Bachelorarbeit

Untersuchung der Yosys Hardwaresynthese von der internen Datenstruktur RTLIL bis zur Netzliste

Verfasst von:	**Patryk Szymon Janik**
Matrikelnummer:	**1 1 1 2 8 3 3 2**
Studiengang:	**Bachelor Technische Informatik**
Erstgutachter:	**Prof. Dr. rer. nat. Tobias Krawutschke**
Zweitgutachter:	**Prof. Dr.-Ing. Michael Karagounis**

Abgabefrist: 0 2 . 1 1 . 2 2

Ich erkläre an Eides statt, dass ich die vorgelegte Abschlussarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum, Unterschrift: _____

Fakultät für Informations-, Medien- und Elektrotechnik

Technology Arts Sciences
TH Köln

Bachelorarbeit

Titel: Untersuchung der Yosys Hardwaresynthese von der internen Datenstruktur RTLIL bis zur Netzliste

Gutachter:

- Prof. Dr. rer. nat. Tobias Krawutschke (TH Köln)
- Prof. Dr.-Ing. Michael Karagounis (FH Dortmund)

Zusammenfassung: Das Ziel dieser Arbeit ist die Beantwortung der Frage "Wie funktioniert Synthese?". Yosys ist ein offenes Synthesewerkzeug, welches untersucht wurde, um diese Frage zu beantworten. Yosys implementiert eine Datenstruktur RTLIL, mit der ein Entwurf in allen Synthesephasen dargestellt wird. Yosys ist modular aufgebaut, was dem Nutzer ermöglicht, das Programm zu erweitern. Die Synthese in Yosys ist auf Pässe unterteilt, die jeweils eine bestimmte Aufgabe erfüllen. Im Rahmen der Arbeit wurde die Datenstruktur und die Passes im einzelnen analysiert. Es wurde auch untersucht, wie in Yosys Erweiterungen zu implementieren sind. Die Analyse hat gezeigt, dass ein wichtiger Teil der Synthese die Umwandlung von Prozessen in eine RTL-Beschreibung darstellt. Im Rahmen der Synthese werden die, von einem Frontend vorläufig erzeugten RTL-Komponenten, umgewandelt. Der letzte Schritt der Synthese ist das Technologiemapping, welches die umgewandelten Komponente auf die verwendete Hardware anpasst.

Stichwörter: Yosys, RTLIL, Synthese, Hardwareentwicklung

Datum: 02.11.2022

Bachelor Thesis

Title: Analysis of the Yosys hardware synthesis from the internal data structure RTLIL to the netlist

Reviewers:

- Prof. Dr. rer. nat. Tobias Krawutschke (TH Köln)
- Prof. Dr.-Ing. Michael Karagounis (FH Dortmund)

Abstract: The aim of this thesis is an answer to the question "How does synthesis work?". Yosys is an open synthesis tool that was analyzed to answer this question. Yosys implements a data structure RTLIL that is used to represent a design at all stages of synthesis. Yosys has a modular structure, which allows the user to expand the program. Synthesis in Yosys is divided into passes, each of which performs a specific task. As part of the thesis, the data structure and the passes were analyzed in detail. It was also examined how to implement extensions in Yosys. The analysis showed that an important part of the synthesis is the conversion of processes into RTL components. During the synthesis, the provisional RTL components generated by a frontend are converted. The last step of synthesis is technology mapping, which adapts the converted components to the used hardware.

Keywords: Yosys, RTLIL, synthesis, hardware development

Date: 02.11.2022

Danksagungen

An dieser Stelle möchte ich mich bei dem Prof. Dr. rer. nat. Tobias Krawutschke und dem Prof. Dr.-Ing. Michael Karagounis, für die Betreuung und Unterstützung bei dem Verfassen dieser Arbeit.

Viel Dank geht auch an meinen Kommilitonen dem Herrn Christopher Parnow für die konstruktive Kritik, Korrekturen und seelische Unterstützung. Ohne ihn wäre die Erstellung dieser Arbeit nicht möglich gewesen.

Ich möchte meinen Eltern für die finanzielle Unterstützung danken, die es mir ermöglicht hat, mich ganz auf die Verfassung dieser Arbeit zu konzentrieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichniss

Abkürzungsverzeichnis

1 Einleitung

In der Hardwareentwicklung werden oft Hardwarebeschreibungssprachen (engl. Hardware Description Languages, HDLs) für die Darstellung von Hardwareentwürfen verwendet. Damit ist es möglich die Entwürfe zu analysieren, zu simulieren, und zu synthetisieren. Zu den bekanntesten HDLs gehören Verilog und VHDL[1]. Den Vorgang, mit dem eine HDL Darstellung in eine Netzliste von in der Zielhardware verfügbaren Komponenten erstellt wird, nennt man Synthese. Eine Netzliste ist eine andere Form der Darstellung eines Hardwareentwurfs. Anders als bei Verwendung einer HDL ist der Sinn einer Netzliste nicht mehr die für Menschen lesbare Darstellung, sondern die spätere automatisierte Verarbeitung der Syntheseergebnisse.

1.1 Motivation

Yosys Open SYnthesis Suite (Yosys) ist eine Free and open-source software (FOSS) deren Hauptfunktion die Durchführung einer Verilog-Synthese ist. Yosys wurde mit einem modularen Ansatz entwickelt. Das heißt, dass Yosys zwar im aktuellen Zustand bereits für die vollständige Synthese und Technologieabbildung auf bestimmte Field Programmable Gate Arrays (FPGAs) genutzt werden kann, aber eigentlich als Grundlage für spezialisierte Synthesewerkzeuge fungieren soll[14, S. 12].

An der Fachhochschule Dortmund wird ein Forschungsprojekt durchgeführt, dass zum Ziel hat, das Verständnis der Funktionsweise von Synthesewerkzeugen, zu erarbeiten. Da Yosys quelloffen entwickelt ist, eröffnet sich eine gute Gelegenheit auf Basis von Yosys den Synthesevorgang zu verstehen. Zu diesem Zweck wurde das von Yosys bereitgestellte Handbuch und der Quelltext betrachtet. Da der Quelltext über wenige Kommentare verfügt und das Handbuch nicht auf die Vorgänge im Detail eingeht, wurde es klar, dass eine detailreiche Analyse des Quelltexts benötigt wird. Diese Arbeit soll die Ergebnisse dieser Analyse vermitteln.

1.2 Zielsetzung und Umfang

Für die interne Repräsentation eines Entwurfs wird die speziell für Yosys entwickelte Datenstruktur RTLIL genutzt. Die Einlesung eines Entwurfs und dessen Überführung in RTLIL liegt nicht im Fokus dieser Arbeit. Das Ziel der Arbeit ist die Analyse der Prozesse, die einen in RTLIL vorliegenden Schaltungsentwurf auf Komponenten abbilden, die auf reeller Hardware implementiert werden können. Dazu gehört die Ausgabe des Entwurfs in einem geeignetem Format nach der Durchführung der Synthese. Ferner soll auch die Modularität von Yosys untersucht werden. Es soll ersichtlich werden, wie Yosys mit eigenen Plugins erweitert werden kann, um ggf. speziellen Anforderungen gerecht zu werden.

Im Rahmen dieser Arbeit soll keine genaue Codeanalyse durchgeführt werden. Vielmehr geht es um die allgemeine Beschreibung der Vorgänge, die für eine erfolgreiche Synthese erforderlich sind. Die

[1] Very-High-Speed Integrated Circuit (VHSIC) Hardware Description Language

Codeanalyse wird nur so detailliert beschrieben, wie es für die Entwickelung eigener Plugins und für das Verständnis der Vorgänge in Yosys erforderlich ist.

1.3 Aufbau der Arbeit

Zunächst werden die wichtigsten Grundlagen und die Synthese selbst im allgemeinen beschrieben, ohne auf die Implementierung von Yosys einzugehen. Darauf folgt die Analyse von Yosys, bei der die einzelnen Vorgänge der Synthese betrachtet werden und die Erweiterbarkeit untersucht wird. Zum Schluss werden die Ergebnisse der Analyse zusammengefasst.

Kapitel 1 "Einleitung" gibt dem Leser einen Überblick auf den Inhalt, die Motivation und den Aufbau der Arbeit.

Kapitel 2 "Theoretische Grundlagen" vermittelt die wichtigsten Grundlagen, die bei dem Verständnis des darauffolgenden Kapitels helfen sollen.

Kapitel 3 "Yosys" ist der Hauptteil der Arbeit, in dem die Ergebnisse der Analyse dargelegt werden.

Kapitel 4 "Zusammenfassung" schließt die Arbeit mit den Erkenntnissen und Ausblicken für weitere Untersuchungen ab.

2 Theoretische Grundlagen

Um den Vorgang einer Synthese zu verstehen, ist ein Basiswissen darüber, warum eine Synthese erforderlich ist, hilfreich. Die Synthese ist einer der ersten Schritte zur Implementierung eines Hardwareentwurfs auf einem FPGA oder einem ASIC. Deswegen werden im folgendem zunächst FPGAs und ASICs eingeführt. Anschließend, wird das Konzept einer HDL erläutert, um zu verdeutlichen, warum ein Entwurf in einer HDL verarbeitet werden muss, bevor die weiteren Entwicklungsschritte durchgeführt werden können.

Damit ein Syntheseprogramm die Synthese durchführen kann, muss der HDL-Entwurf zunächst eingelesen werden. Um den eingelesenen Entwurf zu verarbeiten, muss es im Programm in einer geeigneten Struktur vorliegen. Der Prozess des Einlesens soll kurz als Grundlage für die weitere Betrachtung einzelner Prozesse von Yosys eingeführt werden.

Ein verarbeiteter Entwurf muss für die weiteren Implementierungsschritte bereitgestellt werden. Das Ergebnis einer Synthese ist in der Regel eine Netzliste, die von weiteren Programmen verarbeitet wird. Durch die Einführung des Aufbaus und der Bestandteile einer Netzliste soll deutlich werden, welche Konzepte einer HDL nicht darstellbar sind und umgewandelt werden müssen.

Mit diesem Wissen wird letztendlich die Synthese im allgemeinen betrachtet, bevor auf die eigentliche Implementierung in Yosys eingegangen wird.

2.1 FPGAs und ASICs

Ein FPGA ist eine integrierte Schaltung (engl. Integrated Circuit, IC) dessen Funktion konfigurierbar ist. Die Schaltung im FPGA bleibt gleich, ist jedoch in der Lage die Funktion einer vorher definierten Schaltung zu übernehmen. Möglich wird dies durch den Einsatz von den sogenannten Look-Up Tables (LUTs).

Ein LUT ist ein $2\text{–}4 \times 1$ bit Speicher[1]. Solche LUTs ermöglichen die Implementierung von beliebiger kombinatorischer Logik. Wenn die einzelnen Adressbits als Eingänge und der gespeicherte Wert als Ausgang betrachtet wird, kann die Funktion eines LUTs durch die Änderung der gespeicherten Bits modifiziert werden.

Ein LUT ist in der Regel ein Bestandteil einer Logikzelle, die wiederum meistens aus LUTs, einem Volladdierer und einem D-Flipflop besteht. Eine Logikzelle ist wiederum ein Bestandteil eines Logikblocks, der aus einer Reihe von Logikzellen besteht. So können zum Beispiel die Übertragsbits der Volladdierer in einem Logikblock verbunden werden, um eine mehrbittige Addition durchzuführen.

Zwischen den Logikblocks verlaufen Leitungen, die an den Verbindungspunkten mit den Ein- und Ausgängen der Logikzellen an konfigurierbare Verbindungen(Verbindungsbox) zusammengeführt werden, um die Signale zu propagieren. Somit können die jeweiligen Signale der LUTs mit der passenden

[1] FPGAs mit LUTs mit mehr als Vier Eingangbits sind theoretisch möglich

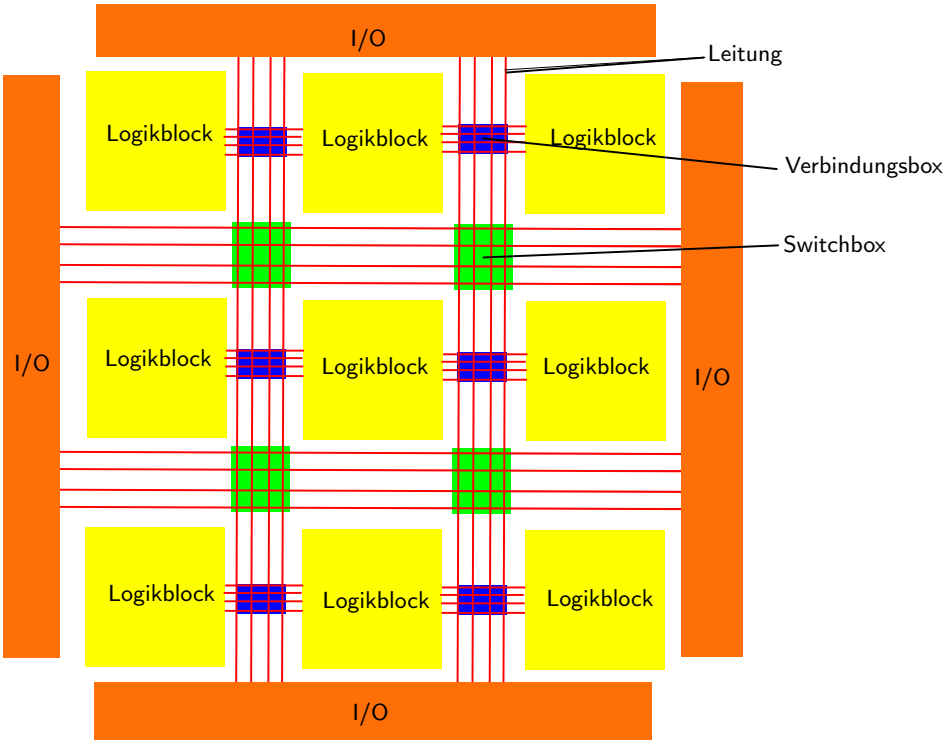


Abbildung 2.1: Grundsätzliches Aufbau eines FPGAs

Leitung verbunden werden. Um zu ermöglichen, dass ein Signal überall im FPGA hingeführt werden kann, sind die Leitungen in horizontale und vertikale Leitungen unterteilt. An den Schnittpunkten der Leitungen werden konfigurierbare Switches(Switchbox) vorgesehen. In dem die Signale zwischen den vertikalen und horizontalen Leitungen wechseln können, kann ein Signal potenziell jede Stelle des FPGAs erreichen. An den Rändern eines FPGAs erreichen die Leitungen I/O Anschlüsse. Die jeweiligen I/O Pins können auch konfigurierbare Elemente beinhalten (input, output, inout, register, ...). Die Abbildung 2.1 zeigt den grundsätzlichen Aufbau eines FPGAs.

Zusätzlich zu der oben genannten Hardware integrieren viele FPGA-Hersteller spezialisierte Hardware wie Analog-to-Digital Converters (ADCs), Digital-to-Analog Converters (DACs), Phase-Locked Loops (PLLs) und Random Access Memories (RAMs) in ihre Bausteine. Durch die Verwendung eines dedizierten RAMs können zum Beispiel Platzersparnisse erzielt werden. Das Implementieren eines RAMs aus Logikzellen ist platzineffizient und reserviert Logikzellen, die dann nicht für die Implementierung der eigentlichen Logik genutzt werden können. Manche analogen Funktionen wie zum Beispiel eine PLL sind in einer digitalen Schaltung nicht implementierbar, weshalb solche Hardware oft zusätzlich in einem FPGA integriert wird.

Im Gegensatz zu FPGAs sind anwendungsspezifische integrierte Schaltkreise (engl. Application Specific Integrated Circuits, ASICs) nicht konfigurierbar. Dafür benötigt die resultierende Schaltung weniger Platz auf dem Chip. Beide Technologien erfüllen dennoch den Zweck einer anwendungsspezifischer Implementierung. In der Regel werden FPGAs für das Prototyping oder in Endprodukten genutzt, bei denen die Stückzahl den Aufpreis bei der Entwicklung eines ASICs nicht rechtfertigt. Bei einer entsprechend großen Stückzahl kann die Entwicklung eines ASICs kostensparender sein. Ein anderer Anwendungsfall für einen ASIC liegt vor, wenn ein FPGA für die benötigten Funktionen nicht geeignet ist. Dies kann der Fall sein, wenn ein FPGA nicht die benötigte Leistung liefern kann, oder wenn eine bestimmte analoge Funktion benötigt wird und kein geeigneter FPGA mit dieser Funktion gefunden werden kann.

Ein ASIC ist kein bereits existierender Chip der genutzt werden kann. Ein ASIC muss entworfen werden und am Ende bei einem Chiphersteller produziert zu werden. Der Unterschied zu anderen auf dem Markt erhältlichen Chips ist, dass ein ASIC für einen bestimmten Zweck entwickelt wird und seine Funktionalitäten nur auf diesen Zweck zugeschnitten werden. Ein sonst erhältlicher Chip könnte zusätzliche Funktionen bieten, die für den bestimmten Zweck nicht benötigt werden.

Die Herstellung eines ASICs unterscheidet sich im Grundprinzip nicht von einer sonstigen Chipherstellung. Um die Produktion von ASICs kostensparender zu machen, existieren spezielle Herstellungsprozesse die bei ASICs genutzt werden. Somit können ASICs basierend auf dem Entwurfs- und Herstellungsprozess kategorisiert werden.

Gate-Array-ASICs ähneln den FPGAs in dem sie aus Reihen von Gattern bestehen. Diese Gatter werden auf Wafern vorproduziert und Liegen bei den Chipherstellern vorrätig vor. Beim ASIC Entwurf werden lediglich die Verbindungen zwischen den Gattern definiert. Diese Verbindungen werden bei der Produktion auf den vorgefertigten Wafern nachgeätzt.

Bei Standard-Cell-ASICs sind die Wafer nicht mehr vorgefertigt, sondern werden für jeden Entwurf angepasst. Die einzelnen Logikzellen werden weiterhin in Reihen angeordnet. Die jeweiligen Funktionen der Logikzellen werden durch die sogenannten Standardzellen definiert. Eine Standardzelle kann zum Beispiel ein UND-Gatter, ein ODER-Gatter, ein Flip-Flop oder auch ein analoges Element wie ein ADC oder eine PLL sein. Damit wird Platz auf dem Chip gegenüber den Gate-Arrays gespart und in der Regel eine höhere Leistung ermöglicht.

Die letzte Kategorie sind die Full-Custom-ASICs. Hier muss die Platzierung jedes einzelnen Transistors bei dem Entwurf definiert werden. Dies ermöglicht in der Regel weitere Platzersparnisse und Leistungserhöhungen, ist aber mit höherem Entwurfsaufwand verbunden. Es ist prinzipiell möglich, Teile des Entwurfs mit einer Standardzellenmethodologie zu entwerfen und nur die wichtigsten Teile des Chips mit der Full-Custom Methodologie zu ergänzen.

Wie bei FPGAs ist es üblich, dass Teile des Chips mit spezialisierten Schaltungen versehen sind, um zum Beispiel bei einem Gate-Array-ASIC eine analoge Funktionalität zu ermöglichen, oder bei einem Standard-Cell-ASIC einen vordefinierten Mikrocontroller zu implementieren. Es ist auch möglich Teile eines ASICs wie einen FPGA zu entwickeln. Damit wäre ein Teil des Chips fest verdrahtet, während ein anderer Teil des Chips, wie bei einem FPGA nach der Produktion angepasst werden kann.

Zusammenfassend kann festgestellt werden, dass sowohl FPGAs als auch ASICs im Prinzip aus Zellen und den Verbindungen zwischen den Zellen bestehen. Dabei kann eine Zelle ein einzelnes Logikgatter, ein gesamtes RAM-Block oder sogar ein kompletter Mikrocontroller sein.

2.2 Hardwarebeschreibungssprachen

Die Hardware Description Languages (HDLs) Verilog und VHSIC Hardware Description Language (VHDL) unterscheiden sich von Programmiersprachen, indem sie in erster Linie keine sequentiellen Vorgänge beschreiben. Dies ist wichtig, um das Verhalten von reellen digitalen Schaltungen modellieren zu können. In Hardware kann sich der Zustand von mehreren Signalen zu gleichen Zeit ändern. Diese Eigenschaft muss in einer HDL darstellbar sein. Da Yosys ein Syntheseprogramm für Verilog ist, wird Verilog als Beispiel einer HDL genutzt. Bis auf die Syntax und einige Besonderheiten ist VHDL ähnlich aufgebaut.

Um in Verilog Logik zu implementieren, müssen Module entwickelt werden. Ein Modul fasst Logik in einer zusammenhängenden Einheit zusammen. Die Schnittstelle eines Moduls besteht aus einer Reihe von Signalen(Ports), die als ein Eingang, Ausgang oder Ein- und Ausgang definiert werden können. Ein Modul kann Parameter festlegen mit denen das Modul später angepasst werden kann. So kann ein Modul beispielsweise einen Addierer definieren, bei dem die Anzahl der Bits einstellbar ist. Das Listing 2.1 zeigt die Syntax für Definition eines Moduls.

```
1 module modul_name #(parameter parameter_name=default_wert, ...)
2   (input signal_name1, output signal_name2, inout signal_name3, ...);
3
4   //Hier wird die Logik des Moduls implementiert
5
6 endmodule
```

Listing 2.1: Syntax einer Verilog-Moduldefinition

Innerhalb des Moduls können interne Signale und Variablen definiert werden. Ein Signal stellt dabei ein Netz dar und wird mit dem Schlüsselwort `wire` definiert. Eine Variable wird meistens als `reg` definiert und kann als ein Register, das in der Regel als ein Flip-Flop implementiert wird, dargestellt werden. Der wesentliche Unterschied zwischen Signalen und Variablen zeichnet sich durch die Stellen im Entwurf, in denen sie genutzt werden können. Variablen können in prozeduralen Teilen des Entwurfs eingesetzt werden, Signale dagegen nicht. Andere Typen, die Variablen annehmen können sind `integer`, `time`, `realtime`, `real`.

Sowohl Signale als auch Variablen können als Vektoren definiert werden. Damit können im Fall von Netzen Busse und im Fall von Variablen n-Bit Register definiert werden. Um ein Vektor zu definieren, werden die Indizes vor dem Namen des Netzes oder der Variable in eckigen Klammern in der Form `[MSB:LSB]`[2] angegeben.

Arrays benutzen die gleiche Syntax für Indizes mit dem Unterschied, dass die Indizes nach dem Namen des Netzes oder der Variable definiert werden. Solche Definitionen können bei der Synthese zu einem Speicher umgewandelt werden. Der Unterschied zwischen einem Vektor und einem Array ist, dass der Vektor einen zusammenhängenden Bus darstellt und ein Array mehrere einzelne Signale oder Vektoren definiert. Durch die Kombination der beiden Schreibweisen kann zum Beispiel ein n-Bit Speicher mit m Worten der Länge n definiert werden. Weiter sind auch mehrdimensionale Vektoren und Arrays möglich. Ports eines Moduls unterstützen ebenfalls die Vektorschreibweise mit der Besonderheit, dass nur eindimensionale Vektoren definiert werden können. Das Listing 2.2 zeigt die verschiedenen Definitionsmöglichkeiten.

Um einem Signal ein Wert zuzuweisen, wird die `assign`-Anweisung genutzt. Auf der linken Seite der Zuweisung muss ein Netz vorliegen. Eine Variable wie ein `reg` kann nicht mit der `assign`-Anweisung zugewiesen werden. Es ist allerdings möglich auf der linken Seite der Zuweisung eine Verkettung von mehreren Netzen anzugeben. Auf der rechten Seite der Zuweisung dürfen Konstanten, Netze, Variablen oder Aufrufe von Funktionen[3], die einen geeigneten Wert zurückgeben, stehen. Das Listing 2.3 zeigt die Beispiele von `assign`-Anweisungen.

