aus, dass es sich um eine Verzweigung handelt. In RTLIL werden Verzweigungen mit SwitchRule und CaseRule Objekten realisiert. Verschachtelte Verzweigungen führen in eine Verschachtelung von SwitchRule und CaseRule Objekten.

[1] Non-Blocking Zuweisungen besitzen „<=“

[2] Blocking Zuweisungen besitzen „=“

Als Beispiel wird der Code in Abbildung 59 untersucht. Ein AstNode Objekt vom Typ AST_CASE enthält drei Unterknoten, wenn eine If-Else Bedingung beschrieben wird. Der linke Unterknoten beschreibt das auszuwertende Signal für die Bedingung. In diesem Fall stellt der AST-Knoten AST_EQ mit einem Knoten AST_IDENTIFIER das Signal „clk“ und einen Knoten AST_CONSTANT den Wert „1’b1“ dar. Die zwei Knoten vom Typ AST_COND stellen die zwei Fälle if und else dar.

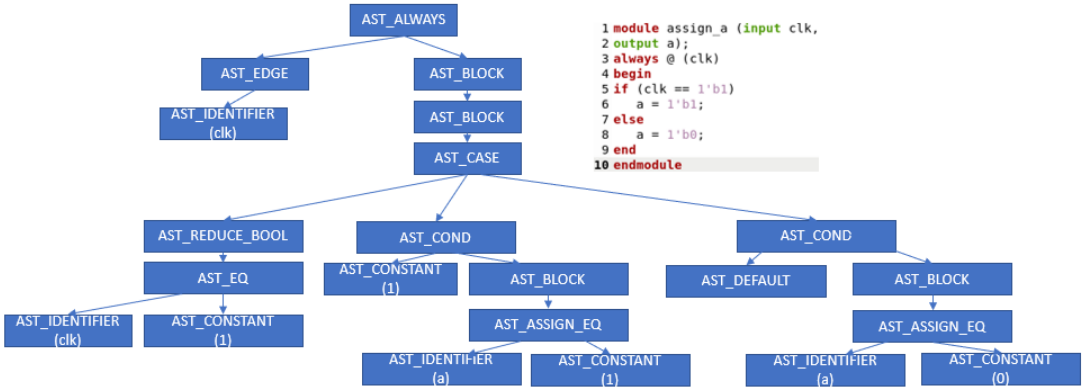

Abbildung 59: Darstellung der AST Struktur einer Verzweigung mit Verilogbeispiel

Für die If-Bedingung wird ein SwitchRule Objekt erzeugt. Es enthält das Signal, das ausgewertet wird, als SigSpec Objekt und Fälle mit den Aktionen als CaseRule Objekte. Dann werden temporäre Signale für die linke Seite der Zuweisungen erzeugt. Danach werden die Fälle bearbeitet, die als AST_COND Knoten vorhanden sind. Für jeden Fall wird ein CaseRule Objekt erzeugt. Dieses Objekt besitzt die Bedingung, also den Zustand des Signals für eine erfüllte Bedingung, als SigSpec Objekt „compare“. Zuweisungen werden in einen SigSig Vektor im CaseRule Objekt gespeichert. Weitere Verzweigungen sind als Unterknoten im Knoten AST_COND gelistet und führen dazu, dass processAst() rekursiv auch für diesen Knoten ausgeführt wird. Andere Knotentypen werden ignoriert oder führen zum Abbruch. In Abbildung 60 ist die RTLIL Darstellung des Prozesses von dem Beispiel in Abbildung 59 dargestellt.

```
process $proc$/home/user/Desktop/test/assign1.v:3$1
  assign { } { }
  assign $0\a[0:0] $1\a[0:0]
  attribute \src "/home/user/Desktop/test/assign1.v:5.1-8.13"
  switch $eq$/home/user/Desktop/test/assign1.v:5$2_Y
    attribute \src "/home/user/Desktop/test/assign1.v:5.5-5.16"
    case 1'1
      assign { } { }
      assign $1\a[0:0] 1'1
    attribute \src "/home/user/Desktop/test/assign1.v:7.1-7.5"
    case
      assign { } { }
      assign $1\a[0:0] 1'0
  end
  sync always
    update \a $0\a[0:0]
end
```

Abbildung 60: Darstellung von RTLIL aus Beispiel in Abbildung 59

Ast Knoten vom Typ AST_MEMWR stellen Speicherschreibzugriffe in Verilog Arrays dar. Sie werden in RTLIL::MemWriteAction Objekte konvertiert, die als gesonderter Stapel in den SyncRule Objekte auftauchen. Zum Schluss werden Initialisierungssignale, die als SigSpec Objekt „initSyncSignals“ vorhanden sind, umgewandelt. Es wird eine neu SyncRule vom Typ „STi“ erstellt, der die Initialisierungen als Aktionen enthält.

Zusammenfassend kann festgehalten werden, dass bei der Überführung der AST Knoten in RTLIL, der Typ des Knotens den RTLIL Generator wesentlich beeinflusst. Der Prozessgenerator stellt die Struktur aus Process, SyncRule, CaseRule und SwitchRule mit den enthaltenen Objekten her. In Abbildung 61 wird vereinfacht in einem Diagramm dargestellt, welche RTLIL Objekte welches RTLIL::Process Objekt enthalten.