Zuweisungen mit der `assign`-Anweisung wirken alle zur gleichen Zeit. Das bedeutet, dass es keine Reihenfolge gibt, in der die Signale zugewiesen werden. Sobald die rechte Seite der Zuweisung ihren Wert ändert, ändert sich auch der Wert auf der linken Seite. Sollten einem Signal zwei widersprüch-

[2] Most Significant Bit/Byte (MSB) und Least Significant Bit/Byte (LSB)
[3] In Verilog können Funktionen, die aus mindestens einem Eingang ein Rückgabewert produzieren, definiert werden

```
1  input [3:0] signal_name;          // Legal
2  input signal_name [3:0];          // Illegal
3  input [3:0] signal_name [7:0];    // Illegal
4  input [3:0][7:0] signal_name;     // Illegal
5
6  //wire kann mit reg ersetzt werden
7  wire signal_name; // Ein Signal/Netz
8  wire [3:0] signal_name; // Ein 4-Bit Vektor/Bus
9  wire signal_name [3:0]; // Ein Array von 4 einzelnen Signalen
10 wire [3:0] signal_name [7:0]; // Ein Array von 8 4-Bit Vektoren
11 wire [3:0] signal_name [7:0][1:0]; // Ein 2-D Array von 8x2 4-Bit Vektoren
12 wire [3:0][7:0]signal_name; // Ein Vektor von 8 4-Bit Vektoren
13 wire [3:0] sig1 [1:0], sig2 [2:0]; // sig1 und sig2 sind Arrays von n-Bit Vektoren
14 wire [3:0] sig1, [1:0] sig2; // Illegal
```

Listing 2.2: Syntax für die Definitionen von Vektoren und Arrays

```
1  wire [3:0] a;             // 3-Bit Vektor
2  wire b;                   // 1-Bit Signal
3  wire [4:0] c;             // 4-Bit Vektor
4
5  assign {a, b} = c;        // a = c[4:1], b = c[0]
6  assign c = {a, 1'b1};     // c[4:1] = a, c[0] = 1
7  assign a[2:1] = 2'b11;
8  assign a[2:1] = c[3:2];
```

Listing 2.3: Syntax für die Zuweisungen mit der `assign` Anweisung

liche Werte zugewiesen werden, kann es zu Konflikten kommen. Das kann zum Beispiel der Fall sein, wenn einem Signal in zwei Zuweisungen einmal eine Eins und einmal eine Null zugewiesen wird. Das Signal wird dann undefiniert.

Die rechte Seite einer Zuweisung kann auch ein Ergebnis eines Logikausdrucks sein. So kann einem Signal zum Beispiel das Ergebnis einer UND-Verknüpfung zugewiesen werden. Ein Ausdruck kann beliebig komplex werden. Möglich ist auch die Modellierung von Multiplexern. Das Listing 2.4 zeigt einige Beispiele für Ausdrücke in Verilog.

```
1  wire a, b, c, d, e;
2  wire out;
3
4  assign out = a & b; // out ist das Ergebnis einer UND-Verknüpfung von a und b
5  assign out = (b | c) & (d | e); // UND-Verknüpfung von zwei ODER-Verknüpfungen
6  assign out = a || b; // Verknüpfung mit logischem ODER
7  assign out = a ? (b | c) : d; // Multiplexer mit dem ternären Operator
8  //a = 1 ⇒ out = (b | c), a = 0 ⇒ out = d
```

Listing 2.4: Beispiele von Ausdrücken in Verilog

Ein Hardwareentwurf kann hierarchisch aufgebaut werden. Das bedeutet, dass innerhalb eines Moduls Instanzen eines anderen Moduls instanziiert werden können. Bei der Instanziierung werden Parameterwerte zugeteilt und die Ports des Untermoduls mit definierten Signalen des Hauptmoduls verbunden. Die Zuweisung der Werte kann basierend auf der Position der Ports in der Untermodul-

definition erfolgen oder anhand der Namen der Ports angegeben werden. Das Listing 2.5 zeigt die Syntax einer Instanziierung. Auf diese Weise lassen sich nicht nur andere Module instanziieren, sondern es kann auch eine Instanz einer Komponente verwendet werden, die bereits auf der Zielhardware verfügbar ist, wie zum Beispiel eine PLL.

```
1 // Die Instanziierung mit positionellen Zuweisungen
2 module_name #(p1, p2, ...) instance_name (sig1, sig2, ...);
3
4 // Die Instanziierung mit Zuweisungen mit Portnamen
5 module_name #(.p1(p1), .p2(p2), ...) instance_name (
6   .port1(sig1),
7   .port2(sig2),
8   ...
9 );
```

Listing 2.5: Syntax für die Instanziierung eines Moduls oder einer Zelle

Mit `always`-Blöcken ermöglicht Verilog die Implementierung von sequentieller Logik. Diese Blöcke werden auch als Prozesse bezeichnet. In Prozessen wird eher das Verhalten der resultierenden Schaltung beschrieben als die eigentliche Schaltung. In einem Prozess kann zwischen einzelnen Anweisungen keine zeitliche Verzögerung festgestellt werden. Das hinterlässt den Eindruck als würden die Anweisungen eines Prozesses zu gleichem Zeitpunkt wirksam, dennoch kann die Reihenfolge unter Umständen wichtig sein. Eine Anweisung kann die Zuweisung einer Variable sein. Anders als bei der `assign`-Anweisung können in Prozessen nur Variablen und keine Signale zugewiesen werden.

Es wird in einem Prozess zwischen blockierenden(=) und nicht blockierenden(<=) Zuweisungen unterschieden. Die Ausführung der Zuweisungen in einem Prozess geschieht gleichzeitig. Bei blockierenden Zuweisungen ist der zugewiesene Wert unmittelbar nach der Zuweisung in der Variable vorzufinden. Bei nicht blockierenden Zuweisungen ist der Wert in der Variable erst nach der vollständigen Ausführung des `always`-Blocks vorzufinden.[4] Bei nicht blockierenden Zuweisungen hat somit der zuletzt in dem Prozess zugewiesene Wert Vorrang bei mehreren Zuweisungen der gleichen Variable. Nicht blockierende Zuweisungen werden in der Regel für sequentielle Logik verwendet, die mit einer Clock synchronisiert ist. Dabei können blockierende Zuweisungen zu temporären Variablen genutzt werden, um komplexe Ausdrücke aufzuteilen. Blockierende Zuweisungen werden in der Regel in kombinatorischen Prozessen genutzt. Ein kombinatorischer Prozess hängt zu jeder Zeit von den Eingangssignalen ab und wird nicht synchronisiert. In der Regel sollten die blockierenden und nicht blockierenden Zuweisungen nicht in einem Prozess gemischt werden, es ist allerdings möglich. Das Listing 2.6 zeigt die Funktionsweise der verschiedenen Arten von Zuweisungen.

Andere Anweisungen, die in einem Prozess vorgefunden werden können, sind die `if`- und `case`-Anweisungen. Mit diesen Anweisungen ist es möglich, Entscheidungsbäume in einem Prozess zu implementieren. Eine `if`-Anweisung beinhaltet eine Bedingung, die angibt, wann die Anweisungen innerhalb der `if`-Anweisung ausgeführt werden sollen. Mit `else if` kann eine weitere Bedingung eingeführt werden, die geprüft wird, wenn die vorherige Bedingung nicht eingetreten ist. Mit `else` wird angegeben, welche Anweisungen ausgeführt werden, wenn keine vorherigen Bedingungen eingetreten sind.

[4] In der Simulation können zeitliche Verzögerungen angegeben werden. Nach einer Verzögerung werden die Werte in den Variablen auch übernommen. Verzögerungen sind nicht synthetisierbar.

```
wire a, b, c;
reg q;

always begin
  q <= a & b; // Wird nie eintreten denn die nächste Zuweisung überschreibt den Wert
  q <= q | c; // q hängt von dem letztem Wert von q und von dem aktuellen c Wert ab

  q = a & b; // q wird a & b zugewiesen
  q = q | c; // Zu dem Zeitpunkt befindet sich in q das Ergebnis von a & b.
  //Die eigentliche Logik ist also q = (a & b) | c;
end
```

Listing 2.6: Blockierende und nicht Blockierende Zuweisungen

Bei einer `case`-Anweisung wird ein Ausdruck angegeben, dessen Ergebnis mit angegebenen Fällen verglichen wird. Für jeden Fall werden Anweisungen angegeben, die ausgeführt werden sollen, wenn der Ausdruck der `case`-Anweisung mit dem Ausdruck des Falls übereinstimmt. Der `default` Fall gibt die Anweisungen an, die ausgeführt werden, wenn kein anderer Fall eingetreten ist. Das Listing 2.7 zeigt die Syntax der Anweisungen.

```
wire a, b, c;
reg q1, q2;
always begin
  if (a) begin                // Wenn a gleich 1 ist
  q1 <= 1'b1;
  q2 <= 1'b0;
  end else if (b == 1'b0)     // Wenn b gleich 0 ist
  q1 <= 1'b0;
  else begin                  // Ansonsten
  q1 <= 1'b1;
  q2 <= 1'b1;
  end
  case (a & b & c)
  1'b1: begin                 // Wenn a, b und c gleich 1 sind
    q1 <= 1'b1;
    q2 <= 1'b0;
  end
  1'b0: q2 <= 1'b1;           // Wenn a, b oder c nicht gleich 1 sind
  default: begin              // In allen anderen Fällen
    q1 <= 1'b1;
    q2 <= 1'b1;
  end
  endcase
end
```

Listing 2.7: `if` und `case`-Anweisungen

In einem Prozess kann eine sogenannte Sensitivitätsliste angegeben werden. Diese Liste gibt an, wann der Prozess ausgeführt wird. Die Angabe eines Signals bedeutet, dass die Logik des Prozesses immer dann ausgeführt wird, wenn das Signal seinen Wert ändert. Weiter kann mit `posedge` und `negedge` für ein Signal festgelegt werden, dass der Prozess nur bei den positiven oder negativen Flanken des Signals ausgeführt werden soll. Das Listing 2.8 zeigt ein Beispiel eines Prozesses mit einer Sensitivitätsliste und einem asynchronem Reset. Bei einer positiven Flanke des clk Signals wird

```
always @(posedge clk, negedge rst) begin
  if (rst == 0)
  // Die Anweisungen für einen Reset
  else
  // Die normalen Anweisungen nach einem Taktzyklus
end
```

Listing 2.8: Ein Prozess mit einer Sensitivitätsliste

der Block ausgeführt. Ist rst nicht 0, werden die Anweisungen, die nach einem Taktzyklus stattfinden sollen, durchgeführt. Andernfalls wird ein Reset ausgeführt, wenn der Block wegen einer negativen Flanke des rst Signals aktiviert wurde, denn nach einer negativen Flanke ist das rst Signal gleich 0.

```
module top_module(
  input clk,
  input rst,
  input [1:0] btn,
  output [1:0] led);
  wire [1:0] neg_btn;
  wire [3:0] pmr;

  assign neg_btn = ~btn;

  assign led = {pmr[1:0] == pmr[3:2], pmr[0] | pmr[1] | pmr[2] | pmr[3]};

  proc_module pm (
  .clk(clk),
  .rst(rst),
  .a(neg_btn[0]),
  .b(neg_btn[1]),
  .q(pmr)
  );
endmodule

module proc_module(
  input clk,
  input rst,
  input a,
  input b,
  output reg [3:0] q);

  always @(posedge clk)
  if (rst == 0)
    q <= 0;
  else case ({a, b})
    2'b00: q <= q;
    2'b01: q <= q + 1;
    2'b10: q <= q + 2;
    2'b11: q <= q + 4;
  endcase
endmodule
```

Listing 2.9: Ein Beispiel eines Verilogentwurfs

Das Listing 2.9 zeigt ein Beispiel eines kompletten Hardwareentwurfs in Verilog. Das Modul `proc_module` stellt ein Beispiel Modul mit einem Prozess dar. Der Prozess ist mit einem Takt synchronisiert und der Reset ist synchron implementiert. Abhängig von den Eingängen wird der Register `q` um eins, zwei oder vier inkrementiert oder behält seinen Wert. In dem `top_module` wird eine Instanz des `proc_module` instanziiert. `clk` und `rst` werden ohne Änderungen an die Instanz übergeben. Im Falle der Taster(`btn`) werden diese vorerst in der Zeile 9 negiert und erst dann übergeben. Der Wert des Registers `q` soll über Light Emitting Diodes (LEDs) angezeigt werden. Die erste LED leuchtet, wenn die ersten Bits des Registers mit den letzten Bits übereinstimmen. Die zweite LED leuchtet, wenn mindestens ein Bit des Registers gleich Eins ist. Die Zuweisung findet in der Zeile 11 statt.

2.3 Parsing und Abstract Syntax Tree (AST)

Der Einlesevorgang eines Verilogentwurfs kann im Prinzip ähnlich wie ein Kompiliervorgang aufgebaut werden. Im Kern des Vorgangs liegt wie in gängigen Kompilierern die Generierung einer Abstract Syntax Tree (AST) Struktur[13, S. 3]. Bei einer Kompilierung können zwei wesentliche Phasen festgestellt werden. Die Analysephase, in welcher der Quelltext auf lexikalische, syntaktische und semantische Korrektheit überprüft wird und anschließend ein AST aufgebaut wird, und die Synthesephase bei der auf dem AST Optimierungen ausgeführt werden und anschließend eine ausführbare Datei produziert wird[13, S. 3].

Eine solche Unterteilung kann auch bei Yosys festgestellt werden. Die Analysephase dient der Verarbeitung des Quelltextes und dessen Bereitstellung in einer geeigneten Form. Die Synthesephase führt Optimierungen auf dem Entwurf aus und wandelt die in Hardware nicht darstellbaren in darstellbare Konstrukte um[13, S. 13].

In Verilog können Kompiliereranweisungen und Makros angewendet werden. Damit können Teile des Entwurfs noch vor dem eigentlichen Einlesevorgang angepasst werden. So kann zum Beispiel Quelltext, der nur für die Simulation relevant ist, für die Synthese ausgelassen werden. Die Aufgabe, diese Anweisungen zu finden und zu verarbeiten, übernimmt der Präprozessor[13, S. 16].

Ein mit dem Präprozessor verarbeiteter Quelltext wird in der lexikalischen Analyse weiter verarbeitet. Der Teil des Programms, der für die lexikalische Analyse zuständig ist, wird Lexer genannt. Dort werden die einzelnen Sprachkonstrukte erkannt und mit sogenannten Tokens ersetzt. Ein Token identifiziert das jeweilige Sprachkonstrukt und hält gegebenenfalls dessen Wert[13, S. 4]. Ein lexikalisch korrekter Quelltext besteht nach der lexikalischen Analyse ausschließlich aus Tokens.

Die lexikalische Analyse und die Umwandlung in Tokens vereinfacht die Implementierung der syntaktischen und semantischen Analyse im sogenannten Parser. Ein Parser prüft die Position und Reihenfolge der Tokens auf Korrektheit in Bezug auf die definierte Grammatik. Eine Grammatik ist die Menge aller Regeln einer Sprache. Während des Parsings wird nach und nach der AST aufgebaut[13, S. 8-9].

Ein AST ist eine baumförmige Datenstruktur, deren Aufgabe es ist, die Syntax eines Programms hierarchisch abzubilden. Ein erstellter AST weist nach dem Parsing in der Regel Redundanzen auf. Durch die Entfernung der Redundanzen bzw. durch die Vereinfachung des ASTs erhält man eine Struktur mit der die eigentliche Synthese prinzipiell durchführbar ist[13, S. 31]. In Yosys wird der AST genutzt, um eine interne Datenstruktur, die besser für eine Synthese geeignet sei, zu füllen[14, S. 29]. Die eigentliche Synthese wird dann auf der internen Datenstruktur durchgeführt.

2.4 Netzlisten

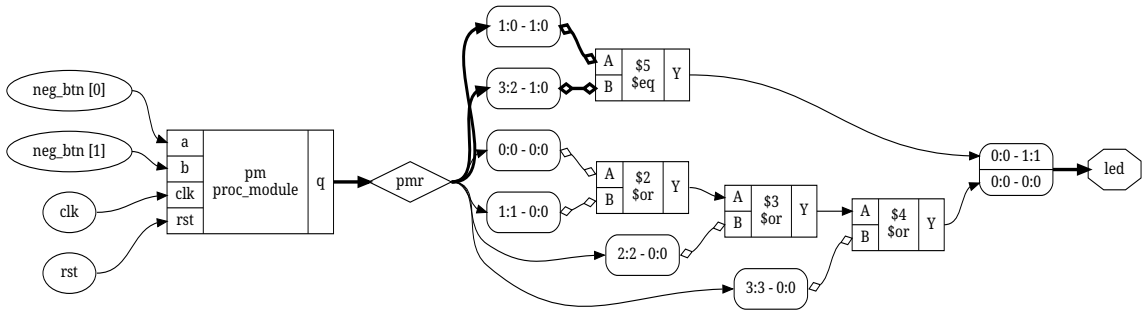


Abbildung 2.2: Visualisierung einer Netzliste des Entwurfs aus Listing 2.9 vor der Synthese

Eine Netzliste dient zur logischen Darstellung einer Schaltung. Logisch heißt in diesem Fall, dass die physische Position der einzelnen Elemente nicht berücksichtigt wird. Wichtig ist die Identifizierung der Elemente der Schaltung und die Verbindungen zwischen den Elementen.

In der simpelsten Form besteht eine Netzliste somit aus Knoten und Netzen. Es ist möglich in einer Netzliste weitere Informationen in Form von Attributen mitzuspeichern. Damit können einzelne Knoten für eine bestimmte Anwendung angepasst werden.

Grundsätzlich kann ein Knoten eine gesamte Instanz eines Moduls darstellen. Für die Implementierung eines Entwurfs auf einer reellen Hardware muss eine Netzliste aus Knoten bestehen, die einer auf dem Chip implementierbaren Hardware entsprechen. Dies können zum Beispiel Logikzellen, Flip-Flops oder Speicherzellen sein.

Die Abbildung 2.2 zeigt eine Visualisierung einer Netzliste des Entwurfs aus dem Listing 2.9. Visualisiert wurde das `top_module`. Das `proc_module` kann vor der Synthese auf Grund der Verwendung eines Prozesses nicht vernünftig visualisiert werden.

Das Listing 2.10 zeigt die grundlegende Syntax einer Netzliste. Eine Netzliste ist eine Auflistung aller bekannten Module. Ein Attribut kann Metadaten für einen Modul enthalten. Die Parameter stellen die Verilog-Parameter dar. Mit Ports wird die Schnittstelle des Moduls für die Wiederverwendung in einem anderen Modul definiert. Eine Zelle kann sowohl eine physische Hardware darstellen, als auch ein anderes Modul instanziieren. In `memories` können Speicherdefinitionen angegeben werden. Anschließend werden Netze definiert. Um Verbindungen zwischen den Elementen umzusetzen, definieren Netze, Ports und Zellen die Signale, mit denen sie verbunden sind. Ein Signal ist in einer Netzliste ein ganzzahliger Identifikator. Existiert ein Netz mit der Signalnummer drei und zwei Zellen, die beide mit der Signalnummer drei verbunden sind, so sind die Zellen miteinander verbunden.

```
"modules": {
  "top_module": {
    "attributes": {
      ...
    },
    "parameter_default_values": {
      ...
    },
    "ports": {
      ...
    },
    "cells": {
      ...
    },
    "memories": {
      ...
    },
    "netnames": {
      ...
    }
  }
}
```

Listing 2.10: Die Syntax einer von Yosys produzierten Netzliste

2.5 Synthese

Um ein Entwurf in der Hardware nutzen zu können, muss er in eine Darstellung, die nur aus Zellen, Verbindungen und gegebenenfalls Speichern besteht, gebracht werden. Kombinatorische Logik, wie Und-Gatter, Vergleichsgatter oder Addierer, müssen in kombinatorische Zellen umgewandelt werden. Enthält der Entwurf keine Prozesse, ist es bereits möglich den letzten Schritt der Synthese auszuführen, nämlich das Technologiemapping. Dabei werden die bestehenden Zellen mit Zellen, die in der Zielhardware existieren, ersetzt. Existieren Prozesse im Entwurf müssen sie vorerst aufgelöst werden. Dabei kann beispielsweise folgende Logik erkannt und entsprechend als Zellen dargestellt werden:

- Asynchrone Resets
- D-Flip-Flops (DFFs)
- Latches
- Multiplexerbäume als Entscheidungsbäume aus den `if`- und `case`-Anweisungen
- Read Only Memories (ROMs)

Nachdem die Prozesse aufgelöst wurden, kann der Entwurf vor dem Technologiemapping optimiert werden. Dabei wird überflüssige Logik festgestellt, wie z.B. eine doppelte Negation, oder die bestehenden Zellen werden zu größeren Zellen zusammengefasst, welche die eigentliche Aufgabe besser bewältigen können. Da auch Speicher in einer Netzliste angegeben werden können, folgt, dass es nach der Synthese Werkzeuge gibt, welche mit einer abstrakten Speicherbeschreibung umgehen können. Die Alternative ist die Speicher eines Entwurfs ebenfalls als einzelne Zellen darzustellen.

3 Yosys

In diesem Kapitel soll Yosys im Detail untersucht werden. Dafür ist zunächst ein Verständnis der Programmstruktur notwendig. In der Softwarestrukturanalyse wird der Programmfluss und die Erweiterbarkeit von Yosys untersucht. Yosys führt die Synthese nicht direkt auf dem AST aus, sondern implementiert eine spezielle Datenstruktur RTLIL und ein textbasiertes Austauschformat, mit dem die Inhalte der Datenstruktur serialisiert und deserialisiert werden können. Dafür werden die Inhalte des AST noch vor der Synthese in RTLIL überführt. Die Synthese wird dann mit Hilfe von RTLIL durchgeführt. Sowohl das Austauschformat als auch die interne Darstellung im Code sollen erläutert werden. Dieses Hintergrundwissen soll im Unterkapitel 3.3 das Verständnis der Analyse der eigentlichen Yosys-Synthese unterstützen. Zum Schluss wird die Ausgabe in Form einer Netzliste beschrieben.

3.1 Softwarestrukturanalyse

Im Rahmen der Arbeit wird die Softwarestrukturanalyse in zwei Teile unterteilt. Zum einen ist im folgendem die Yosys-Struktur von Interesse. Zum zweiten soll die Erweiterbarkeit von Yosys gesondert behandelt werden.

3.1.1 Strukturanalyse

Der Quelltext von Yosys ist in einer Verzeichnisstruktur geordnet, welche in Abbildung 3.1 dargestellt ist. Die markierten Verzeichnisse enthalten Quelldateien, die für die Synthese relevant sind und untersucht werden. Der Einstiegspunkt von Yosys befindet sich im `kernel`-Verzeichnis, wo mitunter auch die Definitionen der RTLIL-Datenstrukturen, allgemeine Zellendefinitionen, die Unterstützung von Plugins und sonstige Hilfsfunktionen abgelegt sind.

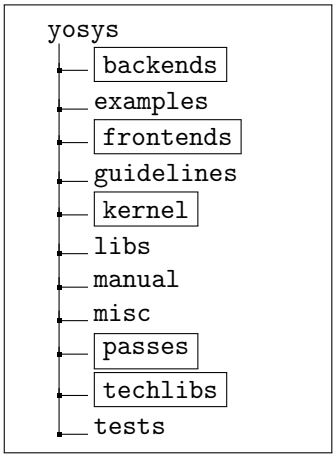


Abbildung 3.1: Yosys Dateibaum

Das passes-Verzeichnis beinhaltet die eigentlichen Prozessschritte der Synthese und sonstige Befehle, die in der Yosys-Shell[1] genutzt werden können. Im frontends-Verzeichnis werden unter anderem Befehle definiert, die das Einlesen eines Designs ermöglichen. Dort ist z.B. das Verilog und RTLIL-Frontend definiert. Das backends-Verzeichnis definiert Befehle, die für die Ausgabe des Designs zuständig sind. Dort ist zum Beispiel das Netzlisten-Backend definiert. Mit den RTLIL und Verilog-Backends ist es beispielsweise möglich, das Design im RTLIL-Format oder als Verilog-Quelltext zu speichern.

Unter dem techlibs-Verzeichnis befinden sich sämtliche Technologiedefinitionen für verschiedene FPGAs. Befehle für die Durchführung einer Synthese für die jeweiligen FPGAs sind hier zusammen mit den Definitionen untergebracht.