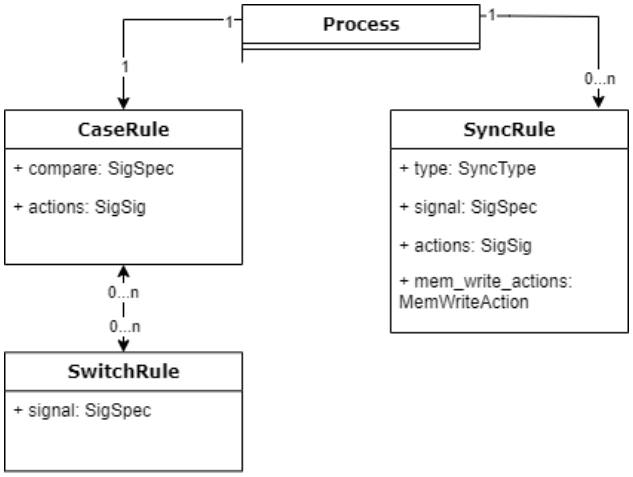

Abbildung 61: Vereinfachte Darstellung des RTLIL::Process Objekts mit Attributen

4 Die Datenstruktur RTLIL

Alle Frontends, Backends und Passes in Yosys arbeiten auf einem Hardwaredesign in RTLIL. Die einzige Ausnahme stellen die High-level Frontends dar, die als Zwischenschritt, vor der Generierung des RTLIL, die AST Struktur verwenden [1, p. 31]. RTLIL wurde von Claire Wolf für Yosys entwickelt und setzt sich aus den zwei Begriffen RTL und „Intermediate Language“ zusammen. Der Begriff Intermediate heißt „zwischen“ und beschreibt zutreffend, wofür RTLIL genutzt wird. RTLIL ist eine Zwischensprache und liegt zwischen HDL und der Netzliste, die Yosys als Ausgabe für die Synthese ausgibt. RTL steht für „Register-Transfer Level“ und ist die erste Abstraktion in Richtung einer Netzliste aus Registern, kombinatorischer Logik und Signalen [1, p. 18].

Dieses Kapitel beschreibt RTLIL Objekte und stellt eine Verbindung zu den Elementen in Verilog her, die sie repräsentieren. Damit liegt der Fokus auf der Menge von RTLIL Objekten, die ein Hardwaredesign beschreiben. Interne Funktionalitäten von Yosys, die von Passes genutzt werden, aber kein Bestandteil des Designs sind, werden nicht erläutert.

4.1 Die Objekte RTLIL::IdString und RTLIL::AttrObject

Alle RTLIL Objekte, die einen Namen besitzen, besitzen auch ein RTLIL::IdString Objekt. Er dient als Identifier des Objektes und lässt sich in zwei Arten aufteilen. Wenn die Bezeichnung mit „\“ anfängt, handelt es sich um einen Namen, der im Sourcecode definiert wurde und nicht von Yosys erstellt wurde. Wenn Yosys Namen generiert, ist das erste Zeichen in der Bezeichnung immer „$“ [1, p. 32].

Einige RTLIL Objekte besitzen auch Attribute, die gespeichert werden. Für diese Objekte gilt, dass sie eine Unterklasse von RTLIL::AttrObject sind. Wie in Abbildung 62 dargestellt, besitzt die Klasse RTLIL::AttrObject nur ein dict Objekt. Ein dict Objekt ist ein Container, der Schlüssel-Werte Paare speichert. In diesem Fall ist ein Objekt IdString der Schlüssel und Const der Wert.

```
695  struct RTLIL::AttrObject
696  {
697      dict<RTLIL::IdString, RTLIL::Const> attributes;
```

Abbildung 62: Deklaration von AttrObject in rtlil.h

4.2 Signale und Verbindungen

Für die Darstellung von Signalen in RTLIL wird der Datentyp SigSpec genutzt [1, p. 35]. Ein SigSpec Objekt ist ein Container für Vektoren aus SigChunk und SigBit Objekten. Eine Integervariable gibt die Breite des Signals an, wie auch der SigSpec Deklaration im Code-Ausschnitt in Abbildung 63 zu entnehmen ist.

```
804   struct RTLIL::SigSpec
805   {
806   private:
807       int width_;
808       unsigned long hash_;
809       std::vector<RTLIL::SigChunk> chunks_; // LSB at index 0
810       std::vector<RTLIL::SigBit> bits_; // LSB at index 0
```

Abbildung 63: Deklaration von SigSpec in rtlil.h

RTLIL::SigBit

Ein SigBit Objekt beschreibt genau ein Signalbit. Abhängig davon, ob das Signalbit einen konstanten Wert enthält oder ein Bit eines Wire darstellt, besitzt SigBit ein Wire Objekt oder nicht. Wenn das Bit einem Wire zugehörig ist, enthält das SigBit Objekt neben dem RTLIL::Wire Objekt eine Integervariable „offset“, welche die Position des Bits im Vektor angibt. Enthält das Signalbit einen konstanten Wert, ist das RTLIL::Wire Objekt NULL und SigBit besitzt ein Objekt des Typen RTLIL::State. Abbildung 64 zeigt die Deklaration des SigBit Objekts und in Abbildung 65 wird die Beschreibung anschaulicher anhand eines grafischen Beispiels veranschaulicht.

```
755   struct RTLIL::SigBit
756   {
757       RTLIL::Wire *wire;
758       union {
759           RTLIL::State data; // used if wire == NULL
760           int offset;        // used if wire != NULL
761       };
```

Abbildung 64: Deklaration von SigBit in rtlil.h

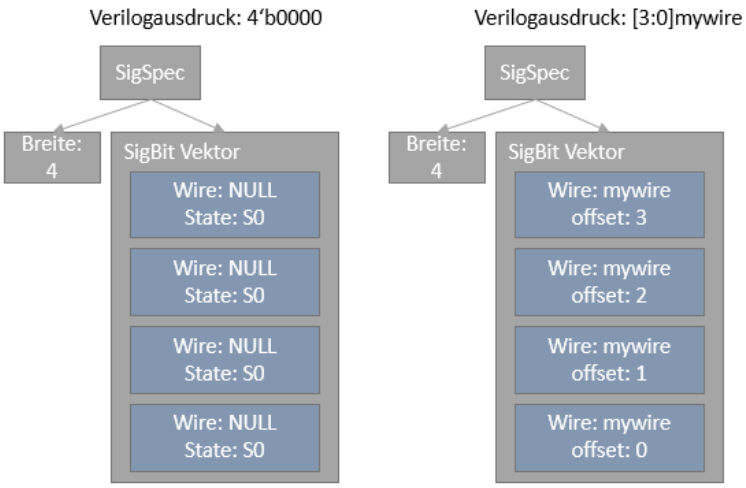

Abbildung 65: Beispiel für SigBit

RTLIL::State

Für die Darstellung der Werte von Bits existiert das Enum State. Abhängig von vordefinierten Konstanten wird der Wert wie in der folgenden Tabelle gelistet festgelegt.