Yosys stellt eine interaktive Benutzer-Shell bereit, in der die jeweiligen Befehle aus den verschiedenen Verzeichnissen ausgeführt werden können. Es besteht die Möglichkeit, ein Skript oder eine Abfolge von Befehlen direkt beim Start von Yosys anzugeben. In diesem Fall arbeitet Yosys ohne Benutzereingabe und führt die angegebenen Aktionen durch.

```
348   int opt;
349   while ((opt = getopt(argc, argv, "MXAQTVSgm:f:Hh:b:o:p:l:L:qv:tds:c:W:w:e:r
        :D:P:E:x:")) != -1)
350   {
351     switch (opt)
352     {
353     case 'M':
354       memhasher_on();
355       break;
356     case 'X':
357       yosys_xtrace++;
358       break;
484     default:
485       fprintf(stderr, "Run '%s -h' for help.\n", argv[0]);
486       exit(1);
487     }
488   }
```

Listing 3.1: Verarbeitung der Befehlszeilenoptionen in Yosys[2]

Der Einstiegspunkt bei der Ausführung von Yosys befindet sich in der Datei kernel/driver.cc in Zeile 195. In der Main-Funktion werden Variablen definiert, welche die Ausführung des Programms steuern. Die wichtigsten Variablen und deren Bedeutungen sind in der Tabelle 3.1 zusammengefasst. Das Program startet mit der Wiederherstellung der bisherigen Befehlshistorie des Programms. Nachfolgend werden gegebenenfalls Hilfe oder Versionsausgaben getätigt, wenn die entsprechenden Befehlszeilenoptionen angegeben wurden. Das Listing 3.1 zeigt ein Ausschnitt aus der Verarbeitung der Befehlszeilenoptionen. In der Regel werden die in der Tabelle 3.1 angegebenen Variablenwerte basierend auf den Optionen verändert. Eine unvollständige Zusammenfassung der wichtigsten Befehlszeilenoptionen und deren Auswirkungen befindet sich in Tabelle 3.2. Einige der Optionen verändern zusätzlich den Wert in run_shell zu false. Dies hat die Auswirkung, dass die interaktive Shell von Yosys nicht ausgeführt wird. Die Optionen mit dieser Eigenschaft sind: -S, -b, -p, -o, -s und -c. Dies ist sinnvoll, da jede dieser Optionen einen gewissen Automatisierungsgrad impliziert. Mit der -s Option wird zum Beispiel ein Befehl, der mehrere einzelne Befehle zusammenfasst,

[1] Befehlszeilenschnittstelle

Variable	Standardwert	Bedeutung
frontend_command	"auto"	Spezifiziert den Namen des Frontends, mit dem die Eingabedateien eingelesen werden sollen. Der Wert "auto" bedeutet, dass das Frontend abhängig von der Dateiendung gewählt wird.
backend_command	"auto"	Spezifiziert den Namen des Backends, mit dem die Ausgabedatei ausgegeben werden soll. Der Wert "auto" bedeutet, dass das Backend abhängig von der Dateiendung gewählt wird.
vlog_defines	leerer Vektor	Speichert die Verilog Defines, die vor der Ausführung eines Frontends gesetzt werden sollen.
passes_commands	leerer Vektor	Speichert die Befehle, die Yosys nach der Ausführung der Frontends ausführen soll.
plugin_filenames	leerer Vektor	Speichert die Namen der Plugins, die geladen werden sollen.
output_filename	leerer String("")	Speichert den Namen der Ausgabedatei.
scriptfile	leerer String("")	Speichert den Dateinamen eines Skripts, das ausgeführt werden soll.
topmodule	leerer String("")	Speichert den Namen des Topmoduls. Das Topmodul ist das Modul, das an der obersten Stelle in der Modulhierarchie steht.
scriptfile_tcl	false	Gibt an, ob das angegebene Skript ein TCL-Skript ist oder nicht.
run_shell	true	Gibt an, ob die interaktive Shell ausgeführt werden soll.

Tabelle 3.1: Variablen der Main-Funktion und deren Bedeutungen

Option	Syntax	Funktion
-S	yosys -S	Fügt den synth Befehl zu dem passes_commands Vektor hinzu. Der synth Befehl führt eine Zielhardware unabhängige Synthese durch.
-m	yosys -m <plugin_file>	Fügt einen Pluginnamen zu plugin_filenames hinzu.
-f	yosys -f <frontend>	Setzt den Wert von frontend_command.
-b	yosys -b <backend>	Setzt den Wert von backend_command.
-p	yosys -p "<commands>"	Fügt den String mit den Befehlen in das passes_commands Vektor hinzu.
-o	yosys -o "output_file"	Setzt den Wert von output_filename.
-s	yosys -s "script_file"	Setzt den Wert von scriptfile.
-c	yosys -c "tcl_script_file"	Setzt den Wert von scriptfile_tcl.
-r	yosys -r <module_name>	Setzt den Wert von topmodule.
-D	yosys -D <macro>[=<value>]	Fügt ein Verilog-Define zum vlog_defines.

Tabelle 3.2: Liste der wichtigsten Befehlszeilenoptionen von Yosys

ausgeführt. In diesem Fall wird davon ausgegangen, dass die Yosys-Shell nicht benötigt wird und alle auszuführenden Befehle in dem Befehl enthalten sind.

Als Nächstes gibt Yosys ein Banner aus und bereitet das Program auf die Durchführung einer Synthese vor. Von besonderer Wichtigkeit ist hier die Funktion `yosys_setup()`. Das Listing 3.2 zeigt die Definition der Funktion in der Datei `kernel/yosys.cc`. In den Zeilen 552-554 werden vordefinierte IDs (Namen) für die einzelnen Signale, Ports und Zellen generiert. In Zeile 563 werden die Befehle registriert. Dies wird weiter im Unterkapitel 3.1.2 behandelt. In der folgenden Zeile wird ein leeres Design erstellt und in einer globalen Variable gespeichert. Die Zeile 565 verknüpft die einzelnen Zellen mit Portnamen.

```
544 void yosys_setup()
545 {
546   if(already_setup)
547     return;
548   already_setup = true;
549   init_share_dirname();
550   init_abc_executable_name();
551
552 #define X(_id) RTLIL::ID::_id = "\\" # _id;
553 #include "kernel/constids.inc"
554 #undef X
563   Pass::init_register();
564   yosys_design = new RTLIL::Design;
565   yosys_celltypes.setup();
566   log_push();
567 }
```

Listing 3.2: Die Funktion `yosys_setup()` in Yosys[10]

Das Listing 3.3 zeigt den Aufruf von `load_plugin`. An dieser Stelle werden die spezifizierten Plugins geladen. Der genaue Vorgang des Ladens eines Plugins wird im Unterkapitel 3.1.2 betrachtet.

```
520   for (auto &fn : plugin_filenames)
521     load_plugin(fn, {});
```

Listing 3.3: Laden von Plugins[2]

Der Quelltext im Listing 3.4 setzt die durch die Befehlszeilenoptionen geforderten Verilog-Defines im Design. Hierfür wird der `read`-Befehl mit der Option `-define` genutzt. Dadurch werden die angegebenen Defines im Design vermerkt und vom Verilog-Frontend beim Einlesen des Entwurfs einbezogen.

```
523   if (!vlog_defines.empty()) {
524     std::string vdef_cmd = "read -define";
525     for (auto vdef : vlog_defines)
526       vdef_cmd += " " + vdef;
527     run_pass(vdef_cmd);
528   }
```

Listing 3.4: Anwendung der Verilog-Defines aus den Befehlszeilenoptionen[2]

Alle Argumente, die Yosys bei der Ausführung mitgegeben werden, die keine Optionen sind, werden von Yosys als Eingabedateien behandelt. Das heißt, dass für jedes angegebene Argument ein Frontend ausgeführt wird. Das Listing 3.5 zeigt den Aufruf der Frontends. Die Wahl des Frontends hängt von der Dateiendung und vom `frontend_command`, das durch die `-f` Option gesetzt wird, ab. Ein Argument kann der Name einer Skript-Datei sein. Wird eine solche Datei erkannt, wird das Skript ausgeführt. Dabei gibt `run_frontend` den Wert `true` zurück, was bewirkt, dass die Yosys-Shell nicht ausgeführt wird. Hierbei muss beachtet werden, dass die Reihenfolge der Argumente eine Rolle spielt, da die Frontends in der Reihenfolge ausgeführt werden. Das bedeutet, dass ein Skript nur dann mit Dateien arbeiten kann, wenn sie vor der Ausführung des Skripts bereits eingelesen worden sind. Deswegen muss ein Skript immer nach den Eingabedateien angegeben werden. Alternativ kann das Einlesen der Dateien ebenfalls im Skript enthalten sein. In diesem Fall wird die Angabe der Eingabedateien nicht benötigt.

```
530   while (optind < argc)
531     if (run_frontend(argv[optind++], frontend_command))
532       run_shell = false;
```

Listing 3.5: Ausführung der Frontends[2]

Das Listing 3.6 zeigt die Wirkung der `-r` Option. Mit der Ausführung des `hierarchy`-Befehls mit der -top Option wird das Topmodul des Designs gesetzt. Wurde ein Skript in den Optionen angegeben, so wird es, wie im Quelltext in Listing 3.7 dargestellt, ausgeführt.

```
534   if (!topmodule.empty())
535     run_pass("hierarchy -top " + topmodule);
```

Listing 3.6: Setzen des Topmoduls aus den Befehlszeilenoptionen[2]

```
537   if (!scriptfile.empty()) {
538     if (scriptfile_tcl) {
540       if (Tcl_EvalFile(yosys_get_tcl_interp(), scriptfile.c_str()) != TCL_OK)
541         log_error("TCL interpreter returned an error: %s\n",
              Tcl_GetStringResult(yosys_get_tcl_interp()));
545     } else
546       run_frontend(scriptfile, "script");
547   }
```

Listing 3.7: Ausführung des Skripts aus den Befehlszeilenoptionen[2]

Eine weitere Möglichkeit zur Befehlsausführung ist die `-p` Option. Diese Befehle werden im Quelltext im Listing 3.8 ausgeführt. Dort können zum Beispiel die benötigten Synthesebefehle angegeben werden.

```
549   for (auto it = passes_commands.begin(); it != passes_commands.end(); it++)
550     run_pass(*it);
```

Listing 3.8: Ausführung der Befehle aus den Befehlszeilenoptionen[2]

Abhängig vom bisherigen Programmablauf wird im Listing 3.9, basierend auf der `run_shell` Variable, entweder die Shell oder ein Backend ausgeführt. Wurde weder `output_filename`, noch `backend_command` spezifiziert, wird kein Backend ausgeführt. Hiernach folgen Debug bedingte Überprüfungen, Endausgaben von Yosys und die Ressourcenfreigabe.

```
552   if (run_shell)
553     shell(yosys_design);
554   else
555     run_backend(output_filename, backend_command);
```

Listing 3.9: Ausführung der Shell oder eines Backends[2]

Die Erkenntnis der Strukturanalyse ist, dass Yosys standardmäßig keine Aufgabe erfüllt. Es liegt am Nutzer, die richtigen Befehle auszuführen, um eine bestimmte Aufgabe, wie zum Beispiel eine Synthese, durchzuführen. Hierfür definiert Yosys spezielle Befehle, die wiederum mehrere andere Befehle, mit dem Ziel, die Synthese durchzuführen, ausführen. Die eigentliche Logik der Synthese ist in den einzelnen Befehlen von Yosys enthalten. Um die Synthese vollständig nachzuvollziehen, müssen diese Befehle analysiert werden.

3.1.2 Erweiterbarkeit

Die im Unterkapitel 3.1.1 angesprochenen Befehle werden in Yosys als Passes bezeichnet. Ein Pass stellt einen in der Shell oder in einem Skript ausführbaren Befehl dar. Die allgemeine Definition eines Passes findet in der Datei `kernel/register.h` statt. Das Listing 3.10 zeigt die vom Pass gespeicherten Variablen. `pass_name` speichert den Namen des Passes und `short_help` dessen Kurzbeschreibung. `next_queued_pass` ist für die Erweiterbarkeit mit Plugins wichtig. Die Variablen in Zeilen 37 bis 39 werden für die interne Funktionsweise von Yosys benötigt.

```
27 struct Pass
28 {
29   std::string pass_name, short_help;
37   int call_counter;
38   int64_t runtime_ns;
39   bool experimental_flag = false;
66   Pass *next_queued_pass;
73 };
```

Listing 3.10: Die Variablen eines Passes[8]

Um einen ausführbaren Pass zu erzeugen, muss ein Typ definiert werden, der die Definition in `kernel/register.h` erbt. Dafür wurden einige Methoden als `virtual` definiert. Dadurch können die Methoden im Listing 3.11 überschrieben werden[2]. `help` gibt die Hilfeausgabe für den jeweiligen Pass aus. Die Methode `execute` führt den Pass aus. Die restlichen Methoden sind Callback-Funktionen, die zu bestimmten Zeitpunkten der Programmausführung aufgerufen werden.

Bevor die Pässe ausgeführt werden können, müssen sie registriert werden. Dafür ist der Konstruktor des Passes zuständig. Die Implementierung des Konstruktors zeigt das Listing 3.12. Der Name und die Kurzbeschreibung werden in den jeweiligen Variablen gespeichert. Im `next_queued_pass`

[2] Methodenüberschreibung bedeutet die Erstellung mehrerer Methodendefinitionen mit dem gleichen Funktionskopf. Die richtige Definition wird dann anhand des Typs, in dem die Methode definiert wurde gewählt.

```
27 struct Pass
28 {
29   std::string pass_name, short_help;
33   virtual void help();
34   virtual void clear_flags();
35   virtual void execute(std::vector<std::string> args, RTLIL::Design *design)
       = 0;
67   virtual void run_register();
71   virtual void on_register();
72   virtual void on_shutdown();
73 };
```

Listing 3.11: Die virtuelen Methoden des Passes[8]

wird der Zeiger aus `first_queued_pass` gespeichert. Im `first_queued_pass` wird der Zeiger auf den aktuellen Pass gesetzt. Diese Variablen werden in der `init_register` Methode genutzt, um die Pässe zu registrieren. Die Methode wird in der Funktion `yosys_setup` aus dem Listing 3.2 aufgerufen. Die restlichen Variablen werden auf Standardwerte gesetzt.

```
92  Pass *first_queued_pass;
101 Pass::Pass(std::string name, std::string short_help) : pass_name(name),
      short_help(short_help)
102 {
103   next_queued_pass = first_queued_pass;
104   first_queued_pass = this;
105   call_counter = 0;
106   runtime_ns = 0;
107 }
```

Listing 3.12: Der Konstruktor des Passes[7]

Ein Pass gilt als registriert, wenn er in der `pass_register` Map gespeichert wurde. Dafür sorgt die Methode `run_register`. Aufgerufen wird sie von der `init_register` Methode. Dort wird die Methode `run_register` des Objekts, das in der Variable `first_queued_pass` gespeichert ist, aufgerufen und der Pass im `next_queued_pass` wird in `first_queued_pass` geschrieben. Allgemein handelt es sich um eine Art einer Warteschlange-Datenstruktur, die vom Pass-Konstruktor aufgebaut wird. Auf den hinzugefügten Pässen wird anschließend die Funktion `on_register` ausgeführt. Den kompletten Vorgang zeigt das Listing 3.13.

Ungeklärt bleibt, wann die Konstruktoren der Pässe aufgerufen werden, um die Warteschlange aufzubauen. Dafür wird für jeden Pass eine Variable vom Typ des Passes definiert. Variablen werden noch vor dem Start des Programms auf ihre Standardwerte gesetzt. Für ein Objekt des Typs einer Klasse entspricht dies dem Wert, der vom Standardkonstruktor erzeugt wird. Das Listing 3.14 zeigt die Art und Weise, wie die Pässe in Yosys definiert werden. In der Zeile 133 wird am Ende der Definition ein Variablenname angegeben. Der Typ der Variable ist der direkt davor definierte `struct`. Damit die Registrierung erfolgreich ist, ruft der neudefinierte Standardkonstruktor den in der Pass-Klasse definierten Konstruktor mit dem Namen und der Kurzbeschreibung des Passes auf. Innerhalb der Definition des `struct`s können Methoden überschrieben werden und weitere Hilfsfunktionen und Variablen definiert werden.

```
92  Pass *first_queued_pass;
96  std::map<std::string, Pass*> pass_register;
109 void Pass::run_register()
110 {
111   log_assert(pass_register.count(pass_name) == 0);
112   pass_register[pass_name] = this;
113 }
114
115 void Pass::init_register()
116 {
117   vector<Pass*> added_passes;
118   while (first_queued_pass) {
119     added_passes.push_back(first_queued_pass);
120     first_queued_pass->run_register();
121     first_queued_pass = first_queued_pass->next_queued_pass;
122   }
123   for (auto added_pass : added_passes)
124     added_pass->on_register();
125 }
```

Listing 3.13: Die Registrierung der Pässe in `init_register` und `run_register`[7]

```
28  struct ProcPass : public Pass {
29    ProcPass() : Pass("proc", "translate processes to netlists") { }
30    void help() override
31    {
72    }
73    void execute(std::vector<std::string> args, RTLIL::Design *design) override
74    {
132   }
133 } ProcPass;
```

Listing 3.14: Ein Beispiel einer Passdefinition anhand des Proc-Passes[4]

Um Yosys mit weiteren Pässen zu erweitern, können Quelldateien in die `passes`, `frontends`, `backends` und `techlibs` Verzeichnisse hinzugefügt werden. Die Quelldateien eines Plugins müssen dort in einem separaten Verzeichnis gespeichert werden. Dieses Verzeichnis benötigt eine Datei `Makefile.inc`, welche die nötigen Make-Befehle beinhaltet, um den Plugin mit Yosys zusammenzukompilieren. Auf diese Art und Weise wird Yosys direkt mit der Unterstützung für den entsprechenden Plugin kompiliert. Die in dem Plugin definierten Befehle sind genauso nutzbar, wie die nativen Befehle von Yosys.

Es besteht auch die Möglichkeit ein Plugin dynamisch zu laden. Dies geschieht mit der `-m` Option in Yosys. Das Listing 3.3 zeigt den Aufruf der `load_plugin` Funktion, die für das dynamische Laden eines Plugins zuständig ist. Der Inhalt der Funktion zeigt das Listing 3.15. In der Zeile 77 wird versucht eine dynamische Bibliothek, die den Plugin beinhalten soll, zu laden. Direkt nach dem Laden der Bibliothek werden alle in ihr definierte Variablen auf Standardwerte gesetzt und somit auch jegliche Passdefinitionen, die korrekt aufgebaut sind. In Zeile 83 wird die `init_register` Funktion aufgerufen, um die potenziell neugeladene Pässe zu registrieren.

Yosys bietet für das Kompilieren eines Plugins ein Bash-Skript an. Das `yosys-config` Skript wird vom Makefile erzeugt. Die Inhalte basieren auf den Kompiliereroptionen mit denen Yosys kompi-

```
34 std::map<std::string, void*> loaded_plugins;
38 std::map<std::string, std::string> loaded_plugin_aliases;
41 void load_plugin(std::string filename, std::vector<std::string> aliases)
42 {
43   std::string orig_filename = filename;
44
45   if (filename.find('/') == std::string::npos)
46     filename = "./" + filename;
51   if (!loaded_plugins.count(filename)) {
77     void *hdl = dlopen(filename.c_str(), RTLD_LAZY|RTLD_LOCAL);
78     if (hdl == NULL && orig_filename.find('/') == std::string::npos)
79       hdl = dlopen((proc_share_dirname() + "plugins/" + orig_filename + ".so"
         ).c_str(), RTLD_LAZY|RTLD_LOCAL);
80     if (hdl == NULL)
81       log_cmd_error("Can't load module `%s': %s\n", filename.c_str(), dlerror
         ());
82     loaded_plugins[orig_filename] = hdl;
83     Pass::init_register();
88   }
89
90   for (auto &alias : aliases)
91     loaded_plugin_aliases[alias] = orig_filename;
92 }
```

Listing 3.15: Die `load_plugin` Funktion[3]

liert werden würde. Auf diese Art und Weise lassen sich die Plugins mit den passenden Optionen kompilieren. Ein so kompiliertes Plugin kann in Yosys mit der `-m` Option geladen werden. Diese Vorgehensweise versichert die Kompatibilität mit der eigenen Yosys Installation. Der Kompilierbefehl lautet:

```
yosys-config --build <Outputdatei> <Quelldateien>...
```

Yosys definiert auch spezialisierte Passarten. Außer der Pass-Klasse gibt es weitere Skript-, Frontend- und Backend-Pass-Klassen, die von der Pass-Klasse erben. Ein Skript ist dabei ein Pass, dessen Hauptaufgabe die Ausführung von anderen Pässen ist. Dabei soll in einem Skriptpass keine nicht triviale Logik implementiert werden. Diese soll in den aufgerufenen Pässen stattfinden. Der Skriptpass dient lediglich zur Festsetzung der Reihenfolge, in der die Unterpässe ausgeführt werden[14, S. 60].

Ein Frontend unterscheidet sich von einem normalen Pass in der Definition der `execute` Funktion. Zusätzlich zu den Argumenten und dem Design wird ein *Input-Stream* und ein Dateiname übergeben.

```
virtual void execute(std::vector<std::string> args, RTLIL::Design *design)
    = 0;
```

wird zu

```
virtual void execute(std::istream *&f, std::string filename, std::vector<
    std::string> args, RTLIL::Design *design) = 0;
```

Bei einem Backend wird die Definition zu

```
virtual void execute(std::ostream *&f, std::string filename, std::vector<
    std::string> args, RTLIL::Design *design) = 0;
```

Statt dem *Input-Stream* wird dort ein *Output-Stream* übergeben. Für Frontends und Backends gibt es weitere Register, die während des `init_register`-Aufrufs registriert werden müssen. Dafür werden die `run_register` Funktionen überschrieben. Diese Register lauten:

```
std::map<std::string, Frontend*> frontend_register;
std::map<std::string, Backend*> backend_register;
```

3.2 RTLIL

Register Transfer Level Intermediate Language (RTLIL) ist der Name der internen Datenstruktur in Yosys, die für die Darstellung eines eingelesenen Designs in allen Phasen der Synthese zuständig ist. Die Datenstruktur kann alle synthetisierbaren Verilog-Konstrukte wie Prozesse und Speicher(Arrays in Verilog) abbilden. Es existiert ein textbasiertes Format mit dem die RTLIL-Datenstruktur serialisiert werden kann. Yosys kann eine Datei im RTLIL-Format einlesen. Dafür existiert das RTLIL-Frontend. Im Folgenden wird zunächst die Syntax einer textuellen RTLIL-Beschreibung untersucht. Darauf basierend werden die Konstrukte des RTLIL-Formats mit internen Datenstrukturen in Zusammenhang gebracht und somit die Implementierung von RTLIL in Yosys analysiert. Zum Schluss wird der Vorgang des Einlesens des RTLIL-Formats im RTLIL-Frontend untersucht.

3.2.1 Austauschformat

Das RTLIL-Format ist im Appendix D des Yosys-Manuals beschrieben. Allgemein entspricht das Austauschformat einer Auflistung aller Modulen des Designs und ihren Inhalten. Am Anfang der Datei kann ein `autoidx`-Statement stehen. Diese Anweisung speichert den Zustand einer globalen Variable `Yosys::autoidx`, die für die Nummerierung der RTLIL-Identifikatoren genutzt wird. Die Nummerierung wird für die von Yosys erstellten Objekte des RTLIL-Formats genutzt. Jedes von Yosys erstellte Objekt bekommt eine fortlaufende Nummer, die im Identifikator eingebettet ist. Somit werden Namenskollisionen für den Fall vermieden, dass Yosys zwei Objekte mit gleichem Namen erstellen sollte. Die Syntax des Statements hat folgendes Aussehen: `autoidx 1`[14, S. 258]. Daraufhin folgt die Auflistung der Module.

Sowohl Module als auch die einzelnen Inhalte der Module besitzen einen Identifikator. Es wird zwischen zwei Arten von Identifikatoren unterschieden[14, S. 256]. Die *publically visible identifiers* sind Identifikatoren, die aus den ursprünglichen Namen des Verilog-Entwurfs abgeleitet, oder vom Benutzer selbst angegeben wurden. Die *auto-generated identifiers* werden während des Programmablaufs von Yosys erstellt. Die beiden Arten werden anhand des ersten Zeichens unterschieden. Bei einem *publically visible Identifier* wird ein backslash(\) und bei einem *auto-generated Identifier* ein Dollarzeichen($) verwendet. Es kommt vor, dass Yosys während der Vereinfachung des Designs einzelne Signale, die nicht mehr benötigt werden, löscht. In diesen Fällen werden die *auto-generated identifiers* gelöscht, während die *publically visible identifiers* beibehalten werden. Auf diese Weise werden auch Namenskollisionen zwischen benutzerdefinierten und den von Yosys generierten Identifikatoren vermieden.[14, S. 33-34]

Module und ihre Inhalte können Attribute annehmen. In der textuellen Repräsentation stehen Attribute immer unmittelbar vor dem Objekt, das sie beschreiben sollen. Ein Attribut besteht aus einem Identifikator und einer Konstanten. Eine Konstante kann einen Wert, eine Ganzzahl oder eine Zeichenkette darstellen. Ein Wert besteht aus einer Angabe der Bitbreite als Dezimalzahl und den einzelnen Bits[14, S. 257]:

- 0: Logische Null
- 1: Logische Eins
- x: Unbekannt/Undefiniert/Don't care
- z: Hochohmig/Don't care
- m: Markiert(Wird in Yosys intern genutzt)
- -: Don't care

Ein beispielhafter Wert ist: `6'01xzm-`. Eine Ganzzahl muss im Wertebereich $[-2147483648, 2147483648)$ liegen und eine Zeichenkette ist eine Ansammlung von beliebigen Zeichen, eingeschlossen in Doppelhochkommas. Das Attribut `attribute \src "test.v:28.1-33.10"` gibt zum Beispiel an, wo im Quelltext die Definition des Objekts stattfindet.

Ein Modul bekommt einen Identifikator und besteht aus einer beliebigen Folge an Parametern, Signalen, Speichern, Zellen, Prozessen und Verbindungen. Die Syntax für die Definition eines Moduls mit Attributen zeigt das Listing 3.16[14, S. 258]. Zuerst werden die Attribute des Moduls genannt. Danach wird mit der `module`-Anweisung das Modul benannt. Zwischen der Anweisung und dem Schlüsselwort `end` können die obengenannten Inhalte eines Moduls vorkommen.

```
1 attribute \cells_not_processed 1
2 attribute \src "test.v:28.1-33.10"
3 module \test
4   # Inhalte des Moduls
5 end
```

Listing 3.16: Syntax für die Definition eines Moduls

Ein Parameter wird innerhalb des Moduls definiert, und besteht aus einem Identifikator und einer optionalen Konstanten, die als Standardwert des Parameters dient(`parameter \param 32222`). Ein Signal wird mit dem Schlüsselwort `wire` definiert. Für ein simples Signal muss nur ein Identifikator angegeben werden. Zwischen dem Schlüsselwort und dem Identifikator können zusätzliche Optionen für ein Signal angegeben werden. Mit `width` wird die Breite des Signals festgelegt. So lässt sich ein Bus definieren. Die Option `offset` gibt an, welchen Index das erste Bit des Signals haben soll. Ein Signal kann ein Port eines Moduls sein. In diesem Fall können die Optionen `input`, `output` und `inout` angegeben werden. Diese Optionen setzen die Richtung und den Index des Signalports. Mit der Option `upto` wird spezifiziert, ob das erste Bit des Busses ein Most Significant Bit/Byte (MSB) oder ein Least Significant Bit/Byte (LSB) ist. Zuletzt gibt die Option `signed` an, dass der Wert des Signals als vorzeichenbehaftet verarbeitet werden soll. Die Optionen können miteinander kombiniert werden. Das Listing 3.17 zeigt die verschiedene Definitionsmöglichkeiten[14, S. 259].

```
wire \w1                  # Simples Signal mit der Name \w1
wire width 5 \w2          # Das Signal \w2 ist ein Bus mit Breite 5
wire offset 3 \w3         # Das Bit des Signals würde das Index 3 besitzen
wire input 1 \w4          # Das Signal \w4 ist ein input
wire output 2 \w5         # Das Signal \w5 ist ein output
wire inout 3 \w6          # Das Signal \w6 ist ein input und output
wire output 1 \w7         # Das Signal \w7 ist ein output
wire width 5 upto \w8     # Das Signal ist ein Bus mit dem MSB an letzter Stelle
wire width 5 signed \w9 # Der Bus wird als ein zeichenbehafteter Wert behandelt
```

Listing 3.17: Syntax für die Signaldefinitionen in einem Modul

Eine Zelle beschreibt eine Instanz eines Moduls und ist Teil des umliegenden Moduls. Für die Definition einer Zelle ist ein Zellentyp und ein Identifikator nötig. Ein Zellentyp ist dabei entweder eine vordefinierte Zelle oder ein bereits bekanntes Modul. Innerhalb einer Zelle werden Parameterwerte festgelegt und Signale mit den Ports verbunden[14, S. 260]. Das Listing 3.18 zeigt die Instanziierung einer Zelle. Im Beispiel existiert ein Modul `\test_module`. Die Zelle ist eine Instanziierung des Moduls und heißt `\test_module_inst`. Das Modul definiert einen Parameter `\param1` dem hier der Wert acht zugewiesen wird. Dem Port `\port1` wird ein Signal `\some_wire` zugewiesen.

```
cell \test_module \test_module_inst
  parameter \param1 8
  connect \port1 \some_wire
end
```

Listing 3.18: Syntax für die Zelleninstanziierung in einem Modul

Die rechte Seite der Zuweisung in der Zelleninstanziierung wird von Yosys `sigspec` genannt[14, S. 260]. Ein `sigspec` kann eine Konstante, ein Signal, ein Teil eines anderen `sigspecs` oder eine Verkettung von mehreren anderen `sigspecs` sein[14, S. 259]. Die linke Seite der Zuweisung ist ein Identifikator eines Signals des Untermoduls, das als ein Port definiert wurde. `sigspecs` werden auch außerhalb von Zelleninstanziierungen für die Verbindung von Signalen genutzt. Im Modul lassen sich Signale mit Hilfe der `connect`-Anweisung verbinden. Eine solche Anweisung besteht aus einem Paar von `sigspecs`. Das Listing 3.19 zeigt die Syntax der Anweisung und der `sigspecs`.

Ein weiterer Bestandteil eines Moduls sind Speicher(`memory`). Sie können eine Breite, Größe und ein Offset besitzen. In der Regel werden Verilog Arrays mit Hilfe von Speichern in RTLIL dargestellt. Die Syntax für einen Speicher ist: `memory width 8 size 128 offset 0 \mem1`. Die Optionen des Speichers sind optional.

Ein Modul kann auch Prozesse(`process`) beinhalten. Ein Prozess erhält einen Identifikator. Die Inhalte befinden sich ähnlich wie bei einer Zelleninstanziierung innerhalb der Definition. Auf der obersten Ebene eines Prozesses befinden sich Zuweisungen und die sogenannten `switch`-Anweisungen. Als Letztes werden die Synchronisationsregeln(`sync`) angegeben. Eine Zuweisung im Prozess wird mit dem Schlüsselwort `assign` angegeben und benötigt wie bei der `connect`-Anweisung zwei `sigspecs`. Der Unterschied ist, dass hier die Reihenfolge der `sigspecs` wichtig ist. In das erste `sigspec` wird der Wert des zweiten `sigspecs` geschrieben[14, S. 260]. Damit sollen RTLIL-Prozesse `always`-Blöcke widerspiegeln.

Eine `switch`-Anweisung stellt ein Entscheidungsbaum dar. Die `switch`-Anweisung vergleicht den Wert eines Signals mit den Werten anderer Signale. Die erste Übereinstimmung entscheidet darüber

```
wire width 8 \w1
wire width 8 \w2
wire width 8 \w3
wire width 8 \w4
wire width 12 \w5
wire width 4 \w6

connect \w1 8'01100001          # \w1 erhält den Wert 1100001(97)
connect \w2 97                  # \w2 erhält den Wert 1100001(97)
connect \w3 "a"                 # \w3 erhält den Wert 1100001(97)(a in ASCII)
connect \w4 \w3                 # \w4 wird mit \w3 verbunden
connect \w5 [3:0] \w4 [7:4]     # 4 Bits von \w5 werden mit 4 Bits von \w4 verbunden
# 8 Bits von \w5 werden mit einer 11, dem \w6 und 2 Bits von \w4 verbunden.
connect \w5 [11:4] { 2'11 \w6 \w4 [1:0] }
```

Listing 3.19: Syntax für die `connect`-Anweisung und verschiedene `sigspec`-Arten

welcher Fall(`case`) weiter betrachtet wird. Die beiden Signale werden als `sigspecs` gelistet. Innerhalb einer `switch`-Anweisung werden also `case`-Anweisungen angegeben. Eine `case`-Anweisung beinhaltet ein `sigspec`, mit dem das `switch`-Signal verglichen werden soll. Innerhalb der `case`-Anweisung können sich weitere `switch`-Anweisungen und Zuweisungen(`assign`) befinden. Damit wird ein Entscheidungsbaum aufgebaut.

```
process \proc1
  assign \w5 \w9
  switch \w6
    case 1'1
      switch \w7
        case 1'1
          assign \w10 1'0
        case 1'0
          assign \w10 1'1
      end
      assign \w5 1'0
    case 1'0
      switch \w8
        case 2'00
          assign \w10 1'1
        case 2'01
          assign \w10 1'0
        case
          assign \w10 \w5
      end
      assign \w5 1'1
  end
  assign \w4 \w5
  assign \w2 \w10
  sync posedge \clk
  update \w1 \w2
  update \w3 \w4
end
```

Listing 3.20: Syntax eines Prozesses

Die Sensitivitätsliste der Verilog always-Blöcke wird in RTLIL mit sync-Anweisungen am Ende des Prozesses implementiert. Eine sync-Anweisung benötigt die Angabe des Typs der Synchronisierung und gegebenenfalls ein sigspec. Der sigspec stellt das Signal dar, dessen Änderung die Synchronisationsregel auslösen kann. Die Aktionen, die stattfinden sollen, wenn eine Regel ausgelöst wird, werden mit der update-Anweisung angegeben. Die update-Anweisung ist wie die assign-Anweisung aufgebaut. Auch hier werden zwei sigspecs angegeben, wovon auf das eine sigspec geschrieben und vom anderen sigspec gelesen wird.

Die Synchronisationstypen low, high, posedge, negedge und edge benötigen die Angabe eines sigspecs. Bei low und high werden die update-Aktionen so lange aktiv, solange der Wert des sigspecs den jeweiligen Pegel hat. Die anderen Typen werden an den Flanken der sigspecs einmalig ausgelöst. Beim edge-Typ wird die Aktion sowohl bei der steigenden als auch bei der fallenden Flanke ausgeführt. Der init-Typ gibt an, dass die Aktion direkt beim Start des Designs ausgeführt werden soll. always bedeutet, dass die Aktionen immer gelten. In diesem Fall verhalten sich die update-Anweisungen ähnlich wie die connect-Anweisungen außerhalb eines Prozesses. global gibt an, dass der globale Takt des Designs genutzt werden soll.

Das Listing 3.20 zeigt einen beispielhaften Prozess. In Zeile 2 wird der Wert von \w9 in \w5 gespeichert. Der switch in der Zeile 3 entscheidet anhand von \w6, welcher Wert endgültig \w5 zugewiesen wird. Die Zuweisungen befinden sich in den Zeilen 11 und 21. Darüber hinaus wird hier entschieden, welcher switch den Wert von \w10 bestimmen soll. Die case-Anweisung in Zeile 18 besitzt keinen sigspec. Dies ist der Standardfall, der ausgeführt wird, wenn keiner der vorherigen Fälle zugetroffen ist. Am Ende in den Zeilen 23-24 werden die für \w5 und \w10 ermittelten Werte für die Zuweisungen von \w2 und \w4 genutzt. Die Synchronisationsregel in Zeile 25 wird bei der steigenden Flanke des Signals \clk, das den Taktgeber des Designs darstellen soll, ausgelöst. Bei der steigenden Flanke werden die Werte in \w2 und \w4 in \w1 und \w2 übernommen.

3.2.2 Interne Datenstrukturen

In Yosys wird ein RTLIL-Design durch Datenstrukturen aus dem Yosys::RTLIL namespace dargestellt. Ein gesamtes Design wird durch ein RTLIL::Design Objekt abgebildet. Im Wesentlichen hält das Design die zugehörigen Module in einer Map(modules_), in der ihre Identifikatoren als Schlüssel dienen. Das in der Hierarchie oberste Modul wird mit einem Attribut \top des Moduls markiert.

Attribute werden durch RTLIL::AttrObject-Objekte umgesetzt. Ein AttrObject speichert eine Map(attributes) zwischen Identifikatoren und RTLIL::Const Objekten. Das AttrObject implementiert eine Reihe an Hilfsfunktionen, die beim Umgang mit Attributen helfen sollen. Die Funktionalität der Attribute wird anderen Typen durch Vererbung weitergegeben. Die Typen, die Attribute annehmen sollen, erben von AttrObject. Das Listing 3.21 zeigt die Definition.

Ein Identifikator in Yosys wird mit dem Typ RTLIL::IdString dargestellt. Ein IdString bildet einen Identifikator auf eine ganze Zahl ab. Damit bleibt ein IdString Objekt leicht und kann im Quelltext

```
695 struct RTLIL::AttrObject
696 {
697   dict<RTLIL::IdString, RTLIL::Const> attributes;
727 };
```

Listing 3.21: Definition von Yosys::RTLIL::AttrObject[9]

einfach kopiert und übergeben werden. Die textuelle Darstellung eines Identifikators wird in einem statisch definierten Vektor abgespeichert. Die Zahl, die in einem IdString Objekt gespeichert wird, ist die Position der textuellen Darstellung im Vektor. Zusätzlich wird in einer ebenfalls statischen Map die Abbildung der textuellen Darstellung auf eine Zahl gespeichert. Damit kann die Umwandlung der Zahl in die textuelle Darstellung und umgekehrt schnell erfolgen. Letztendlich ist ein IdString eine Zeichenkette, die im Programm wie eine ganze Zahl behandelt werden kann. Das Listing 3.22 zeigt die Definition von RTLIL::IdString.

```
79   struct IdString
80   {
94     static std::vector<char*> global_id_storage_;
95     static dict<char*, int, hash_cstr_ops> global_id_index_;
263    int index_;
386    bool isPublic() const { return begins_with("\\"); }
```

Listing 3.22: Definition von Yosys::RTLIL::IdString[9]

Ein RTLIL::Const Objekt speichert einen Vektor aus RTLIL::State Objekten in der Variable bits ab. Im einfachsten Fall stellt ein Const Objekt eine Bitfolge dar. Es ist auch möglich mit dem Const Objekt eine Zeichenkette oder eine ganze Zahl zu speichern. Bei einer ganzen Zahl werden die Bits, die für die Darstellung der Zahl benötigt werden, im Vektor gespeichert. Im Fall einer Zeichenkette werden die Zeichen auf Bits abgebildet, die dem ASCII-Wert des Zeichens entsprechen. Diese Bits werden dann im Vektor gespeichert. Ein Const Objekt findet überall da Anwendung, wo im RTLIL-Format eine Zeichenkette, Zahl oder Bitfolge angegeben werden kann. Dies ist zum Beispiel bei einem Parameter der Fall. Ein RTLIL::State Objekt ist eine einfache Definition der möglichen Werte eines Bits. Das Listing 3.23 zeigt diese Definition. S0 und S1 stellen die zwei Bitzustände Null und Eins dar. Sx bedeutet ein undefiniertes Bit. Sz wird für die Darstellung von hochohmigen Zuständen eines Signals genutzt. Sa stellt ein don't care Bit dar, das zum Beispiel bei einem Multiplexer nicht beachtet werden soll. Sm wird während der Synthese intern benutzt, um Bits zu markieren. Die Bedeutung der Markierung hängt von der konkreten Stelle im Programm ab.

```
29   enum State : unsigned char {
30     S0 = 0,
31     S1 = 1,
32     Sx = 2, // undefined value or conflict
33     Sz = 3, // high-impedance / not-connected
34     Sa = 4, // don't care (used only in cases)
35     Sm = 5  // marker (used internally by some passes)
36   };
```

Listing 3.23: Definition von Yosys::RTLIL::State[9]

Zusätzlich zu den Modulen speichert das RTLIL::Design weitere Daten. In Yosys können Teile des Designs ausgewählt werden. Dadurch kann in bestimmten Zeitpunkten der Synthese die Verarbeitung auf einen Teil des Designs begrenzt werden. Das Design ist für die Verwaltung der ausgewählten Module zuständig. Die Auswahl wird in der Variable selection_stack gespeichert. Die aktuelle Auswahl ist immer die oberste im Stack. Zusätzlich kann ein komplettes Modul mit der selected_active_module Variable gewählt werden. Diese Auswahl überschreibt die des Stacks. Damit ist nur das gegebene Modul gewählt und keine Untermodule, die darin als Zelleninstanziierungen angegeben wurden. Mit selection_vars kann die Auswahl mit einem Namen in einer Map

zwischengespeichert werden. Eine Auswahl wird mit dem RTLIL::Selection Typ umgesetzt. Das Listing 3.24 zeigt die Member-Variablen der Selection. Die Variable full_selection gibt an, ob das gesamte Design gewählt ist. Mit selected_modules können gesamte Module gewählt werden. Im Gegensatz zu selected_active_module werden bei dieser Auswahl auch die Untermodule betrachtet. Mit selected_members lassen sich einzelne Inhalte (Zellen, Signale, Prozesse, ...) eines Moduls auswählen. selected_members ist eine Map, die den Namen des Moduls und eine Menge der Namen der Inhalte speichert.

```
980  struct RTLIL::Selection
981  {
982    bool full_selection;
983    pool<RTLIL::IdString> selected_modules;
984    dict<RTLIL::IdString, pool<RTLIL::IdString>> selected_members;
1008 };
```

Listing 3.24: Definition von Membern von Yosys::RTLIL::Selection[9]

Yosys bietet die Möglichkeit, beliebige Daten im Design-Objekt abzuspeichern. Damit ist ein simpler Datenaustausch zwischen verschiedenen Pässen realisierbar. Die Daten werden in der scratchpad-Map des Design-Objekts gespeichert. Die Map besteht aus Zeichenkettenpaaren. Eine Zeichenkette ist der Schlüssel, während die zweite als der eigentliche Datenspeicher selbst dient. Das Listing 3.25 zeigt die Definition des Scratchpads im Design. Im Design sind Methoden definiert, mit denen Zahlen, boolesche Werte und Zeichenketten geschrieben und gelesen werden können.

```
1033 struct RTLIL::Design
1034 {
1039   dict<std::string, std::string> scratchpad;
1133 };
```

Listing 3.25: Definition des Scratchpads im Yosys::RTLIL::Design[9]

Das Design speichert auch die gesetzten Verilog-Defines in der Map verilog_defines ab. Die Defines können aus der auf Seite 17 beschriebenen Logik und aus dem Verilog-Frontend gesetzt werden.

Darüber hinaus definiert das Design weitere Variablen, die für interne Zwecke in Yosys verwendet werden, aber nicht weiter für die Synthese von Bedeutung sind. Es existieren auch weitere Methoden, die für das Hinzufügen und Löschen von Modulen zuständig sind.

Ein RTLIL::Module erbt von RTLIL::AttrObject und ist somit fähig, Attribute anzunehmen. Das Module muss in der Lage sein, die im Unterkapitel 3.2.1 beschriebene Konstrukte zu speichern. Dazu gehören Parameter, Signale, Speicher, Zellen, Prozesse und die jeweiligen Verbindungen zwischen diesen. Das Listing 3.26 zeigt die Variablen der Module-Klasse, die für die Speicherung der obengenannten Konstrukte zuständig sind.

Signale (wire), Zellen (cell), Speicher (memory) und Prozesse (process) werden in Maps gespeichert, in welchen die jeweiligen Identifikatoren der RTLIL-Objekte als Schlüssel fungieren. Für die Parameter existieren zwei Variablen. avail_parameters enthält alle mögliche Parameter des Moduls. In parameter_default_values werden nur die Parameter gespeichert, die auch einen Standardwert besitzen. Der ports-Vektor speichert eine Liste der Signale, die als Ein- und Ausgänge des Moduls definiert sind. Verbindungen zwischen den Objekten werden mit Hilfe des connections_

```
1135 struct RTLIL::Module : public RTLIL::AttrObject
1136 {
1152   dict<RTLIL::IdString, RTLIL::Wire*> wires_;
1153   dict<RTLIL::IdString, RTLIL::Cell*> cells_;
1154
1155   std::vector<RTLIL::SigSig>   connections_;
1159   idict<RTLIL::IdString> avail_parameters;
1160   dict<RTLIL::IdString, RTLIL::Const> parameter_default_values;
1161   dict<RTLIL::IdString, RTLIL::Memory*> memories;
1162   dict<RTLIL::IdString, RTLIL::Process*> processes;
1182   std::vector<RTLIL::IdString> ports;
1458 };
```

Listing 3.26: Variablen eines Yosys::RTLIL::Module-Objekts[9]

Vektors umgesetzt. Dieser speichert RTLIL::SigSig-Objekte, die im Quelltext als einfache Paare aus RTLIL::SigSpec-Objekten definiert sind.

Ein RTLIL::SigSpec implementiert ein auf der Seite 25 bereits eingeführtes sigspec. Das SigSpec bietet die Möglichkeit aus definierten wires beliebige Zusammensetzungen von Signalen darzustellen. Die Implementierung basiert auf den zwei Typen RTLIL::SigChunk und RTLIL::SigBit. Das Listing 3.27 zeigt die Definition des SigSpec-Typs. Ein SigChunk ist in der Lage, mehrere Bits eines Signals darzustellen, während ein SigBit nur einem einzelnen Bit eines Signals entspricht. Ein SigSpec besitzt zwei stabile Zustände: packed und unpacked. Packed bedeutet, dass der gesamte SigSpec nur durch SigChunk-Objekte beschrieben ist. Dagegen bedeutet der unpacked Zustand, dass der SigSpec nur aus SigBit-Objekten besteht.

```
804 struct RTLIL::SigSpec
805 {
809   std::vector<RTLIL::SigChunk> chunks_; // LSB at index 0
810   std::vector<RTLIL::SigBit> bits_; // LSB at index 0
978 };
```

Listing 3.27: Variablen eines Yosys::RTLIL::SigSpec-Objekts[9]

Das SigSpec definiert Methoden, welche die einfache Umwandlung zwischen den beiden obengenannten Zuständen ermöglichen. Weitere Methoden dienen zur Veränderung eines SigSpec-Objekts. Yosys definiert Eigenschaften, die ein SigSpec besitzen kann. Die Eigenschaften sind in der Tabelle 3.3 gelistet.

Die Typen RTLIL::SigChunk und RTLIL::SigBit sind ähnlich aufgebaut. Beide besitzen einen Zeiger auf RTLIL::Wire und ein offset. Der offset gibt an, um welches Bit des wires es sich handelt. Ein RTLIL::SigChunk besitzt zusätzlich eine Breite und kann damit mehrere Bits eines wires gleichzeitig referenzieren. Hier gibt der Offset, die Position des ersten Bits im wire an. Im Fall, dass die RTLIL::Wire-Zeiger Null sind, bedeutet dies, dass das SigBit oder SigChunk einen konstanten Wert darstellt. Ein konstanter Wert wird mit Hilfe des RTLIL::State-Enums gespeichert. Bei dem SigChunk werden die Bits in einem Vektor gespeichert. Das Listing 3.28 zeigt die beiden Definitionen. Durch die Nutzung von SigChunk- und SigBit-Objekten im SigSpec ist es möglich, einzelne Bits mehrerer Signale und konstante Bits zusammenzuführen, was den Möglichkeiten der sigspecs des Austauschformats entspricht.

Eigenschaft	Bedeutung
is_wire	Das SigSpec referenziert einen wire mit Breite Eins
is_chunk	Das SigSpec besteht aus nur einem SigChunk
is_bit	Das SigSpec hat die Breite Eins
is_fully_const	Das SigSpec referenziert keinen wire und besteht nur aus Konstanten
is_fully_zero is_fully_ones	Das SigSpec besteht jeweils nur aus Nullen oder Einsen(Setzt is_fully_const voraus)
is_fully_def is_fully_undef	Das SigSpec besteht jeweils nur aus Nullen und Einsen oder nur aus undefinierten Bits (Setzt is_fully_const voraus)
has_const	Das SigSpec ist zum Teil konstant definiert
has_marked_bits	Das SigSpec beinhaltet markierte Bits (Setzt has_const voraus)

Tabelle 3.3: Eigenschaften eines SigSpec-Objekts

```
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;
753 };
754
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   };
780 };
```

Listing 3.28: Yosys::RTLIL::SigBit- und Yosys::RTLIL::SigChunk-Definitionen[9]

Der RTLIL::Wire-Typ ist für die Darstellung eines wire zuständig. Das Listing 3.29 zeigt die Definition eines Wires. Entscheidend sind hier die Variablen in Zeilen 1478 und 1479. Diese Variablen speichern die Optionen des Austauschformats, die auf Seite 24 angesprochen worden sind. Die Variable width legt die Breite des Signals fest und mit start_offset wird die offset Funktion implementiert, die den Index des ersten Bits des Signals angibt. Ein wire kann auch als Port dienen. Dafür speichert die Variable port_id die Nummer des Ports. port_input und port_output legen für einen Port fest, ob es sich um einen Eingang, Ausgang oder beides handelt. Die Variable upto spezifiziert, ob die Bits des Signals in umgekehrter Reihenfolge verarbeitet werden sollen. Mit is_signed wird spezifiziert, dass es sich bei dem Signalwert, um eine vorzeichenbehaftete Zahl handelt.