Wert von State	*Bedeutung*
S0	Logische 0
S1	Logische 1
Sx	Undefinierter Zustand
Sz	Hochohmig
Sa	Don’t care
Sm	„marker“, interner Wert

Tabelle 6: Werte vom Typ RTLIL::State

RTLIL::SigChunk

Anders als SigBit beschreibt SigChunk mehr als nur ein Signalbit. Deshalb besitzt es eine eigene Integervariable für die Breite und statt einem State Objekt, wird ein Vektor verwendet. Mit der Integervariable „offset“ wird die Position des LSB (Least Significant Bit) angegeben. Abbildung 66 zeigt anhand des Ausdrucks {2’b00, 2’b11}, wie das dazugehörige SigSpec Objekt aussieht. In Abbildung 67 wird die Deklaration von SigChunk angegeben.

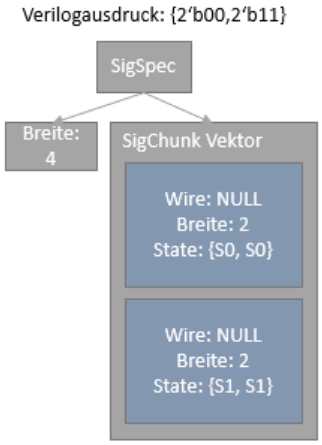

Abbildung 66: Beispiel eines SigSpec Objekts mit SigChunk Elementen

```
729   struct RTLIL::SigChunk
730   {
731       RTLIL::Wire *wire;
732       std::vector<RTLIL::State> data; // only used if wire == NULL, LSB at index 0
733       int width, offset;
```

Abbildung 67: Deklaration von SigChunk in rtlil.h

RTLIL::SigSig

Eine Verbindung von zwei Signalen entspricht in RTLIL einen SigSig Objekt. Das SigSig Objekt ist ein Paar aus zwei SigSpec Objekten. Diese tauchen als „connections“ in Modulen und „actions“ in Prozessen auf.

RTLIL::Wire

Ports und Verbindung zu Komponenten werden mit einem Wire Objekt umgesetzt. In den Signalen sind sie als Zeiger in SigBit und SigChunk enthalten. Ein Wire Objekt besitzt neben einem Namen einen Zeiger für das zugehörige Modul und Attribute in Form von Integer- und booleschen Variablen. Im Folgenden werden die Eigenschaften erläutert, die auch in Abbildung 68 in der Deklaration von Wire entnommen werden können:

- width: Breite des Busses
- start_offset: Position von LSB
- port_id: Eindeutiger Identifier für den Port
- port_input: Flag für Inputport. Port ist inout, wenn beide Flags für Input und Output gesetzt sind.
- port_output: Flag für Inputport. Port ist inout, wenn beide Flags für Input und Output gesetzt sind.
- upto: Flag für Bitreihenfolge
- is_signed: Flag, ob der Bus vorzeichenbehaftet, behandelt werden soll

```
RTLIL::Module *module;
RTLIL::IdString name;
int width, start_offset, port_id;
bool port_input, port_output, upto, is_signed;
```

Abbildung 68: Attribute von Wire in rtlil.h

4.3 Beschreibung eines Designs in RTLIL

Ein Schaltungsentwurf wird in RTILIL als hierarchische Struktur dargestellt. An oberster Stelle steht das Design Objekt in RTLIL. In diesem Objekt, wendet Yosys Passes für die Synthese an [1, p. 31]. Ähnlich wie in Verilog, wo ein Verilogdesign in Module strukturiert ist, stellt das Design Objekt ein Container für Module Objekte dar. Ein Objekt vom Typ Module besteht aus zwei Gruppen von RTLIL Objekten. Zum einen enthält Module Cell und Wire Objekte, die Elemente einer Netzliste darstellen, bzw. bei einem Technologie Mapping in reale Gatter umgewandelt werden. Die andere Gruppe enthält Process und Memory Objekte, die in Passes von Yosys in Cell und Wire Objekte umgewandelt werden. Laut Wolf besteht das Design nach der Synthese aus Objekten vom Typ Wire und Cell [1, p. 32]. Abbildung 69 zeigt vereinfacht den Aufbau von RTLIL.

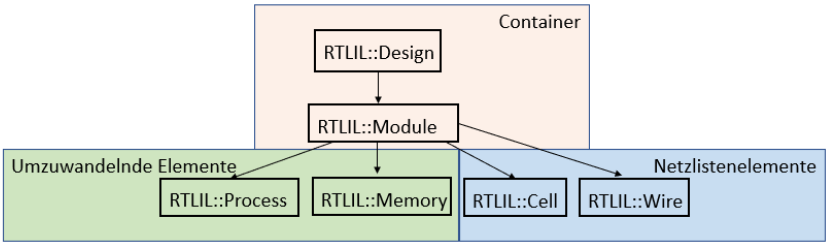

Abbildung 69: Vereinfachte Darstellung RTLIL

RTLIL::Cell

Logische Operationen, Register, Multiplexer etc. werden in RTLIL als Zellen in Form von Cell Objekten abgebildet. Diese Objekte besitzen Eingänge, Ausgänge und Parameter, die das Verhalten beschreiben und werden beim Einlesen und während der Synthese erzeugt. In Abbildung 70 wird ein kleiner Teil der Zelltypen mit den dazugehörigen Verilog-Operation aufgelistet.