Wie bereits im Unterkapitel 3.2.1 erklärt, stellt ein RTLIL::Cell-Objekt eine Instanziierung eines Moduls dar. Eine Zelle spezifiziert die Parameterwerte des Moduls und verbindet seine Ports mit den Signalen des umliegenden Moduls. Dafür werden zwei Maps definiert. connections_ verbindet die

```
1460 struct RTLIL::Wire : public RTLIL::AttrObject
1461 {
1476   RTLIL::Module *module;
1477   RTLIL::IdString name;
1478   int width, start_offset, port_id;
1479   bool port_input, port_output, upto, is_signed;
1484 };
```

Listing 3.29: Definition von Yosys::RTLIL::Wire[9]

Ports der Zelle mit anderen Signalen mit Hilfe eines SigSpec-Objekts. Für die Parameter werden in parameters gegebenenfalls die Standardwerte des Moduls überschrieben. Die Variable type speichert den Namen des Moduls, dessen Instanz die Zelle definiert. Das Listing 3.30 zeigt die Definition von RTLIL::Cell.

```
1501 struct RTLIL::Cell : public RTLIL::AttrObject
1502 {
1517   RTLIL::Module *module;
1518   RTLIL::IdString name;
1519   RTLIL::IdString type;
1520   dict<RTLIL::IdString, RTLIL::SigSpec> connections_;
1521   dict<RTLIL::IdString, RTLIL::Const> parameters;
1559 };
```

Listing 3.30: Definition von Yosys::RTLIL::Cell[9]

Ein RTLIL::Memory-Objekt stellt eine Beschreibung eines Speichers dar. Neben dem Identifikator werden die Breite(width), der Offset(start_offset) und die Größe(size) gespeichert. Die Breite spezifiziert die Anzahl der Bits eines Wortes im Speicher. Ein Offset gibt ähnlich wie bei einem Signal den index an, an dem das erste Wort verfügbar ist. Die Größe spezifiziert die Anzahl der Wörter im Speicher. Das Listing 3.31 zeigt die Definition eines Speichers in RTLIL.

```
1486 struct RTLIL::Memory : public RTLIL::AttrObject
1487 {
1493   RTLIL::IdString name;
1494   int width, start_offset, size;
1499 };
```

Listing 3.31: Definition von Yosys::RTLIL::Memory[9]

Ein RTLIL::Process-Objekt besteht im Wesentlichen aus RTLIL::CaseRule, RTLIL::SwitchRule und RTLIL::SyncRule Objekten. Auf der obersten Ebene wird ein CaseRule-Objekt in der root_case-Variable gespeichert. In einem CaseRule-Objekt können SwitchRule-Objekte enthalten sein, die wiederrum weitere CaseRule-Objekte beinhalten. Damit ist eine beliebig tiefe Verschachtelung der Logik möglich. Die SyncRule-Objekte werden als Vektor in der Variable syncs gespeichert. Dort wird definiert, bei welchen Ereignissen welche Aktionen ausgeführt werden sollen. Listing 3.32 zeigt die Definition des RTLIL::Process.

```
1611 struct RTLIL::Process : public RTLIL::AttrObject
1612 {
1623   RTLIL::IdString name;
1624   RTLIL::Module *module;
1625   RTLIL::CaseRule root_case;
1626   std::vector<RTLIL::SyncRule*> syncs;
1631 };
```

Listing 3.32: Definition von Yosys::RTLIL::Process[9]

Ein RTLIL::SwitchRule-Objekt speichert ein RTLIL::SigSpec-Objekt und einen Vektor aus RTLIL::CaseRule-Objekten. Das SigSpec stellt ein Signal dar, mit dem entschieden wird, welche CaseRule aktiv werden soll. Das Listing 3.33 zeigt die SwitchRule-Definition.

```
1576 struct RTLIL::SwitchRule : public RTLIL::AttrObject
1577 {
1578   RTLIL::SigSpec signal;
1579   std::vector<RTLIL::CaseRule*> cases;
1588 };
```

Listing 3.33: Definition von Yosys::RTLIL::SwitchRule[9]

In einem RTLIL::CaseRule-Objekt wird in der Variable compare ein SigSpec-Objekt gespeichert, welches mit dem SigSpec-Objekt der SwitchRule verglichen wird. Die Übereinstimmung der beiden SigSpec-Objekte bedeutet, dass die CaseRule aktiv wird. Nur dann werden die Verbindungen aus der actions-Variable und die weiteren Verschachtelungen aus der switches-Variable aktiv. Die Besonderheit beim CaseRule-Objekt in der root_case-Variable ist, dass dort die compare-Variable leer bleibt. In allen anderen Fällen kennzeichnet die leere compare-Variable eine CaseRule, die ausgeführt wird, wenn keine der anderen Signale übereinstimmten. Das Listing 3.34 zeigt den Aufbau eines CaseRule-Objekts.

```
1561 struct RTLIL::CaseRule : public RTLIL::AttrObject
1562 {
1563   std::vector<RTLIL::SigSpec> compare;
1564   std::vector<RTLIL::SigSig> actions;
1565   std::vector<RTLIL::SwitchRule*> switches;
1574 };
```

Listing 3.34: Definition von Yosys::RTLIL::CaseRule[9]

Ein RTLIL::SyncRule-Objekt dient zur Festlegung der Aktionen, die bei den Änderungen von Signalen ausgeführt werden sollen. Das Signal, das die Aktionen hervorruft, wird als ein RTLIL::SigSpec-Objekt in der Variable signal gespeichert. In der Variable actions werden Verbindungen festgelegt, die zum Zeitpunkt der Ereignisse aktiv werden sollen. Ein spezieller Fall einer Aktion sind die RTLIL::MemWriteAction-Objekte. Sie abstrahieren die Schnittstelle eines Speichers. Ein RTLIL::SyncType gibt dabei an, bei welchem Ereignis die SyncRule aktiv werden soll. Das Listing 3.35 zeigt die Definition eines RTLIL::SyncRule-Objekts.

Das RTLIL::SyncType-Enum definiert die möglichen Ereignisse, bei denen eine SyncRule aktiv wird. Die pegelsensitive Typen geben an, dass die Verbindungen aus dem SyncRule-Objekt immer aktiv sind, wenn das Signal den gegebenen Pegel aufweist. Weitere Typen sind die flankensensitiven

```
1599 struct RTLIL::SyncRule
1600 {
1601   RTLIL::SyncType type;
1602   RTLIL::SigSpec signal;
1603   std::vector<RTLIL::SigSig> actions;
1604   std::vector<RTLIL::MemWriteAction> mem_write_actions;
1609 };
```

Listing 3.35: Definition von Yosys::RTLIL::SyncRule[9]

Ereignisse. Sie werden zum Zeitpunkt des Pegelwechsels eines Signals aktiv. Dabei wird zwischen positiver und negativer Flanke unterscheiden. Es ist auch möglich, dass eine SyncRule bei beiden Flanken aktiv wird. Die sonstigen Typen geben an, dass die SyncRule immer aktiv, mit einem globalen Takt oder nur am Anfang der Ausführung des Designs aktiv wird. Das Listing 3.36 zeigt die Definition des SyncType.

```
38   enum SyncType : unsigned char {
39     ST0 = 0, // level sensitive: 0
40     ST1 = 1, // level sensitive: 1
41     STp = 2, // edge sensitive: posedge
42     STn = 3, // edge sensitive: negedge
43     STe = 4, // edge sensitive: both edges
44     STa = 5, // always active
45     STg = 6, // global clock
46     STi = 7  // init
47   };
```

Listing 3.36: Definition von Yosys::RTLIL::SyncType[9]

Die RTLIL::MemWriteAction-Objekte spezifizieren den Identifikator des Speichers auf den geschrieben werden soll. Speicher werden letztendlich in Speicherzellen umgewandelt, die selbst bestimmte Ports besitzen. Mit SigSpec-Objekten werden in RTLIL::MemWriteAction-Objekten die Signale gespeichert, die den Adress-, Daten- und Aktivierungsports zugewiesen werden. Die priority_mask legt fest, wie sich die Schaltung verhalten soll, wenn zwei Speicherschreibzugriffe gleichzeitig auf dieselbe Adresse zu schreiben versuchen[14, S. 47]. Das Listing 3.37 zeigt die Definition von RTLIL::MemWriteAction.

```
1590 struct RTLIL::MemWriteAction : RTLIL::AttrObject
1591 {
1592   RTLIL::IdString memid;
1593   RTLIL::SigSpec address;
1594   RTLIL::SigSpec data;
1595   RTLIL::SigSpec enable;
1596   RTLIL::Const priority_mask;
1597 };
```

Listing 3.37: Definition von Yosys::RTLIL::MemWriteAction[9]

3.2.3 RTLIL-Frontend

Das Einlesen eines Designs im RTLIL-Format wird vom RTLIL-Frontend übernommen. Der Aufbau des RTLIL-Frontends ist ähnlich wie der Aufbau des Verilog-Frontends, das im Unterkapitel 2.3 kurz beschrieben wurde. Auch hier wird ein Lexer und Parser zum Einlesen eingesetzt. Dafür werden die Lexer und Parser Generatoren Flex und Bison verwendet. Das Frontend ruft die Hauptfunktion des Parsers auf. Dort werden entsprechend der Definition in Bison nach und nach Tokens eingelesen. Dabei wird geprüft, ob die Tokens in der vorliegenden Reihenfolge auftreten dürfen. Im Parser sind Tokenfolgen definiert, bei denen bestimmte Aktionen ausgeführt werden sollen. Dabei werden die eingelesenen Tokens zu Symbolen zusammengefasst. Die Aktionen, die dabei ausgeführt werden, schreiben die jeweiligen Inhalte in das `yosys_design`.

Die Tokens werden vom Parser durch den Aufruf der Hauptfunktion des Lexers beschaffen. Dabei gibt ein Lexeraufruf jeweils ein Token zurück. Die Eingabedatei wird Zeichen für Zeichen eingelesen. Anhand von definierten regulären Ausdrücken wird geprüft, ob die eingelesene Zeichenkette einem Token entspricht. Ist das der Fall, wird ein Token als Zahl dargestellt und in dieser Form an den Parser zurückgegeben. Manche Tokens, wie zum Beispiel ein RTLIL-Identifikator, müssen ihren Wert an den Parser übergeben. Dafür schreibt der Lexer den Wert des Tokens in eine Variable des Parsers.

Sowohl der Lexer als auch Parser arbeiten mit Zustandsautomaten. Ein Zustandsautomat definiert eine Reihe von Zuständen und die Übergänge zwischen ihnen. So ruft ein neu eingelesener Token im Parser einen Zustandswechsel hervor. Ist der Automat an bestimmten Zuständen angekommen, werden die jeweiligen Aktionen ausgeführt. Wurde die Datei vollständig eingelesen, so muss sich der Zustandsautomat in einem der Endzustände befinden. Ist das nicht der Fall, handelt es sich um einen Syntaxfehler.

3.3 Synthese

Yosys implementiert sogenannte `synth`-Passes. Die Aufgabe dieser Pässe ist die Durchführung der Synthese. Ein `synth`-Pass ist in der Regel ein Skript-Pass, also ein Pass der andere Pässe zu einem bestimmten Zweck ausführt. Um ein Verständnis über den Syntheseverlauf zu gewinnen wird zunächst der generische `synth`-Pass betrachtet, welches eine hardwareunabhängige Synthese durchführt. Die synthetisierte Schaltung besteht aus generischen Komponenten, die keiner speziellen Hardware entsprechen. Basierend auf den daraus gewonnenen Erkenntnissen werden die einzelnen Pässe, die zur Durchführung einer Synthese notwendig sind, näher betrachtet. Um das Design auf einer reellen Hardware einsetzen zu können, muss das Design auf die darin enthaltene Komponenten abgebildet werden. Durch die Betrachtung von anderen `synth`-Pässen, die jeweils auf eine bestimmte Architektur, bzw. auf einen bestimmten FPGA ausgerichtet sind, soll ersichtlich werden, wie die Abbildung des Designs auf die Hardwarekomponenten funktioniert. Anschließend wird der Vorgang der Synthese zusammengefasst.

3.3.1 Makroübersicht der Synthese

Die Makroübersicht der Synthese soll anhand des `synth`-Passes durchgeführt werden. Dieser Pass befindet in der Datei `techlibs/common/synth.cc`. Da es sich bei diesem Pass um ein Skriptpass handelt, werden hier andere Pässe ausgeführt.

Zuerst wird der `hierarchy`-Pass, der unter anderem für die Auflösung von parametrisierten Modulen zuständig ist, ausgeführt. Dieser Pass ist in der Lage, das oberste Modul des Designs automatisch herauszufinden und zu markieren. Das oberste Modul kann auch explizit angegeben werden.

Mit dem `proc`-Pass werden Prozesse in einfache Zellen und Signale umgewandelt. Prozesse sind in der reellen Hardware wegen der sequentiellen Ausführungsart nicht umsetzbar. Ähnlich wie der `synth`-Pass, fasst der `proc`-Pass mehrere Unterpässe zusammen.

Nachfolgend werden die ersten Optimierungen durchgeführt. Mit dem `opt_expr`-Pass können Zellen, dessen Eingänge konstante Werte aufweisen, aufgelöst werden. Zusätzlich ist der Pass in der Lage simple Optimierungen der Logikausdrücke vorzunehmen. Mit dem `opt-clean`-Pass werden ungenutzte Zellen und Signale des Designs entfernt, da nach der Ausführung einiger Passes einige Zellen und Signale nicht mehr benötigt werden.

Zu diesem Zeitpunkt folgt der erste Aufruf des `check`-Passes. Dieser Pass prüft das Design gegen folgende Probleme:

- Kombinatorische Schleifen
- Widersprüchliche Zuweisungen eines Signals
- Signale mit undefiniertem Wert

Nachdem die Prüfung abgeschlossen ist, werden weitere Optimierungen mit dem `opt`-Pass durchgeführt. Ähnlich wie der `proc`-Pass führt der `opt`-Pass verschiedene Pässe, die für die Optimierungen zuständig sind, in einer sinnvollen Reihenfolge auf. Der Pass führt seine Unterpässe in einer Schleife, die erst verlassen wird, wenn keine weiteren Optimierungen durchgeführt wurden, aus. Hier werden vorerst D-Flip-Flop-Umwandlungen, die mit Takteingängen und synchronen Resets arbeiten, ignoriert. Optimierungen von Zustandsautomaten werden in einem separaten `fsm`-Pass durchgeführt. Dieser Pass ruft weitere Pässe auf. Nach den Zustandsautomatenoptimierungen wird der `opt`-Pass nochmals ausgeführt. Diesmal werden auch die obengenannten D-Flip-Flop-Umwandlungen ausgeführt.

Der nächste Pass(`wreduce`) optimiert Wortbreiten der Operationen, die standardmäßig mit 32 Bit erzeugt wurden. Durch diesen Pass wird zum Beispiel ein 32-Bit Adder, der nur 4-Bit zahlen addiert, mit einem 4-Bit Adder ersetzt. Nachfolgend wird eine Reihe der sogenannten *Peephole Optimierungen* in dem `peepopt`-Pass ausgeführt. Eine *Peephole Optimierung* ist eine Optimierungstechnik, die aus der Kompilererentwickelung hervorgeht. Dabei werden jeweils kleine, durch den Kompilierer generierte Gruppen von Befehlen in der Maschinensprache betrachtet und optimiert. Zu den Optimierungen gehört unter anderem die Entfernung von überflüssigen Instruktionen, oder das Zusammenfassen von mehreren Befehlen zu einem Befehl, der die Aufgabe effizienter Lösen kann. Hiernach wird nochmals der `opt_clean`-Pass ausgeführt.

Mit der `-lut k` Option wird mit dem `techmap`-Pass das erste Technologiemapping ausgeführt. Dieser Pass ist in der Lage existierende Zellen durch mehrere andere Zellen zu ersetzen. So können die generischen Zellen von Yosys in die in der Hardware verfügbare Zellen umgewandelt werden. Mit einer Verilog-Datei oder mit einem vorher eingelesenen RTLIL-Design werden die Regeln definiert, mit denen die Zellen ersetzt werden. In diesem Fall werden die Regeln aus den Dateien `techlibs/common/cmp2lut.v` und `techlibs/common/cmp2lcu.v` gelesen. Dadurch werden Vergleichsoperatoren durch LUTs abgebildet.

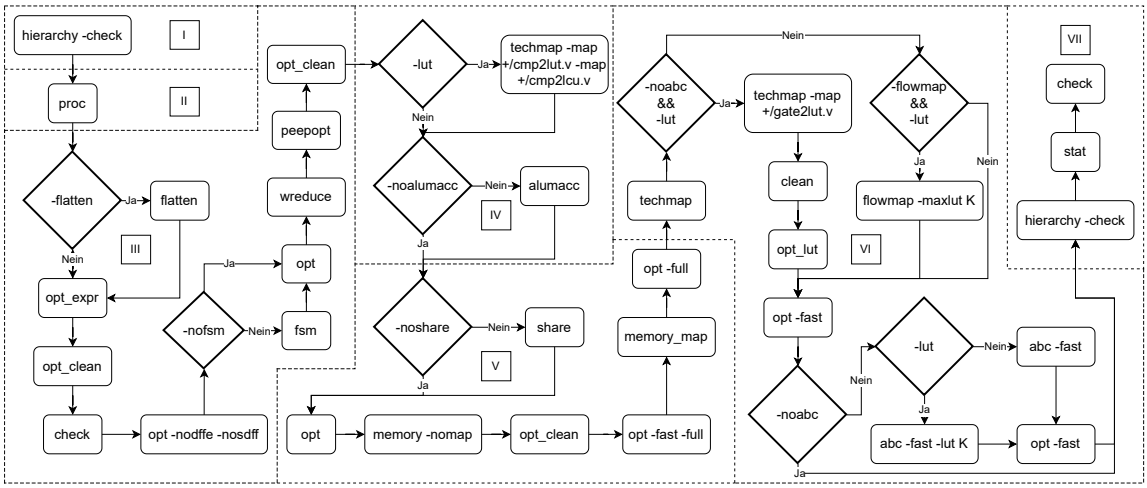


Abbildung 3.2: Ablauf des `synth`-Passes

Nachfolgend werden Zellen die für arithmetische Operationen zuständig sind im `alumacc`-Pass mit `$alu` und `$macc` Zellen ersetzt. Der `share`-Pass sucht nach Ressourcen, die zu einer Ressource zusammengeführt werden können und führt diese zusammen.

Der `opt`-Pass optimiert das Design nochmals, bevor der `memory`-Pass ausgeführt werden kann. Der `memory`-Pass ruft seine Unterpässe in einer sinnvollen Reihenfolge auf, um generische Speicher in Speicherblöcke umzuwandeln. Mit dem `opt_clean`-Pass wird das Design vor dem weiteren Verlauf aufgeräumt.

Das Design wird mit einem Teil des `opt`-Passes weiter optimiert. Nur `opt_expr`, `opt_merge`, `opt_dff` und `opt_clean` werden ausgeführt. Der `memory_map`-Pass konvertiert die vorher generierte Speicherblöcke zu Flip-Flops woraufhin das Design mit dem `opt`-Pass nochmals optimiert wird.

Der `techmap`-Pass wird ausgeführt. Diesmal wird eine interne Regeldefinition genutzt, welche die Register Transfer Level (RTL)-Zellen in die intern definierte, gatterbasierte Zellen umwandelt. Wenn die `-lut k` und `-noabc` Optionen angegeben wurden, werden die gatterbasierten Zellen in LUTs umgewandelt. Dies geschieht wieder mit dem `techmap`-Pass mit der Definition aus der Datei `techlibs/common/gate2lut.v`. Hiernach räumt der `clean`-Pass das Design auf und der `opt_lut`-Pass kombiniert mehrere LUTs nach Möglichkeit zusammen, wenn die Eingänge nicht jeweils voll ausgenutzt wurden. Wurde die `-flowmap` Option statt `-noabc` angegeben, wird stattdessen der `flowmap`-Pass ausgeführt. Dieser Pass nutzt den FlowMap-Algorithmus[12]. Der Algorithmus soll ein LUT-Netzwerk mit minimaler Tiefe erzeugen, was direkt zu minimalen Verzögerungen in dem Netzwerk führe. Gleichzeitig soll er die Anzahl der genutzten LUTs unter Beibehaltung der optimalen Tiefe minimieren können[12, S. 2]. Anschließend wird wieder ein Teil des `opt`-Passes ausgeführt.

Wenn weder `-noabc` noch `-flowmap` angegeben wurde, wird das Technologiemapping mit Hilfe des ABC-Programms durchgeführt. Gegebenenfalls wird auch der Wert aus der `-lut k` Option übergeben. Wenn das ABC-Technologiemapping durchgeführt wurde, wird nochmals ein Teil des `opt`-Passes ausgeführt.

Zum Schluss wird der `hierarchy`-Pass zum zweiten Mal ausgeführt. Mit dem `stat`-Pass werden die Statistiken des Designs ausgegeben. Als Letztes wird mit dem `check`-Pass das Design ein letztes Mal überprüft.

Abbildung 3.2 zeigt den Ablauf des `synth`-Passes. Zusammenfassend besteht die generische Yosys-Synthese aus Sieben Phasen:

I Bestimmung der Design-Hierarchie

II Auflösung der Prozesse

III Erste Optimierungsphase

IV Vorbereitungen für das Technologiemapping

V Zweite Optimierungsphase

VI Technologiemapping

VII Designüberprüfung und Statistikausgabe

3.3.2 Analyse der Passes

Basierend auf den Erkenntnissen der Makroübersicht der Synthese sollen die dort aufgeführten Passes im Detail betrachtet werden. Die Reihenfolge der Pässe wird in etwa der Reihenfolge der Ausführung in dem `synth`-Pass entsprechen.

3.3.2.1 Hierarchy und Check

Der `Hierarchie`-Pass ist in der Datei `passes/hierarchy/hierarchy.cc` definiert. Er übernimmt die Überprüfung der Hierarchie des Designs. Der Pass nimmt die in der Tabelle 3.4 beschriebenen Argumente an und agiert entsprechend.

Zuerst wird versucht, ein Top-Modul zu identifizieren. Das Modul, das mit dem `-top`-Argument angegeben wurde, wird markiert. Dabei werden gegebenenfalls die Parameter aus dem `-chparam` -Argument übernommen und das Top-Modul entsprechend angepasst. Ohne des `-top`-Arguments wird ein Modul mit dem `\top`-Attribut gesucht und in der `top_mod`-Variable gespeichert. Wenn kein Top-Modul gefunden werden konnte und der `-auto-top`-Argument gesetzt ist, wird ein passendes Top-Modul aus dem gesamten Design rausgesucht und markiert. Die Entscheidung, ob ein Modul ein Top-Modul ist, basiert auf der Anzahl der Hierarchieebenen, die unterhalb des Moduls vorzufinden sind. Das Modul mit der tiefsten Hierarchie wird zum Top-Modul. Ein Modulname kann ein `$abstract`-Präfix enthalten. Enthält das Top-Modul dieses Präfix, werden seine Parameter mit den Parametern aus dem `-chparam`-Argument angepasst und das Modul neu evaluiert.

Die Module die in derselben Programmsitzung mit dem AST-Frontend erstellt wurden, also auch solche die mit dem Verilog-Frontend gelesen wurden, sind Objekte des `AstModule`-Typs, der von dem `RTLIL::Module` erbt. Das `AstModule` enthält Methoden, mit denen ein neues Modul mit anderen Parameterwerten erzeugt werden kann. Diese Methoden wurden bereits in dem `RTLI::Module` als `virtual` definiert und sind deshalb auch aus dem `RTLIL::Module` erreichbar. Dazu werden die Informationen aus der AST-Struktur genutzt. Bei der Angabe von `-chparam`-Argumenten werden die Methoden genutzt, um das Modul abzuleiten.

Wenn der `-simcheck`-Argument angegeben wurde, wird das Bestehen eines Top-Moduls überprüft. Anschließend wird sichergestellt, dass maximal ein Modul als das Top-Modul markiert wird. Das Top-Modul wird vorläufig mit dem `\initial_top`-Attribut markiert.

Argument	Beschreibung
-check	Überprüft die Designhierarchie(Fehler bei fehlender Moduldefinition für eine Zelle und bei keiner Übereinstimmung der Zell- und Modulports und -parameter)
-simcheck	Überprüft die Designhierarchie wie -check. Zusätzlich schlagen ein fehlendes Top-Modul und die Instanziierung eines Blackbox-Moduls fehl.
-purge_lib	Löscht ungenutzte Blackbox-Module
-libdir <directory>	Gibt ein Verzeichnis mit Verilog-Moduldefinitionen (<module_name>.v) an. Ein Modul wird eingelesen, wenn eine Zelle einen unbekannten Typ hat.
-keep_positionals	Schaltet die Ersetzung der positionellen Ports und Parameter in Zellen mit den Namen der Ports und Parameter aus
-keep_portwidths	Schaltet die Anpassung der Portbreiten in den Zellen, wenn die Breite mit dem Modul nicht übereinstimmt, aus
-nodefaults	Schaltet die Auflösung des \defaultvalue Attributs der Signale aus
-nokeep_asserts	Schaltet das Setzen des \keep Attributs bei Modulen aus. Das Attribut gibt an, ob in der Designhierarchie, Zellen zur formellen Verifikation existieren.
-top <module>	Setzt das Top-Modul der Hierarchie mit dem Attribut \top
-auto-top	Das Top-Modul wird automatisch erkannt
-chparam name value	Überschreibt die Parameter des Top-Moduls
-generate	Generiert Blackbox-Module mit den angegebenen Ports

Tabelle 3.4: Hierachy-Pass Argumente

Der nächste Abschnitt beschäftigt sich mit dem SystemVerilog-Konstrukt Interface. Damit ist es in Verilog möglich mehrere Signale zu einer zusammenhängender Einheit zusammenzufassen. Dies wird hier nicht weiter behandelt. Nach der Interfaceverarbeitung kann nun das \initial_top-Attribut mit dem endgültigen \top-Attribut ersetzt werden. Ein weiteres SystemVerilog-Feature, dass nicht weiter behandelt werden soll, ist die formelle Verifikation. Das -nokeep_asserts-Argument steuert die Verarbeitung dieses Features.