Verilog	Cell Type
`Y = A & B`	`$and`
`Y = A | B`	`$or`
`Y = A ^ B`	`$xor`
`Y = A ~^ B`	`$xnor`
`Y = A << B`	`$shl`
`Y = A >> B`	`$shr`
`Y = A <<< B`	`$sshl`
`Y = A >>> B`	`$sshr`
`Y = A && B`	`$logic_and`
`Y = A || B`	`$logic_or`
`Y = A === B`	`$eqx`
`Y = A !== B`	`$nex`

Abbildung 70: Auszug aus Yosys Manual für RTL Cell [1, p. 42]

4.3.1 Beschreibung von always-Blöcken in RTLIL

Prozedurale Elemente in Verilog werden mit always-Blöcken realisiert. Das heißt, dass Anweisungen innerhalb solcher Blöcke sequenziell bzw. kombinatorisch interpretiert werden [6]. Signale in einer sogenannte Sensitivityliste lösen die Ausführung des Blocks aus, wenn es zu einer Änderung der Signale kommt. In RTLIL stellen RTLIL::Process Objekte always-Blöcke dar.

RTLIL::Process

Neben eines RTLIL::IdString Objektes für die Bezeichnung des Prozesses enthält ein Objekt vom Typ RTLIL::Process einen Zeiger vom Typ RTLIL::Module für die Zugehörigkeit in ein Modul, ein RTLIL::CaseRule Objekt, das die Anweisungen des always-Blocks enthält und ein Vektor für Zeiger vom Type RTLIL::SyncRule.

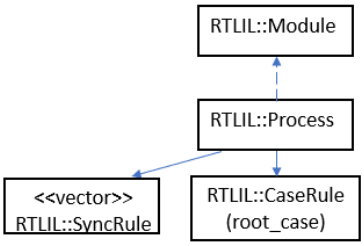

Abbildung 71: Darstellung welche RTLIL Objekte einem Process bekannt sind

RTLIL::SyncRule

Ein SyncRule Objekt besitzt ein Signal in Form eines SigSpec Objektes, ein RTLIL::SyncType Objekt und Aktionen. Aktionen sind Zuweisungen oder Speicherschreibzugriffe. Zuweisungen sind in einem SigSig Vektor enthalten und Speicherschreibzugriffe in einem Vektor des Typen RTLIL::MemWrite Actions.

RTLIL::SyncType

Wie der Typ RTLIL::State ist SyncType ein Enum der den Synchronisierungstyp anhand einer Konstante festhält. Er gibt an, bei welcher Änderung des Signals der Prozess ausgeführt werden soll. Folgende Typen werden in RTLIL unterscheiden:

Wert von SyncType	*Bedeutung*
ST0	Low-active
ST1	High-active
STp	Positive Flanke
STn	Negative Flanke
Ste	Beide Flanken
STa	Immer aktiv
STg	Globale Clock
STi	init

Tabelle 7: SyncType Typen

RTLIL::CaseRule, RTLIL::SwitchRule

Ein RTLIL::CaseRule Objekt enthält die Zuweisungen eines always-Blocks und überführt Verzweigungen in eine Struktur aus CaseRule und SwitchRule Objekten. Das CaseRule Objekt, das in RTLIL::Process enthalten ist, stellt in der Hierarchie das höchste Element unter CaseRule und SwitchRule Objekten dar. Jedes CaseRule Objekt besitzt die drei Vektoren „compare“, „actions“ und „switches“. Enthält die Beschreibung eines always-Blocks keine Verzweigungen und nur Zuweisungen, dann enthält nur der Vektor „actions“ Elemente, die vom Typ SigSig sind. Die Vektoren „compare“ und „switches“ enthalten dann auch keine Elemente. Verzweigungen werden durch den Vektor „switches“ umgesetzt, der dann Elemente des Typs RTLIL::SwitchRule speichert.

Eine Verzweigung erfordert die Auswertung einer Bedingung und führt Anweisungen in einem Block aus, wenn die Bedingung erfüllt ist. Für die Auswertung wird in RTLIL ein SigSpec Objekt definiert, dass ein Signal enthält. Für die Zweige, also Anweisungsblöcke, besitzt ein SwitchRule Objekt einen Stapel aus CaseRule Objekten. Diese CaseRule Objekte innerhalb der SwitchRule besitzen in „compare“ den Ausdruck für die Bedingung des SigSpec Objekt und die Anweisungen in „actions“. Dieser Aufbau ermöglicht die Realisierung von verschachtelten Verzweigungen durch die Verschachtelung von CaseRule und SwitchRule Objekten. Abbildung 72 zeigt die Struktur der RTLIL Objekte in einem Prozess mit Verzweigungen.

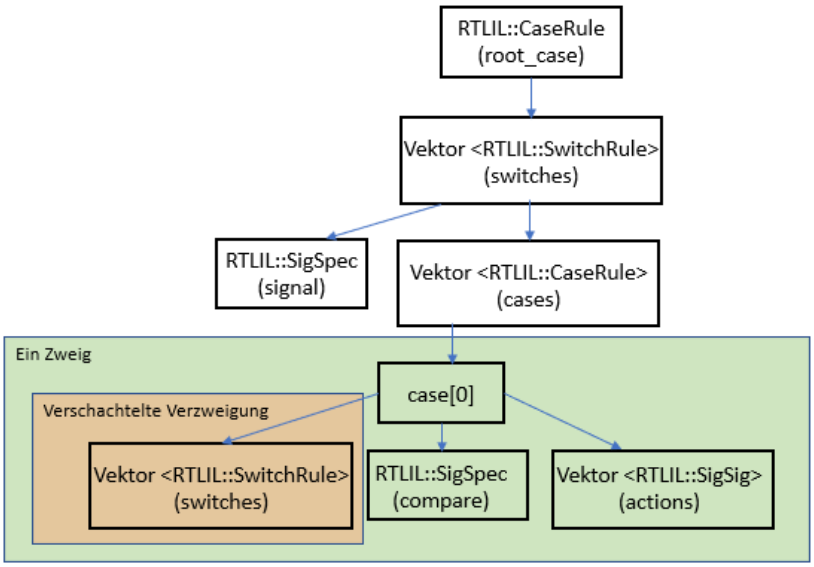

Abbildung 72: Darstellung Verzweigung in RTLIL

4.3.2 Beschreibung von Verilog Arrays

Wie in Kapitel 3 aufbereitet und im vorherigen Absatz erläutert, werden Lesezugriffe auf Arrays in Verilog in Cell Objekte vom Typ „$memrd“ und Schreibzugriffe in Objekte vom Typ MemWriteAction in den SyncRule Objekten überführt. Die Arrays selbst werden in RTLIL zu Memory Objekten umgewandelt. Laut Yosys Handbuch besitzt ein Memory Objekt neben einer Bezeichnung, Variablen für die Breite eines Wortes, die Anzahl der Wörter und Attribute, die das Array beschreiben [1, p. 37]. Bis auf eine zusätzliche Variable für ein Offset, stimmt die Beschreibung mit der Deklaration von Memory in Abbildung 73 überein.

```
1486  struct RTLIL::Memory : public RTLIL::AttrObject
1487  {
1488      unsigned int hashidx_;
1489      unsigned int hash() const { return hashidx_; }
1490
1491      Memory();
1492
1493      RTLIL::IdString name;
1494      int width, start offset, size;
```

Abbildung 73: Deklaration von Memory in rtlil.h

5 Zusammenfassung

Das Ziel dieser Arbeit war, durch eine Analyse des Yosys Source Codes die Konvertierung von Verilog zu RTLIL in Yosys näher zu beleuchten. Dabei wurde das Verilog und AST Frontend untersucht. Als Ergebnisse der Untersuchung des Verilog Frontends kann festgehalten werden, dass Yosys Verilog einliest, indem es durch einen Bison-Parser, bei Anwendung einer vordefinierten Grammatik, AstNode Objekte erstellt. Dabei wird der Aufbau des ASTs allein durch das Hinzufügen von Knoten als Children realisiert, die in den Parser Regeln erzeugt werden. Während dieses Vorgangs wird ein Stapel als Hilfsobjekt verwendet.

Die Untersuchung des AST Frontends und RTLIL hat gezeigt, dass die Generierung von RTLIL abhängig vom Typ der Knoten im AST geschieht, auf dem der RTLIL Generator angewendet wird. Im einfachsten Fall wird für ein AstNode Objekt, das äquivalente RTLIL Objekt erzeugt und Daten in Variablen übernommen. Für verhaltensbeschreibende Knoten wie AST_ALWAYS kommt zusätzlich ein Prozessgenerator zum Einsatz. Weiter hat sich gezeigt, dass Knoten, die Operation darstellen in RTLIL::Cell Objekte überführt werden, die in einem zusätzlichen Technologie Mapping Schritt zu realen Logikgattern überführt werden können. Dies lässt die Deutung zu, dass bereits bei der Generierung von RTLIL erste Schritte zur Synthese durchgeführt werden und die AstNode Struktur ein wesentlicher Bestandteil für die korrekte Umwandlung in RTLIL darstellt.

Zusammenfassend kann gesagt werden, dass die Frage, wie das Einlesen einer Verilogdatei in Yosys funktioniert und wie dabei RTLIL generiert wird, mit den gewonnenen Erkenntnissen als beantwortet betrachtet werden kann. Jedoch wurden nicht alle Unterprozesse behandelt und zu einzelnen Elementen kann keine Aussage getroffen werden. Untersuchungen haben aufgezeigt, dass eine komplette Analyse des Verilog und AST Frontends im vollen Umfang den zeitlichen Rahmen dieser Arbeit sprengen würde und deswegen nicht umzusetzen ist.

Zukünftige Arbeiten könnten, die nicht berücksichtigten Prozesse untersuchen oder diese Arbeit weiterführen, indem die Ergebnisse genutzt werden, um die Umwandlung aller Verilog-Sprachelemente zu verfolgen. Ein anderer Aspekt, der in Folge dieser Arbeit betrachtet werden könnte, ist die Entwicklung eines eigenes Yosys-Frontends in Yosys, der die AstNode Struktur nutzt.

6 Literaturverzeichnis

[1] C. X. Wolf, „Yosys Open SYnthesis Suite,“ o. D.. [Online]. Available: https://github.com/YosysHQ/yosys-manual-build/releases/download/manual/manual.pdf. [Zugriff am 25 05 2022].

[2] M. S. L. R. S. J. D. U. Alfred V. Aho, Compilers: Principles, Techniques, & Tools 2nd Edition, Boston: Addison-Wesley, 2006.

[3] J. R. Levine, flex & bison, Sebastopol: O'Reilly Media, Inc., 2009.

[4] M. H. Christian Wagenknecht, Formale Sprachen, abstrakte Automaten und Compiler, Wiesbaden: Springer Vieweg, 2022.

[5] H. Works, „HDL Works,“ 2022-2022. [Online]. Available: https://www.hdlworks.com/hdl_corner/verilog_ref/items/CompilerDirectives.htm.

[6] ChipVerify, „chipverify.com,“ ChipVerify, 2015-2022. [Online]. Available: https://www.chipverify.com/verilog/verilog-always-block.