Nachfolgend werden alle positional angegebene Portverbindungen und Parameterwerte bei Zellen auf die eigentlichen Namen aus den jeweiligen Modulen abgebildet. Das -keep_positionals-Argument kann die Umwandlung ausschalten. Die Portverbindungen und Parameter, deren linker Identifikator mit einem $ und einer Zahl beginnt, werden als positionelle Portverbindungen erkannt. Der Identifikator wird dann mit dem Port- oder Parameternamen des Moduls ersetzt.

In Verilog können Signale Standardwerte erhalten. Im Fall von Eingangssignalen eines Moduls müssen diese Standardwerte in den jeweiligen Zelleninstanziierungen zugewiesen werden. Dafür werden zunächst die Signale eines jeden Moduls auf der Suche nach dem \defaultvalue-Attribut durchiteriert und zwischengespeichert. Anschließend werden alle Zellen eines jeden Moduls untersucht und die Portzuweisungen gegebenenfalls entsprechend angepasst. Mit dem -keep_positionals-Argument werden Zellen mit positionellen Portzuweisungen ignoriert.

In Verilog können Signale als wand oder wor statt wire definiert werden. Hierbei handelt es sich um Signale, die mehrere Werte gleichzeitig zugewiesen bekommen können. Konflikte werden vor-

gebeugt, indem den Signalen jeweils ein Und- oder ein Oder-Gatter vorgeschaltet wird. Die Signale in den Zuweisungen werden mit den vorgeschalteten Gattern verbunden. Der Ausgang des Gatters entspricht dem eigentlichen Signal. Zuerst werden alle Signale eines jeden Moduls auf der Suche nach dem `\wand` oder dem `\wor`-Attribut gesucht und zwischengespeichert. Alle Verbindungen im Modul werden betrachtet und die Zuweisungen zu den `\wand` und zu den `\wor`-Signalen werden zwischengespeichert. Dies gilt auch für Ausgänge der Zellen. Zuletzt werden für alle gefundenen Signale eine Und- oder eine Oder-Zelle erstellt. Hat das Signal eine Breite von 1, wird eine `reduce`-Zelle genutzt. Eine `reduce`-Zelle verbindet alle Signale mit der gleichen Funktion(Und oder Oder) und produziert ein neues 1-Bit Signal. Bei mehrbittigen Signalen werden mehrere Und- oder Oder-Gatter erstellt und die Signale jeweils verbunden.

Nachfolgend werden Blackbox-Zellen mit parametrisierten Portbreiten mit dem AST-Frontend neu evaluiert. Zuletzt werden die Zellenports an die Modulportbreiten angepasst. Ist die Portbreite kleiner, so wird die Portzuweisung auf die richtige Größe zugeschnitten. Handelt es sich dabei um ein Ausgang, so werden die überflüssigen Bits mit einer separaten Verbindung innerhalb des Moduls gelöst. Ist die Portbreite größer, wird in dem Fall eines Eingangs das Zuweisungssignal um konstante Bits erweitert. Ansonsten werden die zusätzlichen Bits einem neuen Signal zugewiesen.

Der `check`-Pass erkennt kombinatorische Schleifen, Konflikte bei Zuweisungen und Signale mit undefinierten Werten und ist in der Datei `passes/cmds/check.cc` definiert. Zunächst werden aus allen Prozessen die einzelnen Signalbits, die entweder einen Wert zugewiesen bekommen oder zu einem anderen Bit einen Wert zuweisen, in separaten Strukturen gespeichert. Ebenso werden Zellenportzuweisungen durchsucht. Zusätzlich werden Zellen, die evaluierbar(Und, Add, Mux) sind zwischengespeichert. Über diese Zellen kann eine kombinatorische Schleife entstehen. Zuletzt werden Modulports auf Zuweisungen untersucht.

Mit den gesammelten Informationen können nun Probleme mit dem Design erkannt werden. Werden einem Signal mehrere Werte zugewiesen, kann nicht sichergestellt werden, dass die Signale nicht miteinander kollidieren werden. Bekommt ein Signal, das für eine Zuweisung genutzt wird, selber keinen Wert zugewiesen, entsteht ein Signal mit undefinierten Werten. Zuletzt werden kombinatorische Schleifen mit Hilfe des topologischen Sortieralgorithmus erkannt.

3.3.2.2 Auflösung der Prozesse

Die Aufgabe des `proc`-Passes ist die Umwandlung der Prozesse in einfache Zellen und Signale. Der `proc`-Pass ist in der Datei `passes/proc/proc.cc` definiert. Die Aufgaben des Passes sind in mehrere Unterpässe aufgeteilt. Diese Pässe sind im selben Verzeichnis(`passes/proc`) definiert. Abbildung 3.3 zeigt die Reihenfolge der Unterpässe.

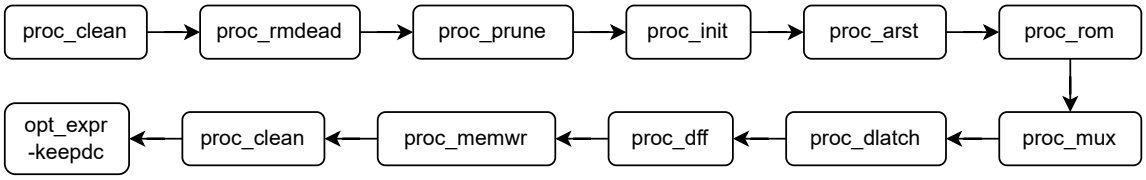


Abbildung 3.3: Ablauf des `proc`-Passes

Entfernung von wirkungslosen Prozessteilen

Der `proc_clean`-Pass entfernt leere Teile der Prozesse. Zuerst werden die Synchronisationsregeln durchiteriert. Die Aktionen, bei denen die linke Seite leer ist, werden entfernt. Die Speicherschreibaktionen werden zunächst beibehalten. Anschließend werden ganze Synchronisationsregeln entfernt, wenn sie keine Aktionen beinhalten.

Als Nächstes wird der switch-case-Baum durchiteriert und aufgeräumt. Ähnlich wie bei den Synchronisationsregeln werden die Aktionen einer `CaseRule` entfernt, deren linke Seite leer ist. Nachfolgend werden leere `SwitchRule`s entfernt. Für die `SwitchRule`s, die nicht leer sind, werden weitere Umstrukturierungen durchgeführt.

Zunächst werden die `SwitchRule`s betrachtet, die ein konstantes Vergleichssignal beinhalten. Dort werden zunächst die `CaseRule`s entfernt, die nicht eintreten können. Dazu gehören `CaseRule`s mit konstantem Vergleichssignal, die mit dem Vergleichssignal der `SwitchRule` nicht übereinstimmen. Eine `CaseRule` mit leerem Vergleichssignal ist für den Standardfall, der eintritt, wenn keine anderen Fälle eingetreten sind, bestimmt. Wird eine solche `CaseRule` gefunden, bevor eine `CaseRule` mit einem übereinstimmenden konstanten Vergleichssignal gefunden wurde, werden alle nachfolgenden `CaseRule`s als nicht eintretbar behandelt, da der Standardfall alle nachfolgenden Fälle abfangen würde. Eine Standard-`CaseRule` wird gelöscht, wenn eine übereinstimmende `CaseRule` bereits gefunden wurde. Wenn die erste `CaseRule` übereinstimmt wird das Vergleichssignal der `SwitchRule` geleert. Wenn eine `SwitchRule` in der übergeordneten `CaseRule` die erste `SwitchRule` ist, werden alle Aktionen und `SwitchRule`s der ersten `CaseRule` in die übergeordnete `CaseRule` verschoben. Die sonstigen `CaseRule`s einer `SwitchRule` werden rekursiv behandelt. Anschließend werden leere Prozesse vollständig gelöscht.

Entfernung von nicht erreichbaren Entscheidungszweigen in Prozessen

Der `proc_rmdead`-Pass entfernt nicht erreichbare Entscheidungszweige aus den Prozessen. Für das Vergleichssignal einer `SwitchRule` werden alle möglichen Bitmuster errechnet. Stimmt ein Vergleichssignal einer `CaseRule` mit keinem der Bitmuster überein, so ist dieser Zweig nicht erreichbar und wird entfernt. Anschließend werden `CaseRule`s ohne einem Vergleichssignal entfernt, wenn die `CaseRule` keine Standard-`CaseRule` ist. Eine Standard-`CaseRule` wird erkannt, wenn die Vergleichssignale der `SwitchRule` und der `CaseRule` leer sind. Für untergeordnete `SwitchRule`s wird der Vorgang rekursiv wiederholt.

Entfernung von Zuweisungen, welche in dem selben Prozess überschrieben werden

Bevor die eigentliche Umwandlung der Prozesse beginnen kann, werden mit dem `proc_prune`-Pass die Zuweisungen entfernt, die später im Prozess bedingungslos überschrieben werden. Werden die Zuweisungen in einem switch-case-Baum, der alle möglichen Fälle abdeckt, in jedem Fall eintreten, so werden die Zuweisungen des gleichen Signals in der höheren Ebene überschrieben und können entfernt werden. Wichtig bei diesem Prozess ist, dass das Verilog-Frontend beim Einlesen der Prozesse temporäre Zwischensignale erstellt, um die Reihenfolge der Zuweisungen aus dem Verilog-Entwurf zu simulieren. Ein beispielhaftes Verilog-Modul

```
1  module proc_prune();
2    reg w1;
3    wire [1:0]w2;
4    always begin
5      case(w2)
6        2'b10:  w1 = 1'b1;
7        2'b01: w1 = 1'b0;
8        default: w1 = 1'b0;
9      endcase;
10     w1 = 1'b1;
11   end;
12 endmodule
```

wird zu folgender RTLIL-Beschreibung übersetzt.

```
1  module \proc_prune
2    wire $1\w1[0:0]
3    wire $0\w1[0:0]
4    wire width 2 \w2
5    wire \w1
6    process $proc$proc_prune.v:4$1
7      assign $0\w1[0:0] 1'1
8      switch \w2
9        case 2'10
10         assign $1\w1[0:0] 1'1
11       case 2'01
12         assign $1\w1[0:0] 1'0
13       case
14         assign $1\w1[0:0] 1'0
15     end
16     sync always
17       update \w1 $0\w1[0:0]
18   end
19 end
```

Die Zuweisung zu `w1` in der Verilogzeile 10 findet erst nach der `case`-Anweisung statt. Die Zuweisungen innerhalb der `case`-Anweisung haben also keine Wirkung. Wäre die Zuweisung vor der `case`-Anweisung stattfinden, würde sie nicht eintreten können, da `w1` in jedem Zweig der `case`-Anweisung zugewiesen wird und der Wert der Zuweisung in jedem Fall überschrieben wäre. In diesem Fall wäre in der RTLIL-Zeile 7 die folgende Zuweisung generiert worden: `assign \$0\w1[0:0] \$1\w1[0:0]`. Damit wären die Zuweisungen aus der `case`-Anweisung in `\$0\w1[0:0]` übernommen worden und somit die Reihenfolge der Zuweisungen korrekt übersetzt.

Der `proc_prune`-Pass untersucht die `CaseRule`s in den tiefsten Ebenen zuerst. Für die dazugehörige `SwitchRule` werden Bits festgehalten, die in allen `CaseRule`s zugewiesen werden. Die in einer

SwitchRule gesammelten Zuweisungen werden in die dazugehörige CaseRule weitergegeben, wenn alle möglichen Entscheidungszweige der SwitchRule abgedeckt werden. Für jede weitere CaseRule und SwitchRule werden die Zuweisungen der tieferen Ebenen mitbetrachtet. Während des Vorgangs werden die überflüssigen Bitzuweisungen entfernt. In der obersten CaseRule können Zuweisungen zu einfachen Verbindungen des Moduls werden, wenn die jeweiligen Bits in den tieferen case-Ebenen nicht zugewiesen wurden.

Auflösung der Init-SyncRules

Der erste Vorgang der Auflösung der Prozesse ist die Umwandlung der init-SyncRules. Eine init-SyncRule entspricht einer Zuweisung, die sich in einem Verilog initial-Block befindet. Die Zuweisungen der SyncRules sollen für die entsprechenden Signale als ein \init-Attribut gespeichert werden. Für alle init-SyncRules werden die Aktionen der SyncRules durchiteriert. Zuerst mit Hilfe einer SigMap wird für die rechten Seiten der Zuweisungen ein repräsentierender Wert ermittelt. Ist dieser Wert keine Konstante, kann die weitere Verarbeitung der SyncRule nicht fortgeführt werden und der Pass schlägt fehl. Anschließend werden die SigChunks der linken Seite durchlaufen und für die dazugehörigen Signale die init-Attribute extrahiert. Wenn nicht bereits erfolgt, wird der init-Wert vorerst mit undefinierten Bits gefüllt. Zuletzt werden die undefinierten Bits mit den Bits der rechten Seite der Zuweisung ersetzt. Wenn ein Bit bereits gesetzt wurde und einen anderen Wert zugewiesen bekommen sollte, werden widersprüchliche Initialisierungswerte festgestellt und der Pass schlägt fehl.

Erkennung von asynchronen Resets in Prozessen

Mit Prozessen lässt sich unter anderem eine resetgesteuerte Logik modellieren. In HDLs werden Prozesse ausgeführt, wenn sich der Wert eines Signals aus der Sensitivitätsliste ändert. Damit lassen sich Speicherelemente auf ihre Standardwerte zurücksetzen. In resetgesteuerter Logik wird ein Reset-Wert so lange gehalten, bis der Reset nicht mehr aktiv ist. Dazu eignet sich eine pegelgesteuerte Erkennung der Werte. In RTLIL können SyncRules pegelsensitiv angegeben werden. Der `proc_arst` -Pass wandelt flankengesteuerte Darstellungen in pegelgesteuerte Darstellungen. Dabei wird ein konstanter Wert ermittelt, der bei einem Reset zugewiesen werden soll. Alle Prozesse des Designs werden durchiteriert. Damit eine resetgesteuerte Logik existiert setzt der Pass voraus, dass in der obersten Ebene des Prozesses nur eine SwitchRule existiert.

Ist das der Fall, werden alle flankengesteuerte SyncRules betrachtet. Das Signal, das die SyncRule aktiviert, wird mit dem Entscheidungssignal der SwitchRule verglichen. Hängen die beiden Signale voneinander ab, handelt es sich um einen asynchronen Reset. Der SyncRule-Typ wird in einen pegelgesteuerten Typ umgewandelt. Eine positive flankengesteuerte SyncRule wird dabei zu einem positiven pegelgesteuertem Typ und eine negative flankengesteuerte SyncRule wird zu einem negativen pegelgesteuertem Typ.

Für alle Signale, die in der SyncRule zugewiesen werden, wird im switch-case-Baum ein konstanter Wert für den Reset gesucht. Konstante Bits der Zuweisungen in der SyncRule werden übernommen und als Resetwerte behandelt. Zuerst werden die obersten Zuweisungen des Prozesses untersucht und Konstanten als Resetwert übernommen. Nachfolgend werden die SwitchRules untersucht. SwitchRules die vom selben Signal wie die SyncRulesabhängen, werden für die Ermittlung der Konstanten betrachtet. Für die CaseRule, die dem Resetpegel entspricht, werden aus den Zuweisungen der CaseRule die Konstanten wie oben beschrieben übernommen. Dort werden die SwitchRules auf die gleiche Art und Weise untersucht. Dabei gilt, dass bei einer SwitchRule mit einem leeren

Entscheidungssignal alle CaseRules verarbeitet werden. Bei SwitchRules deren Entscheidungssignale nicht vom Signal der SyncRule abhängen, werden die in dem Unterbaum zugewiesene Bits markiert. Ein markiertes Bit ist ein Bit, das keinen konstanten Wert zugewiesen bekommen kann. Die Vorgänge werden rekursiv fortgeführt und wiederholt, bis das Signal der SyncRule einen konstanten Wert bekommt. Am Ende werden die Reset Zweige des switch-case-Baumes entfernt und mit dem proc_clean-Pass anschließend die restlichen CaseRules in die obere Ebene verschoben.

Erkennung von ROMs

Der proc_rom-Pass sucht in Prozessen nach SwitchRules, die in eine ROM umgewandelt werden können. Umgewandelt werden SwitchRules, die folgende Voraussetzungen erfüllen:

- Es existiert mindestens eine CaseRule.
- Die CaseRules beinhalten keine weiteren SwitchRules.
- Rechte Seiten der Zuweisungen in den CaseRules sind konstant.
- Alle CaseRules weisen die Werte zu denselben Signalen zu.
- Die Vergleichssignale der CaseRules müssen konstant sein.
- Alle Fälle die eintreten können, werden von CaseRules abgedeckt. Standardfälle gelten hierbei auch.

Zusätzlich werden keine ROMs erstellt, wenn der resultierende ROM folgende Eigenschaften aufweisen würde:

- Die Breite der Adressen ist größer als 30.
- Es werden weniger als Acht Werte im ROM gespeichert.
- Weniger als 25% der Adressen beinhalten einen Wert.

Zuerst werden die tiefsten SwitchRules der Hierarchie verarbeitet. Es wird ermittelt zu welchen Signalen die Zuweisungen in der ersten CaseRule stattfinden. Alle anderen CaseRules müssen auf der linken Seiten ihrer Zuweisungen dieselben Signale aufweisen. Nachfolgend wird das Entscheidungssignal der SwitchRule untersucht. Es werden die letzten zusammenhängenden Nullen gefunden und die Größe des Signals ohne den Nullen ermittelt ((000)1011011).

Für jede CaseRule werden die Zuweisungen untersucht und gegen die obengenannten Voraussetzungen geprüft. Dabei werden die Werte und ihre Adressen aus den rechten Seiten der Zuweisungen und aus den Entscheidungssignalen ermittelt.

Es wird ein neues Signal im Modul erstellt. Die Größe des Signals entspricht der Anzahl der Bits, die in den CaseRules zugewiesen werden. Dies ist gleichzeitig auch die Wortbreite des ROMs. Die Anzahl der Adressbits des ROMs wird aus der höchsten ermittelten Adresse errechnet. Die Größe des ROMs wird aus $(2)^{Anzahladressbits+1}$ errechnet. Die ermittelten Werte werden dem ROM als Initialwerte zugeteilt. Das Entscheidungssignal der SwitchRule wird zum Adresseingang und das vorher erstellte Signal zum Datenausgang des ROMs.

Zuletzt werden die CaseRules der SwitchRule neu aufgebaut. Dafür werden zunächst alle CaseRules gelöscht. Wurden alle Bits des Entscheidungssignals der SwitchRule für den Adresseingang genutzt, wird in der SwitchRule das Entscheidungssignal geleert und eine CaseRule hinzugefügt, in welcher

der Datenausgang mit dem vorher erstelltem Signal verbunden wird. Sind in dem Entscheidungssignal der SwitchRule noch freie Bits verfügbar, werden zwei CaseRules hinzugefügt. Die erste CaseRule gibt den Wert aus dem ROM weiter. Die zweite CaseRule spiegelt den Standardfall der SwitchRule wider.

Diese Vorgänge werden für alle SwitchRules in der tiefsten Ebene ausgeführt. Die restlichen SwitchRules kommen nicht in Frage, da sie die tieferen SwitchRules enthalten.

Umwandlung der Entscheidungsbäume in Multiplexerbäume

Die Aufgabe des proc_mux-Passes ist die vollständige Auflösung der Entscheidungsbäume in den Prozessen. Dafür werden alle Prozesse des Designs durchiteriert. Für jeden Pass werden drei Hilfsobjekte erstellt. Die swpara-Map hält fest, ob die CaseRules einer SwitchRule parallel sind. Die CaseRules sind parallel, wenn alle Entscheidungssignale der CaseRules einer SwitchRule konstant und volldefiniert sind und nur einmal vorkommen. Ein SigSnippets-Objekt enthält alle zusammenhängende Teile der Signale, die in dem Prozess zugewiesen werden. Das Listing 3.38 zeigt die Definition des SigSnippets-structs. sigidx enthält die Snippets und ihre IDs. bit2snippet speichert die Zuordnung von einzelnen Bits zu der Snippet-ID. Da sigidx eine bereits vergebene ID nicht löschen kann, werden in snippets alle validen Snippet-IDs gespeichert. Die insert-Methode in Zeile 98 findet alle linken Seiten der Zuweisungen im gesamten Entscheidungsbaum und ruft mit ihnen die insert-Methode in der Zeile 36, welche im jeden SigSpec überprüft, ob ein Teil des SigSpecs nicht bereits ein Snippet ist, auf.

```
30  struct SigSnippets
31  {
32    idict<SigSpec> sigidx;
33    dict<SigBit, int> bit2snippet;
34    pool<int> snippets;
35
36    void insert(SigSpec sig)
98    void insert(const RTLIL::CaseRule *cs)
107 };
```

Listing 3.38: SigSnippets-struct aus dem proc_mux-Pass[6]

Wenn kein bereits existierender Snippet festgestellt wird, werden, die einzelnen Bits des SigSpecs zusammen mit der ID und die ID selbst in den obengenannten Variablen gespeichert. Sobald ein SigBit gefunden wird, das bereits zu einem anderen Snippet gehört werden zunächst die bereits gesammelten SigBits in den Variablen gespeichert. Das gefundene Snippet muss aufgeteilt werden. Der erste Teil sind die SigBits, die vor dem ersten SigBit liegen, der gerade hinzugefügt werden sollte. Der zweite Teil ist der in dem neuen und alten Snippet übereinstimmender Bereich. Der letzte Teil sind die übrigen SigBits des alten Snippets. Die ID und die einzelnen SigBits des alten Snippets werden aus snippets und bit2snippet gelöscht und die drei Teile rekursiv wieder eingefügt. Wird kein SigBit mehr gefunden, das zu einem bereits existierenden Snippet gehört, werden die restlichen SigBits in ein neues Snippet eingefügt. Ansonsten werden die oben beschriebene Vorgänge wieder ausgeführt. Die Abbildung 3.4 illustriert die Aufteilung der Signale in Snippets.

Ein SnippetSwCache-Objekt ordnet die IDs der Snippets den Unterbäumen zu, in denen sie zugewiesen werden. Das Listing 3.39 zeigt die Definition der Datenstruktur. Ein Unterbaum wird mit einer SwitchRule identifiziert und in der cache-Map mit den IDs der Snippets gespeichert. snippets ist

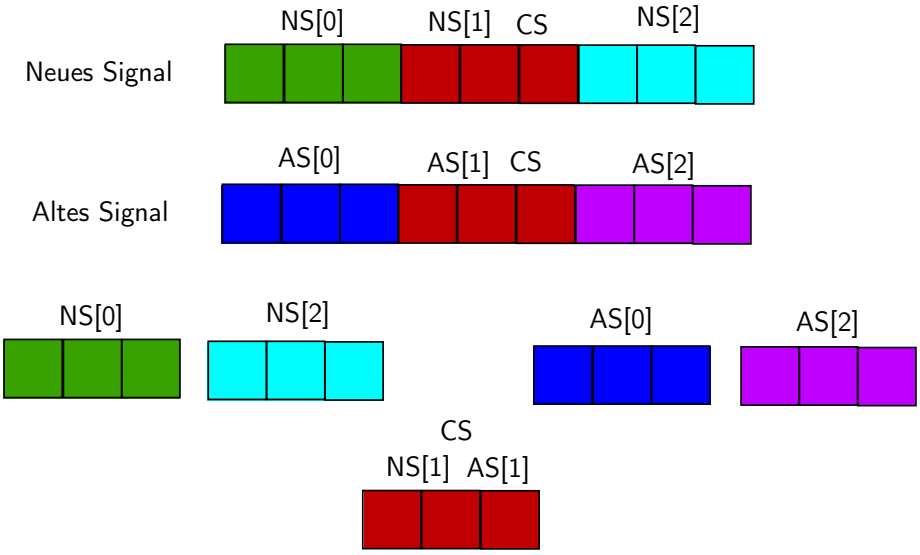


Abbildung 3.4: Veranschaulichung der Snippeterstellung

ein Zeiger auf das vorher erstellte `SigSnippet`-Objekt und die `current_snippet`-Variable speichert die ID des aktuell ausgewählten Snippets. Die Map `full_case_bits_cache` speichert die `SigBit`s, die in einer `SwitchRule` in jedem Fall des Unterbaumes einen Wert zugewiesen bekommen. Die `insert`-Methode in Zeile 141 erstellt einen `SwitchRule`-Stapel und ruft die `insert`-Methode in der Zeile 122, welche die gesuchten `SigBit`s identifiziert und speichert. Die `check`-Methode prüft, ob das aktuell gewählte Snippet in dem Unterbaum der `SwitchRule` zugewiesen wird. Die tiefer in der Hierarchie enthaltenen `SwitchRule`s werden rekursiv behandelt. Dabei wird die aktuelle `SwitchRule` im Stapel gespeichert, damit für eine Zuweisung, für alle `SwitchRule`s in der Hierarchie, die Snippet-IDs gespeichert werden können.

```
109 struct SnippetSwCache
110 {
111   dict<RTLIL::SwitchRule*, pool<RTLIL::SigBit>, hash_ptr_ops>
        full_case_bits_cache;
112   dict<RTLIL::SwitchRule*, pool<int>, hash_ptr_ops> cache;
113   const SigSnippets *snippets;
114   int current_snippet;
115
116   bool check(RTLIL::SwitchRule *sw)
117   {
122   {
141   {
```

Listing 3.39: `SnippetSwCache`-`struct` aus dem `proc_mux`-Pass[6]

Nun können für die jeweiligen Snippets Multiplexerbäume erzeugt werden. Dafür werden die in `SigSnippets` identifizierten Snippets durchlaufen. Das aktuelle Snippet wird in `SnippetSwCache` als das aktuelle Snippet vermerkt. Mit der obersten `CaseRule` angefangen, werden die `CaseRule`s und `SwitchRule`s rekursiv verarbeitet. Die Werte der Zuweisungen des Snippets in einer `CaseRule` werden zunächst zwischengespeichert und anschließend die `SwitchRule`s betrachtet. Für eine `SwitchRule` wird überprüft, ob die einzelne `CaseRule`s parallel sind. Ist das nicht der Fall, werden Gruppen von parallelen `CaseRule`s gesammelt.

Es wird überprüft, welche Bits des Snippets in diesem Unterbaum in jedem Fall zugewiesen werden. Die Bits werden mit State::Sx überschrieben und in SnippetSwCache in full_case_bits_cache gespeichert.

Die CaseRules der SwitchRule werden durchiteriert und jede CaseRule wird rekursiv verarbeitet. So fängt der Aufbau der Multiplexerbäume mit der tiefsten Hierarchieebene an. Die Erstellung der Multiplexerzellen wird im Standardfall und wenn die beiden Multiplexersignale gleich sind nicht durchgeführt. Ansonsten muss für jeden Multiplexer ein Steuersignal ermittelt werden. Die Vergleichssignale der SwitchRule und der CaseRules werden miteinander mit $eq-Zellen verglichen. Daraus entsteht für eine CaseRule ein Signal, das aus Ausgangsbits der $eq-Zellen besteht. Die Bits dieses Signals müssen mit einer $reduce_or-Zelle zu einem Steuersignal zusammengefasst werden. Wenn die CaseRule nur ein Vergleichssignal besitzt wird keine $reduce_or-Zelle benötigt, da der Ausgang einer $eq-Zelle ohnehin nur ein Bit breit ist. Die $eq-Zellen werden nicht benötigt, wenn die CaseRule ein leeres Vergleichssignal enthält und damit ein Standardfall ist. In diesem Fall wird auch keine Multiplexerzelle erzeugt, da die Entscheidung, ob ein Standardfall eintrifft, letztlich von den restlichen CaseRules abhängt. Ist die Größe der Entscheidungssignale und der Wert des Vergleichssignals der CaseRule gleich Eins, würde der Ausgang der $eq-Zelle immer den Wert des Vergleichssignals der SwitchRule annehmen. Damit kann die Erzeugung der $eq-Zelle auch in diesem Fall ignoriert werden.

Die Multiplexerzelle und das Ausgangssignal können nun erzeugt werden. Wenn zwei CaseRules parallel sind, wird eine Multiplexerzelle in eine parallele Multiplexerzelle umgewandelt. Diese Zelle kann mehrere Multiplexer, deren Eingangssignale gleich Breit sind und jeweils nur eines der Steuersignalen Eins wird, zusammenfassen. Die Eigenschaft, dass nur ein Steuersignal gleichzeitig Eins sein kann folgt aus der Existenz der parallelen CaseRules.

Während der Verarbeitung eines Snippets werden die Ausgänge mit den else_signal-Eingängen der Multiplexerzellen verbunden. Damit entsteht ein Multiplexerbaum. Da die CaseRules in umgekehrter Reihenfolge verarbeitet werden, das heißt von der letzten in RTLIL gelisteten CaseRule angefangen, wird richtigerweise die oberste CaseRule den Einfluss darauf haben, ob der Ausgang des restlichen Multiplexerbaumes oder der andere Eingang weitergeleitet wird. Nach der Verarbeitung aller gefundenen Snippets ist der switch-case-Baum in RTLIL vollständig aufgelöst.

Erkennung von Latches

Nach dem proc_mux-Pass bleiben in den Prozessen nur noch die SyncRules. Der nächste Pass (proc_dlatch) beschäftigt sich mit den always-SyncRules, welche im Rahmen des Passes in *dlatches* oder in einfache Verbindungen im Modul umgewandelt werden. Ein Latch entsteht, wenn ein Ausgang eines Multiplexers möglicherweise über mehrere andere Multiplexer auf einen Eingang zurückgespeist wird. Der Pass implementiert eine Hilfsstruktur proc_dlatch_db_t, die im Listing 3.40 dargestellt wird. sigmap enthält Gruppen von SigSpecs, die im Module miteinander verbunden sind und wählt für jede Gruppe einen repräsentierenden SigSpec. In initvals werden alle Initialisierungswerte der Signale im Module gespeichert. Die im Verlauf des Passes erzeugten Latches werden in generated_dlatches gespeichert. mux_srcbits speichern die Eingänge der im proc_mux-Pass erstellten Multiplexer zwischen. mux_drivers enthält die Ausgangsbits der Multiplexer und die Zelle selbst. In sigusers wird für ein SigBit gespeichert, von wie vielen Zellen der SigBit als Eingang genutzt wird.

Für jeden Prozess werden zunächst die always-SyncRule-Aktionen darauf überprüft, ob durch sie eine Rückspeisung des Ausgangssignals auf das Eingangsignal desselben Multiplexers auftritt, was zu

```
32  struct proc_dlatch_db_t
33  {
34    Module *module;
35    SigMap sigmap;
36    FfInitVals initvals;
37
38    pool<Cell*> generated_dlatches;
39    dict<Cell*, vector<SigBit>> mux_srcbits;
40    dict<SigBit, pair<Cell*, int>> mux_drivers;
41    dict<SigBit, int> sigusers;
119   struct rule_node_t
120   {
123     SigBit signal, match;
124     vector<int> children;
139   enum tf_node_types_t : int {
140     true_node = 1,
141     false_node = 2
142   };
143
144   idict<rule_node_t, 3> rules_db;
145   dict<int, SigBit> rules_sig;
301   void fixup_muxes()
```

Listing 3.40: Hilfsstruktur (`proc_dlatch_db_t`) aus dem `proc_dlatch`-Pass[5]

einem Latch führt. Unabhängig davon, ob eine Rückspeisung auftritt, werden die `SyncRule`-Aktionen entfernt, nachdem die einzelnen Latch- und nicht-Latchbits in zwei Gruppen zwischengespeichert wurden.

Für die festgestellten Latchbits wird der Rückspeiseweg über potenziell mehrere Multiplexer untersucht. Dabei werden in dem `rule_node_t-struct` in `rules_db` aus Listing 3.40, die jeweiligen Multiplexer-Arten vermerkt. Erfolgt die Rückspeisung über einen `else_signal` Eingang des Multiplexers muss später dem Aktivierungssignal des Latches eine Negationszelle vorgeschaltet werden. Bei dem `when_signal` müssen im Fall einer vorgeschalteten `$pmux`-Zelle, die Steuersignale mit einer `$reduce_or`-Zelle vereinigt werden. Die Steuersignale des aktuellen Multiplexers und der vorgeschalteten Multiplexern müssen abschließend mit einer Und-Verknüpfung in ein neues Steuersignal vereinigt werden. Die Bits, die zu keinem Latch verbunden werden, werden als Verbindung dem Modul hinzugefügt. In diesem Fall werden die Initialisierungswerte des betroffenen Signals gelöscht.

Ist der letzte Multiplexer erreicht, werden die Steuersignale ein letztes mal vereinigt. Das resultierende Signal wird dann zu dem Aktivierungssignal des Latches. Während des Multiplexer-Durchlaufs wurde das Rückspeisesignal in den Multiplexern als undefiniert markiert, wenn laut der `sigusers`-Variable, das Signal nur als Eingang des Latches gilt. Aus diesen Multiplexern werden zum Schluss die undefinierten Signale extrahiert, sodass die Multiplexer keine Bits unnötig verarbeiten müssen.

Erkennung von Flip-Flops

Die verbleibenden `SyncRule`s sind flanken- und pegelgesteuert. Die pegelgesteuerte `SyncRule`s wurden im `proc_arst`-Pass als erkannte Resetsignale erzeugt. Die verbleibenden flankengesteuerte `SyncRule`s deuten auf Flip-Flops hin. Der `proc_dff`-Pass erzeugt aus den `SyncRule`s DFFs. Ein DFF ist ein Speicherelement, das nur zu den positiven oder negativen Flanken eines Signals den

gespeicherten Wert ändert. Zusätzlich gibt es DFFs, die unabhängig vom obengenannten Signal, den Wert des DFFs bei einem Reset asynchron ändern können.

Der `proc_dff`-Pass iteriert durch alle Prozesse des Designs. Aus allen `SyncRule`s werden linke Seiten der Zuweisungen als Kandidaten für ein Speicherelement behandelt. Die Verarbeitung endet, wenn keine linken Seiten der Zuweisungen mehr verbleiben. Für die linke Seite einer Zuweisung werden folgende Werte ermittelt:

- Resetwerte, welche in pegelgesteuerten `SyncRule`s zugewiesen werden. Ein Wert kann konstant sein oder von weiteren Signalen abhängen und variabel sein. Für jeden Wert werden `SyncRule`s ermittelt, in denen sie zugewiesen werden. Daraus werden später die Pegel für einzelne Steuersignale bestimmt.
- Für die flankengesteuerte `SyncRule` werden ein Eingangssignal, ein Steuersignal und die Art der Flanke für den DFF ermittelt.
- Für eine `SyncRule` mit dem globalen Taktsignal wird das Eingangssignal des DFFs ermittelt.
- Für eine `always`-`SyncRule` wird das Eingangssignal des DFFs ermittelt.

Die Zuweisungen, deren Werte übernommen wurden, werden aus den Zuweisungen der `SyncRule`s entfernt. Für die `always`-`SyncRule` wird keine `$dff`-Zelle erzeugt. Das ermittelte Eingangsignal wird einfach mit der linken Seite der Zuweisung der `SyncRule` verbunden. Ist für einen DFF der Resetwert mit dem Ausgang des DFFs gekoppelt, wird ein Multiplexer vorgeschaltet, der entweder das ermittelte Eingangssignal oder das Ausgangssignal des DFFs in den Eingang speist. Die erstellte Zelle ist eine `$dff`-Zelle. Wurde ein konstanter Resetwert ermittelt, wird eine `$adff`-Zelle erzeugt. Bei einem variablen Resetwert erstellt Yosys eine `$aldff`-Zelle und bei einem globalen Takt eine `$ff`-Zelle.

Wenn mehrere `SyncRule`s für den Resetwert festgestellt wurden, wird das Resetsignal mit einem Ungleichheitsvergleich der Resetsignale und der `SyncRule`-Pegeln ermittelt. Da es hierbei weiterhin nur einen Resetwert gibt, wird wieder eine `$adff`-Zelle erzeugt. Es ist auch möglich, dass mehrere Resetwerte für einen DFF gefunden wurden. In diesem Fall wird eine `$dffsr`-Zelle erstellt. In diesem Fall wird eine komplexere Vorbeschaltung benötigt. Für jeden Resetwert werden zuerst die Resetsignale der `SyncRule`s mit dem Pegel Null mit einer `$reduce_or`-Zelle zusammengefasst und ihr Ausgangssignal negiert, damit er zusammen mit den `SyncRule`s mit dem Pegel Eins verarbeitet werden kann. Die Resetsignale der `SyncRule`s mit dem Pegel Eins werden in einer `$reduce_or`-Zelle zusammen mit dem ermittelten Signal der `SyncRule`s mit dem Pegel Null vereinigt. Dieses Signal wird zum neuen Resetsignal. Aus dem Resetwert wird mit einer `$not`-Zelle ein invertierter Resetwert erzeugt. Es werden zwei Multiplexer für die *Set*- und *Clear*-Eingänge erstellt. Für den *Set*-Eingang wird der nicht invertierte Resetwert mit dem `when_signal` verbunden. Das Gleiche geschieht für den *Clear*-Eingang mit dem invertierten Resetwert. Der für diesen Wert ermittelte Resetsignal wird in beiden Multiplexern als das Steuersignal genutzt. Alle Bits der `else_signal`e der Multiplexer haben für den ersten Resetwert den 0-Zustand. Für weitere Resetwerte ist das oben beschriebene Vorgehen gleich mit dem Unterschied, dass mit den `else_signal`en die Ausgangssignale der davor erstellten Multiplexer verbunden werden. Nachdem alle Resetwerte bearbeitet wurden, wird die `$dffsr`-Zelle erstellt und das *Set*-, *Clear*- und Taktsignal mit den jeweiligen Ports verbunden.

Erkennung von Speicherschreibzugriffen

Der proc_memwr-Pass ist der letzte Pass, welcher sich mit der Auflösung von Prozessen beschäftigt. Mittlerweile bestehen die Prozesse nur noch aus SyncRules, in denen keine Aktionen mehr vorhanden sind. Zusätzlich zu den normalen Aktionen existieren noch die MemWriteActions, welche Speicherschreibzugriffe im Prozess abstrahieren. Der letzte Schritt der Prozessauflösung ist daher die Umwandlung der MemWriteActions in $memwr_v2-Zellen. Diese Zelle besitzt eine Port-ID, welche für jeden Speicher separat nummeriert wird. Somit sind diese IDs innerhalb eines Speichers eindeutig, wobei mehrere Zellen von verschiedenen Speichern die gleiche ID haben können. Deswegen werden am Anfang des Passes die bestehende $memwr_v2-Zellen durchlaufen, um die Port-IDs für die $memwr_v2-Zellen, welche innerhalb dieses Passes erzeugt werden, zu vergeben. Die Speicher werden im Rahmen des Memory-Passes näher betrachtet.

Die Prozesse des Designs werden auf der Suche nach MemWriteActions durchgesucht. Die $memwr_v2-Zelle bekommt die nächste Port-ID für ihren Speicher, die Speicher-ID und eine priority_mask, welche in einem Frontend basierend auf der Reihenfolge der Zuweisungen erstellt wurde. Weiter werden die Anzahl der Adressbits, die Wortbreite und der Adress- und Dateneingang übernommen. Im proc_arst-Pass wurde bereits sichergestellt, dass das Aktivierungssignal des Speichers während einem Reset nicht auf den Speicher schreiben wird. Währenddessen wurden Pegelgesteuerte SyncRules für die Resets erstellt. Im proc_dff-Pass wurden die Inhalte der pegelgesteuerten SyncRules in DFF-Resets umgewandelt. Dazu gehört auch das Aktivierungssignal, weswegen davon ausgegangen wird, dass die leeren SyncRules auch den Reset des Aktivierungssignals ausgelöst haben. Deswegen wird einer $memwr_v2-Zelle ein Multiplexer vorgeschaltet. Das Steuersignal ist das Signal der SyncRule. Die Datensignale sind das Aktivierungssignal und eine konstante Null. Die Multiplexer werden so angeschlossen, dass bei einem Reset die Null ausgegeben wird. Der Multiplexerausgang ist der Aktivierungseingang der $memwr_v2-Zelle. Die MemWriteActions können sich nur in folgenden SyncRules befinden:

- always — Die Zelle bekommt keinen Taktgeber.
- posedge — Das SyncRule-Signal ist der Taktgeber der Zelle. Der Schreibzugriff erfolgt bei der steigenden Flanke.
- negedge — Das SyncRule-Signal ist der Taktgeber der Zelle. Der Schreibzugriff erfolgt bei der fallenden Flanke.

Sonstige SyncRules unterstützen in diesem Pass keine MemWriteActions. Zum Schluss werden die MemWriteActions gelöscht. Somit sind die Prozesse vollständig aufgelöst und werden von dem anschließendem proc_clean-Pass gelöscht.

3.3.2.3 Optimierungen

Nach der Analyse des proc-Passes stellt man fest, dass die Unterpässe im Wesentlichen die Lösung ihrer Aufgaben priorisieren und das Design gegebenenfalls in einem nicht optimalen Zustand verlassen. Aus diesem Grund gibt es den opt-Pass, der sich mit der Optimierung des Designs beschäftigt. Damit werden unter Anderem Artefakte des proc-Passes beseitigt.

Der opt-Pass ist ähnlich wie der proc-Pass ein Skriptpass, der Unterpässe ausführt. Der erste ausgeführte Unterpass ist der opt_expr-Pass. Dieser Pass ist in der Lage kombinatorische Zellen zu ersetzen, wenn die Eingänge der Zellen ganz und in bestimmten Fällen teilweise durch Konstanten getrieben werden. In Tabelle 3.5 wird ein Beispiel aus dem Yosys-Handbuch gezeigt, das die Regeln

A-Input	B-Input	Replacement
any	0	0
0	any	0
1	1	1
X/Z	X/Z	X
1	X/Z	X
X/Z	1	X
any	X/Z	0
X/Z	any	0
a	1	a
1	b	b

Tabelle 3.5: Die Regeln zur Ersetzung einer Und-Zelle mit konstanten Eingangsbits[14, S. 77]

mit denen ein 1-Bit Und-Gatter ersetzt werden kann, demonstriert. Es reicht, wenn an einem der Eingänge ein 0-Bit anliegt, sodass der Ausgang immer ein 0-Bit ausgibt. Ist einer der Eingängen ein 1-Bit, kann der andere Eingang unabhängig von seinem Wert weitergeleitet werden. Die Fälle, bei denen beide Eingänge undefiniert sind oder ein Eingang undefiniert und der andere Eingang ein 1-Bit ist, sind laut Wolf die einzigen drei Fälle, bei denen ein undefinierter Wert weitergeleitet werden darf[14, S. 76]. Der Pass ist ausserdem in der Lage simple Ausdrücke umzuschreiben, wenn dadurch die Anzahl der benötigten Gatter niedriger wird.

Als nächstes werden im `opt_merge`-Pass identische Zellen gesucht. Zellen werden als identisch bewertet, wenn sie den gleichen Zellentyp und identische Eingangssignale besitzen. Es besteht die Möglichkeit, die Vereinigung von Multiplexerzellen und DFFs mit *don't care* Bits in ihrem Initialwert, auszuschalten. Der Pass wird mit ausgeschalteter Multiplexerzellenvereinigung ausgeführt, sodass der nachfolgende Pass seine Optimierungen bestmöglich durchführen kann[14, S. 78].

Die weiteren Passes werden in einer Schleife ausgeführt, die erst verlassen wird, wenn ein Durchlauf der Schleife keine Veränderung erbracht hat. Der erste Pass in der Schleife ist der `opt_muxtree`-Pass. Während der Umwandlung der Entscheidungsbäume aus den Prozessen in Multiplexerbäume, werden auf Grund des rekursiven Algorithmus aus dem `proc_mux`-Pass Multiplexer erstellt, welche keinen Mehrwert im Design bieten. Diese Multiplexer entstehen zum Großteil bei Entscheidungsbäumen, die im Frontend aus den Verilog `if`-Anweisungen erzeugt wurden. Es ist auch möglich, dass überflüssige Multiplexer aus der außerhalb von Prozessen definierten Logik entstehen. Ein Beispiel ist der Verilog-Ausdruck `y = a ? (a ? 1 : 2) : 3`[14, S. 77]. `y` wird nie den Wert 2 annehmen können, da dafür `a` gleichzeitig 1 und 0 sein müsste, was widersprüchlich ist. Der `opt_muxtree`-Pass würde die Multiplexer optimieren und eine Logik erzeugen, die äquivalent zum Ausdruck `y = a ? 1 : 3` ist[14, S. 77].

Der `opt_reduce`-Pass führt zwei Optimierungen in einer Schleife durch, bis ein Durchlauf keine Änderungen mehr hervorruft. Die erste Optimierung ist die Zusammenfassung von Bäumen aus Und- und Or-Gattern, wobei duplizierte Eingänge erkannt und eliminiert werden. Als Zweites werden duplizierte Steuersignale bei Multiplexern erkannt und mit `$reduce_or`-Zellen zusammengefasst.

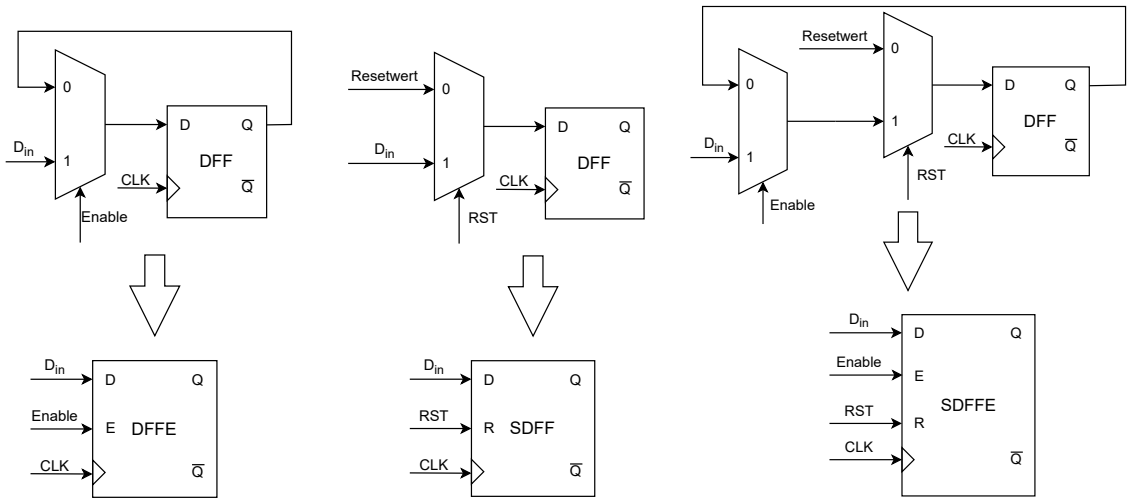
Im `opt_share`-Pass werden Zellen gesucht, auf welche folgende Kriterien zutreffen:

- Die Zellen schließen sich gegenseitig aus, das heißt, dass nur eine der Zellen zu einem Zeitpunkt Wirkung im Rest des Designs zeigt.

- Die Zellen teilen sich ein Eingangssignal.
- Die Zellenausgänge sind mit demselben Multiplexer verbunden.

In diesem Fall können diese Zellen zusammengefasst werden. Die Zelle, welche aus der Zusammenfassung entsteht, bekommt den gemeinsamen Eingang. Der Ausgang der Zelle muss nicht mehr gemultiplext werden. Stattdessen wird der nicht gemeinsame Eingang gemultiplext. Damit können $x - 1$ Zellen gespart werden, wobei x die Anzahl der Zellen ist, welche die obigen Kriterien erfüllen.

Nachfolgend werden im `opt_dff`-Pass DFFs gesucht, welche in andere Typen umgewandelt werden können. Die triviale Fälle der Umwandlungen zeigt Abbildung 3.5. Ein DFF mit einem Vorbereitungseingang (engl. Clock-Enable, CE) wird erkannt, wenn abhängig von einem Steuersignal, das DFF den aktuellen Wert behält oder den Dateneingangswert annimmt. Die Beschaltung eines DFFs, mit welcher die obengenannte Funktion umgesetzt werden kann, sowie die resultierende `$dffe`-Zelle, zeigt Abbildung 3.5a. Dort ist der Datenausgang des DFFs mit seinem Dateneingang über einen Multiplexer zurückgekoppelt. Das Steuersignal des Multiplexers entscheidet, ob das DFF den Wert von D_{in} erhält oder durch die Rückkopplung den aktuellen Wert behält. Ähnlich kann der Dateneingang des DFFs mit einem Multiplexer vorgeschaltet werden ohne einer Rückkopplung des Ausgangssignals. Ein Eingang des Multiplexers ist das D_{in} der Schaltung, während der andere einen Resetwert darstellt. Es handelt sich um einen synchronen Reset, da der Resetwert nur synchron mit dem Takt übernommen wird. Diese Funktionalität setzt eine `$sdff`-Zelle um. Diese Umwandlung zeigt Abbildung 3.5b. Es ist möglich, dass die Beschaltung eines DFFs beide Funktionalitäten implementiert. Für diesen Zweck existiert die `$sdffe`-Zelle, welche beide der obengenannten Funktionalitäten zusammenfasst. Eine Beispiel Beschaltung und die resultierende Zelle wird durch Abbildung 3.5c dargestellt. Zusätzlich entfernt der Pass überflüssige Steuersignale des DFFs oder ersetzt den gesamten DFF mit einer Konstante.