7 Abbildungsverzeichnis

8 Tabellenverzeichnis

9 Anhang Tutorial für Lexer und Parser

In diesem Tutorial soll mit Flex und Bison, ein Lexer und Parser realisiert werden, der die Verilogdatei in Abbildung 74 parsen kann. Mit Hilfe des Lexers wird die Verilogdatei in einzelne Elemente den sogenannten Tokens zerlegt. Der Parser prüft die Syntax, indem eine Token- Sequenz mit Grammatikregeln verglichen wird. Die Verilogdatei, die in diesem Tutorial eingelesen wird, besteht aus einem Modul, zwei Wiren und einer Zuweisung.

```
module beispiel
wire a;
wire b;
assign a = b;
endmodule
```

Abbildung 74: Verilog Beispiel example.v, Lexer und Parser

9.1 Installation von Flex, Bison und C-Compiler

In diesem Abschnitt wird erläutert, wie Flex, Bison und der C-Compiler GCC auf einer Linux-Umgebung installiert wird. Für die Installation von GCC mit allen benötigten Packages geben wir folgenden Befehl im Terminal an:

```
sudo apt install build-essential
```

Nach dem Installieren können C-Dateien kompiliert werden. Dies geschieht mit folgendem Befehl:

```
gcc <file.c> -o <file>
```

Das Ausführen erfolgt mit:

```
./<file>
```

Das Setup für Flex und Bison erfolgt über apt mit:

```
sudo apt-get install flex bison
```

9.2 Der Lexer

Im ersten Schritt wird der Lexer implementiert, indem ein Flex-Programm lexer.l geschrieben wird. Im Hauptprogramm soll der Inputstream „yyin“ vom Lexer generiert werden und der Lexer soll nur eine Datei lesen. Die Angabe ‚%option header-file=“lex.yy.h“‘ teil Flex mit, dass für den Lexer eine Headerdatei erzeugt werden soll. Sie wird im Hauptprogramm eingebunden. Mit „%option noyywrap“ wird dem Lexer mitgeteilt, dass nur eine Datei eingelesen wird. Die Definition der Tokens befindet sich im Header des Parsers. Deshalb muss der Lexer die Datei „parser.tab.h“ einbinden. In diesem Tutorial wird jeder erkannte reguläre Ausdruck ausgegeben. Außerdem wird die Bezeichnung der Wire und des Moduls gespeichert. Diese Funktionalität erfordert die Einbindung der Bibliotheken „string.h“ und „stdio.h“. Als Zusatz wird eine benutzerdefinierte Funktion unknownExpression() deklariert, die eine Meldung ausgibt, wenn nach allen definierten Regel noch Zeichen existieren, die nicht ausgewertet wurden. C-Code Deklaration werden in einem Block mit „%{“ und „%}“ angegeben.

Anschließend wird eine Definition „reg_string“ angegeben, die als Makro für den regulären Ausdruck [a-zA-Z]+ gilt. Der reguläre Ausdruck [a-zA-z]+ wertet alle Ausdrücke, die Groß-, Kleinbuchstaben bestehen und mindestens aus ein Zeichen besitzen als gültig. Damit ist die Sektion Definition fertig. Sektionen werden mit %% getrennt.

Der Definitionsteil des Flex-Programms sieht dann wie folgt aus:

```
%option header-file="lex.yy.h"
%option noyywrap
%{
  #include "parser.tab.h"
  #include <string.h>
  #include <stdio.h>
  void unknownExpression();
%}
reg_string [a-zA-Z]+
%%
```

Die nächste Sektion des Flex-Programms beinhaltet Regeln, die mit<regulärer Ausdruck, Makro> {<Aktionen in C-Code>} aufgelistet werden. Für die Verilogdatei werden die Schlüsselwörter bzw. Zeichen „module“, „wire“, „assign“, „endmodule“, ‚;‘ und ‚=‘ ausgewertet. Für diese Ausdrücke wird als Aktion der betreffende Ausdruck ausgegeben und ein Token zurückgegeben. Die Token heißen TMODULE, TENDMODULE, TASSIGN, TWIRE, TEQUAL, TSEMIKOLON und TNAME. Für das Makro „reg_string“ soll zusätzlich, der Ausdruck selbst als Wert in der Variable „yylval->string_name“ des Parsers gespeichert werden. Die Variable „yytext“ enthält den vom Lexer gelesenen Ausdruck. Mit Hilfe der Funktion sscanf(yytext, %s, yylval->string_name) wird „yytext“ in „yylval->string_name“ als String überführt.

Zeilenende und Leerzeichen geben kein Token zurück. Für alle weiteren nicht behandelten Ausdrücke soll die Funktion unknownExpression() ausgeführt werden, die den Text „unknown Expression found“ ausgibt. Die Funktion wird in der Routine-Sektion implementiert, die mit printf() Meldungen ausgibt.

Das komplette Flex-Programm, welches die oben beschriebene Funktionalität umsetzt, hat das folgende Aussehen::

```
%option header-file="lex.yy.h"
%option noyywrap
%{
  #include "parser.tab.h"
  #include <string.h>
  #include <stdio.h>
  void unknownExpression();
%}
reg_string [a-zA-Z]+
%%
"module" {printf("module\n"); return TMODULE;}
"endmodule" {printf("endmodule\n"); return TENDMODULE;}
"assign" {printf("assign\n"); return TASSIGN;}
"wire" {printf("wire\n"); return TWIRE;}
"=" {printf("equal\n"); return TEQUAL;}
";" {printf("semikolon\n"); return TSEMIKOLON;}
{reg_string} { sscanf(yytext, "%s", yylval.string_name); return TNAME; }
[\n|\t] {;}
" " {printf("space\n");}
. {unknownExpression();}
%%
void unknownExpression() {
  printf("Unknown expression found\n");
}
```

Abbildung 75: Flex-Programm lexer.l

9.3 Der Parser

Der nächste Schritt ist die Implementierung des Parsers durch das Schreiben eines Bison-Programms parser.y. Der Parser soll während und am Ende des Parse-Vorgangs Ausgaben generieren. Deshalb muss wieder die Bibliothek „stdio.h“ eingebunden werden. Kommt es zum Syntaxfehler, wird eine Meldung über die Funktion yyerror(char *error) ausgegeben. Die Funktionsdeklarationen und die Einbindung erfolgen zwischen „%{“ und „%}“.

Mögliche Token, die der Lexer zurückgeben kann, werden mit „%token <Tokenname>“ definiert. Zusätzlich soll der Wert des Tokens TNAME in ein char-Array gespeichert werden. Dafür wird ein char Array „string_name“ in „%union“ definiert. In der Tokendefinition wird der Token TNAME mit <string_name> ergänzt, um den Parser mitzuteilen, dass der Wert des Tokens ein char-Array ist. Das Bison-Programm sieht danach wie folgt aus mit %% für Trennung der Sektion:

```
%{
  #include <stdio.h>
  extern int yylex();
  extern int yyerror(char *error);
%}
%union {
 char string_name[30];
}
%token TMODULE
%token TENDMODULE
%token TASSIGN
%token TWIRE
%token TEQUAL
%token TSEMIKOLON
%token TNAME
%token <string_name> TNAME
%%
```

Nach den Definitionen werden Grammatikregeln in Form von Produktionsregeln mit <Nichtterminalsymbol> : <Ausdruck aus Symbolen> {<Aktion in C-Code>}; aufgelistet. Von unten nach oben sollen folgende Regeln aufgestellt werden.

Das Token TNAME wird mit dem Symbol „name“ ersetzt. Das Symbol „name“ wiederum wird mit „basic_expr“ ersetzt. Bei der Ersetzung durch „name“ soll der Wert ausgegeben werden. MIt $1 wird auf den Wert von TNAME zugegriffen. Für weitere Symbole wird $2, $3 usw. genutzt. Mit $$ wird auf das Symbol auf der linken Seite zugegriffen.

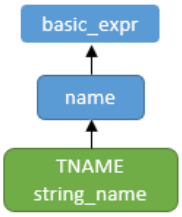

Abbildung 76: Erstzen von TNAME in name und name in basic_expr

```
basic_expr : name;
name : TNAME {printf("name: %s\n", $1);};
```

Die Definition eines Wires wird in Verilog mit „wire <name>;“ angegeben. Die entsprechende Parser-Regel bildet das Symbol „wire_stmt“ mit der Sequenz „TWIRE basic_expr TSEMIKOLON“. Eine Zuweisung bildet das Symbol „assign_stmt“ mit der Sequenz „ASSIGN basic_exp TEQUAL basic_expr TSEMIKOLON“.

```
wire_stmt : TWIRE basic_expr TSEMIKOLON{printf("wire stmt\n");};
assign_stmt : TASSIGN basic_expr TEQUAL basic_expr TSEMIKOLON{printf("assign
stmt\n");};
```

Im nächsten Schritt wird „assign_smt“ und „wire_stmt“ durch „stmt“ ersetzt. Das Symbol stmt_list ersetzt eine Liste aus stmt Symbolen. Das Zeichen ‚|‘ wird in der Parser-Regel als oder Operation interpretiert.

```
stmt_list : stmt | stmt_list stmt;
stmt : wire_stmt | assign_stmt;
```

Als vorletzte Regel wird „module“ definiert, dass die Sequenz „TMODULE basic_expr stmt_list TENDMODULE“ ersetzt. Danach wird mit „input“ die Eingabe akzeptiert und „Input accepted“ ausgegeben.

Das komplette Bison-Programm hat folgendes Aussehen:

```
%{
  #include <stdio.h>
  extern int yylex();
  extern int yyerror(char *error);
%}
%union {
 char string_name[30];
}
%token TMODULE
%token TENDMODULE
%token TASSIGN
%token TWIRE
%token TEQUAL
%token TSEMIKOLON
%token TNAME
%token <string_name> TNAME
%%
input:  module {printf("input accepted\n");};
module:  TMODULE basic_expr stmt_list TENDMODULE;
stmt_list : stmt | stmt_list stmt;
stmt : wire_stmt | assign_stmt;
wire_stmt : TWIRE basic_expr TSEMIKOLON{printf("wire stmt\n");};
assign_stmt : TASSIGN basic_expr TEQUAL basic_expr TSEMIKOLON{printf("assign
stmt\n");};
basic_expr : name;
name : TNAME {printf("name: %s\n", $1);};
```

Abbildung 77: Bison-Programm parser.y

9.4 Das Hauptprogramm

Zuletzt muss das Hauptprogramm main.c geschrieben werden. Das Hauptprogramm öffnet die Verilogdatei und überträgt die Datei als Eingabe in den Inputstream „yyin“ des Lexers. Mit yyparse() startet das Programm das Lexen und Parsen. Am Ende wird der Inputstream „yyin“ mit fclose() geschlossen. Weiter implementiert das Hauptprogramm die Funktion yyerror(). Sie gibt Meldungen aus, wenn es zum einen Syntaxfehler kommt. Damit das Hauptprogramm „yyin“ und yyparse() kennt, muss die Headerdatei lex.yy.h eingebunden werden und yyparse() als externe Funktion deklariert werden. Das komplette Programm ist in Abbildung 78 dargestellt.

```c
#include <stdio.h>
#include "lex.yy.h"
extern int yyparse(void);

void yyerror(char const *error){
  printf("Fehler! Grund ist: %s\n", error);
}

int main(int argc, char const *argv[]) {
  yyin = fopen(argv[1], "r"); //Öffnet Datei zum Lesen und setzt globalen In-
putstream yyin
  yyparse(); //Startet das Lexen und Parsen
  fclose(yyin); //Schließt Inputstream yyin und eingelesene Datei
  return 0;
}
```

Abbildung 78: Hauptprogramm das Lexer und Parser ausführt main.c

Die Generierung des Lexers und Parsers wird durch die Kommandos „flex lexer.l“ und „bison -d parser.y“ ausgeführt. Der Parameter „-d teilt Bison mit, dass für den Parser eine Header-datei erzeugt werden soll. Der erstellte Lexer ist in lex.yy.c umgesetzt und der Parser in parser.tab.c. Nun müssen die C-Dateien wie gewohnt kompiliert werden. Zur Vereinfachung wurde ein Makefile erstellt. Der Befehl „make flex“ erstellt den Lexer, „make parser“ erstellt den Parser und der Befehl „make main“ kompiliert das Hauptprogramm zusammen mit dem genierten Source Codes des Lexers und des Parsers. Das Programm heißt main und wird mit dem Kommandozeilenbefehl „./main <Dateiname>“ ausgeführt , wobei <Dateiname> dem Namen einer Verilog-Datei entspricht, die verarbeitet werden soll.

```makefile
.PHONY: clean
objects = parser.tab.o main.o lex.yy.o lex.yy.c lex.yy.h parser.tab.c par-
ser.tab.h
%.o: %.c %.h
    gcc -g -c -o $@ $<

main: parser.tab.o lex.yy.o main.o
    gcc -o $@ parser.tab.o lex.yy.o main.o

lex: lexer.l
    flex lexer.l

parser: parser.y
    bison -d parser.y

clean:
    -rm main $(objects)
```

Abbildung 79: Inhalt von Makefile

Die Ausführung des Programms in Kombination mit der Verilogdatei führt zu den Ausgaben in Abbildung 80 dargestellten Ausgaben. Man erkennt, dass die Datei akzeptiert wurde und damit syntaktisch korrekt ist.

```
user@syntheseprojekt:~/Desktop/flexbison$ ./main example.v
module
space
name: mname
wire
space
name: a
semikolon
wire stmt
wire
space
name: b
semikolon
wire stmt
assign
space
name: a
space
equal
space
name: b
semikolon
assign stmt
endmodule
input accepted
```

Abbildung 80: Ausgabe nach dem Parsen

9.5 Erweiterung mit zusätzlicher Aufgabe

Lexer und Parser sollen so erweitert werden, dass folgende Verilogdatei eingelesen werden kann. Es soll eine Zuweisung mit dem UND-Operator unterstützt werden.

```
module beispiel
wire a;
wire b;
wire c;
assign c = a & b;
endmodule
```

Lösung

Im Lexer muss das Zeichen mit einer Regel ausgewertet werden.

```
%option header-file="lex.yy.h"
%option noyywrap
%{
  #include "parser.tab.h"
  #include <stdio.h>
  void unknownExpression();
%}
reg_string [a-zA-Z]+
%%
"module" {printf("module\n"); return TMODULE;}
"endmodule" {printf("endmodule\n"); return TENDMODULE;}
"assign" {printf("assign\n"); return TASSIGN;}
"wire" {printf("wire\n"); return TWIRE;}
"=" {printf("equal\n"); return TEQUAL;}
";" {printf("semikolon\n"); return TSEMIKOLON;}
{reg_string} { sscanf(yytext, "%s", yylval.string_name); return TNAME; }
[\n|\t] {;}
" " {printf("space\n");}
"&" {printf("and\n"); return TAND;}
. {unknownExpression();}
%%
void unknownExpression() {
  printf("Unknown expression found\n");
}
```

Im Parser muss ein Token für den UND-Operator definiert und die Regel für assign_stmt erweitert werden.

```
%{
  #include <stdio.h>
  extern int yylex();
  extern int yyerror(char *error);
%}
%union {
 char string_name[30];
}
%token TMODULE
%token TENDMODULE
%token TASSIGN
%token TWIRE
%token TEQUAL
%token TSEMIKOLON
%token <string_name> TNAME
%token TAND
%%
input:  module {printf("input accepted\n");};
module:  TMODULE basic_expr stmt_list TENDMODULE;
stmt_list : stmt | stmt_list stmt;
stmt : wire_stmt | assign_stmt;
wire_stmt : TWIRE basic_expr TSEMIKOLON{printf("wire stmt\n");};
assign_stmt : TASSIGN basic_expr TEQUAL basic_expr TSEMIKOLON{printf("assign
stmt\n");}
        | TASSIGN basic_expr TEQUAL basic_expr TAND basic_expr
TSEMIKOLON{printf("assign stmt\n");};
basic_expr : name;
name : TNAME {printf("name: %s\n", $1);};
```

10 Anhang Quellcode

Der Quellcode von Yosys ist verfügbar unter https://github.com/YosysHQ/yosys. Dabei wurde die Version 0.15 von Yosys genutzt, die unter https://github.com/YosysHQ/yosys/tree/07a43689d8104b973c610e84fd4f98564bc8e859 zu finden ist.

Folgende Dateien von Yosys wurden untersucht:

- /frontends/verilog/verilog_frontend.h
- /frontends/verilog/verilog_frontend.cc
- /frontends/verilog/preproc.h
- /frontends/verilog/preproc.cc
- /frontends/verilog/verilog_lexer.l
- /frontends/verilog/verilog_lexer.cc
- /frontends/verilog/verilog_parser.y
- /frontends/verilog/verilog_parser.tab.h
- /frontends/verilog/verilog_parser.tab.cc
- /frontends/ast/ast.h
- /frontends/ast/ast.cc
- /frontends/ast/simplify.cc
- /frontends/ast/genrtlil.cc
- /kernel/rtlil.h
- /kernel/rtlil.cc

11 Anhang Verilogcode

Sämtliche in der Arbeit genutzte Verilog Beispiele sind hier zu finden. Das Einlesen in Yosys geschieht mit dem Ausführen des Befehls „read_verilog <Pfad zur Verilogdatei>“.

```
module dffp (D, CLK, Q);
input D;
input CLK;
output Q;
always @ (posedge CLK)
begin
   Q <= D;
end
endmodule
```

Abbildung 81: Verilog Beispiel für ein D-FlipFlop

```
module test(input A, B, output reg C);
assign C = A + B;
endmodule
```

Abbildung 82: Verilog Beispiel für eine Zuweisung mit Addition

```
module test(input A, output B);
assign A = B;
endmodule
```

Abbildung 83: Verilog Beispiel für eine Zuweisung

```
module range;
reg [3:0]A;
assign A = 4'b0000;
endmodule
```

Abbildung 84: Verilog Beispiel für AstNode Objekt mit Typ AST_RANGE

```
module condition (input clk,
output a);
always @ (clk)
begin
if (clk == 1'b1)
   a = 1'b1;
else
   a = 1'b0;
end
endmodule
```

Abbildung 85: Verilog Beispiel für Verzweigung