(a) DFF zu DFFE (b) DFF zu SDFF (c) DFF zu SDFFE

Abbildung 3.5: Umwandlung von DFFs und derer Beschaltung in andere DFF-Typen

Der `opt_clean`-Pass räumt das Design auf. Andere Pässe hinterlassen oft nicht mehr benötigte Zellen und Signale, nachdem eine Zelle ersetzt wurde oder Signale woanders hingeführt wurden. Dieser Pass erkennt solche Signale und Zellen, um sie zu entfernen. Zum Abschluss der Schleife wird nochmals der `opt_expr`-Pass ausgeführt. Wenn das Design keine Änderungen aufweist ist der `opt`-Pass fertig, ansonsten fängt der obenbeschriebene Vorgang erneut an. Der `opt`-Pass kann auch

mit der -fast-Option aufgerufen werden. In diesem Fall werden lediglich die Passes opt_expr , opt_merge, opt_dff und opt_clean in einer Schleife ausgeführt, bis keine Änderungen mehr stattfinden.

3.3.2.4 Zustandsautomatenoptimierung

Der fsm-Pass ist ein Skriptpass und führt Unterpässe aus, um Zustandsautomaten zu erkennen, extrahieren, optimieren und abschließend zurück in eine RTL-Beschreibung zu führen. Für die Erkennung sorgt der fsm_detect-Pass, der gefundene Zustandssignale mit dem Attribut \fsm_encoding = "auto" markiert. Bei der Erkennung werden Signale, die dieses Attribut bereits besitzen, ignoriert. Mit dem \fsm_encoding = "none"-Attributwert kann die Verarbeitung als Zustandsautomat verhindert werden. Umgekehrt gilt ein Signal mit dem Attributwert $\neq$ "none" als ein Zustandssignal. Damit kann die Verarbeitung des Signals als Zustandssignal erzwungen werden. Ein Zustandssignal weist folgende Merkmale auf[14, S. 79]:

- Es ist kein Modulausgang.
- Es ist mit dem Ausgang einer $dff oder einer $adff verbunden.
- Der Dateneingang des DFFs ist mit dem Ausgang eines Multiplexerbaumes verbunden. Die äußersten Multiplexer des Baumes müssen an ihren when_signalen und else_signalen entweder Konstanten oder den alten Wert des Zustandssignals aufweisen.
- Das Zustandssignal wird nur im Multiplexerbaum und in Vergleichszellen(wie $eq) genutzt

Der fsm_extract-Pass erzeugt eine $fsm-Zelle, welche den gesamten Zustandsautomaten beschreibt. Dafür werden zuerst die Signale mit dem gesetzten \fsm_encoding-Attribut gesucht. Die ermittelten Signale sind die Zustandssignale des jeweiligen Zustandsautomaten. Das DFF, welches das Zustandssignal speist, wird als Zustandsregister identifiziert. Dabei kann gegebenenfalls der Resetzustand ermittelt werden. Als nächstes werden Zustände und Kontrolleingänge ermittelt, in dem der Multiplexerbaum rekursiv durchlaufen wird. Die Steuersignale der Multiplexer entsprechen den Kontrolleingängen und die when_signale und else_signale an den äußersten Multiplexern den jeweiligen Zuständen. Sind die Zustände keine Konstanten, schlägt der Algorithmus fehl. Ansonsten werden die Kontrollausgänge identifiziert. Kontrollausgänge sind die einzelnen Bits des Zustandssignals. Dazu kommen noch Ausgangssignale der Zellen, welche die Bits des Zustandssignals mit einer Konstanten vergleichen. Zuletzt wird ein Zustandsübergangsdiagramm mit Hilfe der statischen Auswertung der Multiplexer erstellt. Mit den gesammelten Informationen kann die $fsm-Zelle erzeugt werden, mit der alle restlichen fsm-Unterpasses arbeiten.

Mit dem fsm_opt-Pass wird der Zustandsautomat optimiert. Ungenutzte Kontrollausgänge werden entfernt basierend auf dem \unused_bits-Attribut, das vom opt_clean-Pass gesetzt wird. Weiter werden Kontrolleingänge die von derselben Quelle getrieben werden zusammengefasst. Ist ein Kontrollausgang mit einem Kontrolleingang verbunden, wird die Verbindung entfernt und die Logik stattdessen im Zustandsübergangsdiagramm eingebettet. Nachfolgend werden für Zustandsübergänge, die zu dem gleichen Zustand führen, die Kontrolleingänge verglichen. Wird ein einbittiger Unterschied festgestellt, können die Übergänge zusammengefasst werden. Das unterschiedliche Bit wird aus dem Übergang entfernt(*don't care*). Abschließend werden die konstanten oder ungenutzten Kontrolleingänge entfernt und das Zustandsübergangsdiagramm entsprechend angepasst. Hiernach wird der opt_clean-Pass und danach nochmal der fsm_opt-Pass ausgeführt.

Der fsm_recode-Pass konvertiert die Kodierung der Zustände. Aktuell wird die einfache binäre oder die *onehot* Kodierung unterstützt. Bei der *onehot* Kodierung muss die Größe des Zustandssignals gleich der Anzahl der Zustände sein. Ein Bit entspricht einem Zustand und die einzige Eins im Vektor gibt den aktuellen Zustand an. Das hat zu Folge, dass mehr DFFs für das Zustandssignal benötigt werden. Dafür ist die Berechnung des nächsten Zustandes simpler, da die Zustände nicht dekodiert werden müssen[1].

Die fsm_info- und fsm_export-Passes zeigen Informationen zm Zustandsautomaten und exportieren ihn im KISS2-Format. Der optimierte Zustandsautomat wird abschließend mit fsm_map-Pass zurück in eine RTL-Darstellung umgewandelt.

3.3.2.5 Umwandlung der generischen Speicher

Generische Speicher wurden das erste Mal im Unterkapitel 3.2.2 eingeführt. Dort wurden die RTLIL-Typen Memory und MemWriteAction beschrieben. Tatsächlich existieren zwei Möglichkeiten die Speicher im Design darzustellen. Das RTLIL::Memory-Objekt speichert einen Identifikator des Speichers und die allgemeinen Informationen wie die Wortbreite, Anzahl der Wörter und das offset der Anfangsadresse. Um mit dem Speicher interagieren zu können, werden die $memwr_v2, $memrd_v2 und $meminit_v2 Zellen benötigt. Diese Zellen stellen die Ports des Speichers dar und gehören zusammen mit den RTLIL::MemWriteActions zur ersten Möglichkeit der Speicherdarstellung, die standardmäßig von den Frontends erzeugt wird. Die zweite Darstellungsmöglichkeit ist die $mem_v2-Zelle, welche die oben genannten Zellen und Objekte zusammenfasst.

Alle aufgeführten Zellen beinhalten den Identifikator des RTLIL::Memory-Objekts zu dem sie gehören. Ein Leseport kann sowohl synchron als auch asynchron arbeiten, allerdings können Resetwerte nur bei synchronen Leseports funktionieren. Unterstützt werden sowohl synchrone als auch asynchrone Resets, wovon beide einen synchronen Leseport voraussetzen. Die Schreibports können ebenfalls synchron und asynchron arbeiten. Zudem besitzen sie jeweils eine Port-ID, welche für die Priorisierung der Schreibezugriffe verwendet wird. Ein Schreibport kann nur priorisiert werden, wenn der andere Port eine niedrigere Port-ID hat. In einer Prioritätsmaske gibt das Bit an der Port-ID-Stelle an, über welchen Ports der aktuelle Port priorisiert wird.

Die Umwandlung beginnt mit der Durchführung von einfachen Optimierungen im opt_mem-Pass. Zum Beispiel werden dort Schreibports gelöscht, deren Aktivierungssignale nur Nullen aufweisen und somit nie aktiv werden. Im opt_mem_priority werden Prioritätsbeziehungen entfernt, wenn die zugehörigen Schreibeports nie gleichzeitig auf die gleiche Adresse schreiben oder gar aktiv werden. opt_mem_feedback löscht Rückkopplungen eines asynchronen Leseports auf einen Schreibeport. In diesen Fällen wird die Rückkopplung mit einem neuen Kriterium für das Aktivierungssignal ersetzt. Eine $bmux-Zelle mit konstantem Dateneingang wird in memory_bmux2rom zu einem ROM umgewandelt.

Der memory_dff-Pass sucht asynchrone Leseports, die ihren Wert einem DFF zuweisen. In diesem Fall wird der Leseport zu einem synchronen Leseport umgewandelt und der DFF wird konsumiert.

Der memory_share-Pass fasst mehrere Ports zusammen, wenn:

- mehrere Schreibports in dieselbe Adresse schreiben. Der resultierende Port weist dann eine komplexere Datensignal- und Aktivierungssignalbeschaltung auf.
- mehrere Schreib- oder Leseports in benachbarte Adressen schreiben oder aus ihnen lesen. Die Ports werden dann ausgebreitet.

- mehrere Schreibeports nie zu gleichen Zeit schreiben.

Aus dem `memory_share`-Pass entnimmt man, dass die Ports ausgebreitet werden können. Diese Ports emulieren dann den gleichzeitigen Zugriff auf mehrere Adressen. Mit dem `opt_mem_widen`-Pass wird die Breite des `RTLIL::Memory`-Objekts vergrößert, wenn alle Ports breiter sind. Sobald mindestens ein Port die Breite des `RTLIL::Memory`-Objekts aufweist, ist das Ziel des Passes erreicht.

Mit dem `memory_collect`-Pass werden die einzelnen Ports zu einer `$mem_v2`-Zelle zusammengefasst, die alle Ports in einer Zelle vereinigt. Abschließend erzeugt `memory_map` aus den `$mem_v2`-Zellen DFFs und die zugehörigen Adressdekodierer.

3.3.2.6 Technologiemapping

Das bislang verarbeitete Design beinhaltet keine Prozesse mehr und ist gegebenenfalls logisch optimiert. Als letzten Schritt der Synthese muss das Design auf die Zielhardware angepasst werden. Dafür werden die generischen Zellen mit den in der Hardware verfügbaren Zellen ersetzt.

Die Pässe arbeiten großenteils auf der internen Zellenbibliothek von Yosys. Diese Bibliothek beinhaltet allerdings generische Zellen, die in der Regel in der Zielhardware nicht ohne weiteres umsetzbar sind. Als Beispiel verfügt Yosys über Addierer, die eine beliebige Breite annehmen können. In der Praxis werden Addierer für Signale bestimmter Größe bereitgestellt. Auch sind nicht immer alle DFF-Arten verfügbar. Hieraus folgt, dass für die Implementierung eines Designs in Hardware die generischen Zellen in, in der Hardware verfügbare, Zellen umgewandelt werden müssen. Diesen Vorgang nennt man Technologiemapping.

Im `synth`-Pass werden drei Technologiemapping-Pässe genutzt. Der `techmap`-Pass nutzt RTLIL- oder Verilog-Beschreibungen von Zellen. Dort werden Zellentypen angegeben, die mit der Implementierung ersetzt werden können. Yosys bietet eine eigene Verilog-Beschriebung für den Pass. Die Zellen werden dann in mehrere immer einfachere Zellen umgewandelt. Aus diesem Grund empfiehlt das Yosys-Handbuch, eigene Ersetzungsregeln zu erstellen, um gegebenenfalls die Zellen, für die es möglich ist, in Zellen der Zielarchitektur abzubilden, bevor durch eine Vereinfachung Informationen verloren gehen und ein optimales Mapping nicht mehr möglich ist. Später können wieder die internen Definitionen genutzt werden.

Die `flowmap`- und `abc`-Passes implementieren komplexere Algorithmen, deren Erklärung über den Rahmen der Arbeit hinaus geht und deshalb nicht behandelt werden, aber zur Vollständigkeit benannt sind.

3.3.3 Zusammenfassung der Synthese

Durch die Analyse der Yosys-Synthese wurde deutlich, dass die ersten Syntheseschritte bereits im Frontend stattfinden. Dort werden nämlich die ersten Zellen aus den Anweisungen von außerhalb der Prozessen erzeugt. Und-Verknüpfungen werden zum Beispiel direkt beim Einlesen in generische Und-Zellen umgewandelt. Der Vorteil dieser Vorgehensweise besteht darin, dass RTLIL keine Möglichkeit für die Speicherung von Ausdrücken benötigt. Der wichtigste Teil der Synthese ist die Auflösung der Prozesse. Einerseits sind Prozesse nicht in Hardware darstellbar, dafür ermöglichen sie eine übersichtliche Modellierung von Funktionalitäten, wie Multiplexer oder DFFs, welche ohne Prozesse nur mühsam durch gezielte Angabe von Zellen im Entwurf implementierbar wären.

Optimierungen stellen dabei einen unentbehrlichen Teil der Synthese dar. Sie dienen der Übersichtlichkeit bei der Ausführung anderer Pässe und vor allem der Schonung der Hardwareressourcen. Allerdings sollte auch ohne den Optimierungsteil eine erfolgreiche Synthese theoretisch möglich sein. Weiter können Speicher entweder in einer abstrakten Darstellung bleiben oder in Zellen umgewandelt werden. Die erstere Möglichkeit ist nur sinnvoll, wenn die weiteren Werkzeuge, welche die Netzliste konsumieren, mit dieser Darstellung umgehen können.

Zum Schluss kann ohne Technologiemapping nicht von einer vollständigen Synthese gesprochen werden. Das Technologiemapping erweist sich, hinter den Optimierungen, als die komplizierteste Synthesephase. Hier müssen wichtige Entscheidungen getroffen werden. Die Logik muss der existierenden Hardware zugeteilt werden. Dabei sind manche Ressourcen limitiert. Es kann passieren, dass durch eine falsche Entscheidung die Synthese fehlschlägt, wenn die Ressourcen nicht gut genug zugeteilt wurden. Zu diesen Zwecken nutzen die Technologiemappingpässe eine Mehrzahl an Algorithmen, die der Ressourcenverteilung beistehen sollten. Der `flowmap`-Pass implementiert beispielsweise den FlowMap-Algorithmus[11].

3.4 Netzlistenbackend

Nach der Durchführung einer Synthese ist die Verarbeitung eines Verilogentwurfs noch nicht abgeschlossen. Die Zuordnung der Funktionalitäten, die für den Entwurf benötigt werden, zu den in der Hardware verfügbaren Ressourcen ist logisch erfolgt. Soll ein synthetisiertes Design in der reellen Hardware eingesetzt werden, so muss im Weiteren die physische Zuordnung der Komponenten erfolgen. Diesen Zweck erfüllen sogenannten "place and route"-Werkzeuge. Sie versuchen die Einzelteile des synthetisierten Designs in der Zielhardware so zu platzieren, dass das Timing der Signale möglichst klein wird. Im Unterkapitel 2.4 wurde ein Netzlisten-Format, das sich als eine Eingabe in die "place and route"-Werkzeuge gut eignet, eingeführt. Im Folgenden soll der Vorgang einer Ausgabe des Designs als Netzliste erläutert werden.

Das Netzlistenbackend wird in der Datei `backends/json.cc` definiert. Dort sind zwei Typen `JsonPass` und `JsonBackend` enthalten, welche die Yosys-Befehle `json` und `write_json` umsetzen. Im Wesentlichen wird in den Pässen die Ausgabedatei geöffnet und zur weiteren Verarbeitung mit dem `JsonWriter` Typ weitergegeben. Ein `JsonWriter`-Objekt beinhaltet Methoden, die für die Ausgabe eines Designs zuständig sind:

- `write_design` iteriert durch alle Module des Designs und gibt sie an die `write_module`-Methode weiter.
- `write_module` iteriert durch alle Ports, Zellen, Speicher und Signale und gibt sie in geeigneter Form aus. Parameter und Attribute des Moduls und seiner Inhalte werden mit der `write_parameters`-Methode ausgegeben.
- `write_parameters` schreibt die Namen der Attribute und Parameter in die Datei. Die zugehörigen Werte werden mit der `write_parameter_value`-Methode ausgegeben.
- `write_parameter_value` schreibt den Wert eines Attributs oder Parameters in die Datei.

Zusätzlich überprüft die `write_module`-Methode, ob die Module Prozesse beinhalten. Eine Netzliste kann keine Prozesse beinhalten, weshalb in so einem Fall ein Fehler ausgegeben wird. Bevor eine Netzliste ausgegeben werden kann, muss das `ProcPass` ausgeführt worden sein, wenn der Ursprungsentwurf Prozesse beinhaltet hat.

4 Zusammenfassung

Das Ziel dieser Arbeit war die Untersuchung der Hardwaresynthese. In der Arbeit wurde die Implementierung von Yosys untersucht. Der Grundaufbau des Programms bietet gute Möglichkeiten zur Erweiterung an. Damit lässt sich Yosys für die Synthese für andere Architekturen anpassen. Dabei können die bestehenden Teile von Yosys wiederverwendet werden. Durch die Aufteilung in Pässe sind einzelne Prozesse in sinnvolle Einheiten unterteilt, welche eine bestimmte Aufgabe erfüllen. Das vereinfacht das allgemeine Verständnis des Programms.

Die RTLIL-Datenstruktur ermöglicht die Zwischenspeicherung des Zustands des Designs zu beliebigen Phasen der Synthese. Auch wenn die Synthese direkt auf dem AST durchführbar wäre, bietet RTLIL Funktionen, mit denen die Erfüllung der Aufgaben der Pässe vereinfacht werden kann. Die Untersuchung der Datenstruktur hat gezeigt, dass die Verilog-Prozesse nicht genau in RTLIL abgebildet werden. Die Reihenfolge der Aktionen bleibt in RTLIL ohne weiteres nicht erhalten. Es ist dennoch möglich die Prozesse funktional getreu abzubilden. Diese Aufgabe übernimmt bereits das AST-Frontend. Gelöst wird dies durch die Erstellung von zwischen Signalen in allen Hierarchieebenen der Entscheidungsbäume.

In dieser Arbeit wurde der Fokus auf die Auflösung der Prozesse in einzelne Zellen gelegt. Die durch Prozesse darstellbare Logik lässt sich mit dem `proc`-Pass vollständig umwandeln. Es empfiehlt sich die Ergebnisse der Auflösung der Prozesse durch die Ausführung des `opt`-Passes zu optimieren, da der `proc`-Pass keine optimalen Ergebnisse liefert. Im Rahmen dieser Arbeit wurden die Optimierungen nur oberflächlich behandelt. Dies könnte in weiteren Untersuchungen von Yosys weiter vertieft werden. Die Zellen für die Lese- und Schreibzugriffe auf die Speicher wurden eingeführt. Die genaue Analyse der Umwandlung der Speicher würde allerdings über den Rahmen dieser Arbeit hinausgehen.

Die Analyse des Netzlisten-Backends hat gezeigt, dass die Erstellung einer Netzliste vergleichsweise trivial ist. Die Schwierigkeit besteht darin, das Design in einen netzlistentauglichen Zustand zu bringen.

Die Hauptaufgaben der Synthese sind die Umwandlung der Prozesse in eine RTL-Darstellung und das Technologiemapping. Im Verlauf der Analyse wurde klar, dass im Rahmen der Arbeit die Synthese nur teilweise untersucht werden kann. Dennoch kann die Arbeit als eine Einführung in Yosys gesehen werden, mit der eine Grundlage für weitere Untersuchungen gelegt wird. Mit dieser Arbeit sollte es möglich sein, die Einarbeitungszeit in Yosys wesentlich zu verkürzen.

Literaturverzeichnis

[1] Encoding the states of a finite state machine in VHDL - technical articles. URL: https://www.allaboutcircuits.com/technical-articles/encoding-the-states-of-a-finite-state-machine-vhdl/.

[2] yosys/driver.cc at 6f9602b4cfedc6441f55272f6c025d64620cd24f · YosysHQ/yosys. URL: https://github.com/YosysHQ/yosys/blob/6f9602b4cfedc6441f55272f6c025d64620cd24f/kernel/driver.cc.

[3] yosys/plugin.cc at 6f9602b4cfedc6441f55272f6c025d64620cd24f · YosysHQ/yosys. URL: https://github.com/YosysHQ/yosys/blob/6f9602b4cfedc6441f55272f6c025d64620cd24f/passes/cmds/plugin.cc.

[4] yosys/proc.cc at 6f9602b4cfedc6441f55272f6c025d64620cd24f · YosysHQ/yosys. URL: https://github.com/YosysHQ/yosys/blob/6f9602b4cfedc6441f55272f6c025d64620cd24f/passes/proc/proc.cc.

[5] yosys/proc_dlatch.cc at 6f9602b4cfedc6441f55272f6c025d64620cd24f · YosysHQ/yosys. URL: https://github.com/YosysHQ/yosys/blob/6f9602b4cfedc6441f55272f6c025d64620cd24f/passes/proc/proc_dlatch.cc.

[6] yosys/proc_mux.cc at 6f9602b4cfedc6441f55272f6c025d64620cd24f · YosysHQ/yosys. URL: https://github.com/YosysHQ/yosys/blob/6f9602b4cfedc6441f55272f6c025d64620cd24f/passes/proc/proc_mux.cc.

[7] yosys/register.cc at 6f9602b4cfedc6441f55272f6c025d64620cd24f · YosysHQ/yosys. URL: https://github.com/YosysHQ/yosys/blob/6f9602b4cfedc6441f55272f6c025d64620cd24f/kernel/register.cc.

[8] yosys/register.h at 6f9602b4cfedc6441f55272f6c025d64620cd24f · YosysHQ/yosys. URL: https://github.com/YosysHQ/yosys/blob/6f9602b4cfedc6441f55272f6c025d64620cd24f/kernel/register.h.

[9] yosys/rtlil.h at 6f9602b4cfedc6441f55272f6c025d64620cd24f · YosysHQ/yosys. URL: https://github.com/YosysHQ/yosys/blob/6f9602b4cfedc6441f55272f6c025d64620cd24f/kernel/rtlil.h.

[10] yosys/yosys.cc at 6f9602b4cfedc6441f55272f6c025d64620cd24f · YosysHQ/yosys. URL: https://github.com/YosysHQ/yosys/blob/6f9602b4cfedc6441f55272f6c025d64620cd24f/kernel/yosys.cc.

[11] J. Cong and Yuzheng Ding. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. 13(1):1–12. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. doi:10.1109/43.273754.

[12] J. Cong and Yuzheng Ding. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. 13(1):1–12. URL: `http://ieeexplore.ieee.org/document/273754/`, `doi:10.1109/43.273754`.

[13] Christopher Parnow. Analyse der yosys prozesse, die verilog in die interne datenstruktur RTLIL konvertieren. URL: `https://epb.bibl.th-koeln.de/frontdoor/index/index/docId/2034`, `doi:10.57683/EPUB-2034`.

[14] Clifford Wolf. Design and implementation of the yosys open synthesis suite.

5 Anhang

5.1 SigMap

Die `SigMap` ist eine sogenannte *union-find* oder *merge-find* Datenstruktur. Die Daten werden in baumförmigen Graphen gegliedert. Im Fall der `SigMap` entspricht ein Knoten einem Signal im Design. Verbindungen zwischen Knoten, bedeuten, dass die Signale irgendwo verbunden sind. Das heißt nicht zwingend, dass sie mit eine `connect`-Anweisung verbunden sind. Die Verbindungen können über mehrere Signale hinweg bestehen.

An der Spitze des Baumes befindet sich der Hauptknoten. Das Signal aus dem Hauptknoten ist der Repräsentant der Gruppe von Signalen, die in diesem Baum existieren. Die Wahl des Repräsentanten hängt im Wesentlichem von der Reihenfolge der Definitionen in RTLIL ab.

Die Datenstruktur wird mit dem Modul initialisiert. Dabei werden vorhandene Verbindungen untersucht und die Bäume aufgebaut. Mit dieser Struktur ist es möglich zu prüfen, ob zwei Signale im Modul miteinander verbunden sind.

5.2 Synthese-Tutorial

Das Ziel dieses Tutorials ist die Vertiefung des gesammelten Wissens durch die selbstständige Durchführung einer Yosys-Synthese. Nach jedem Befehl folgen unmittelbar die Ausgaben von Yosys. Zur Veranschaulichung wird der Zustand des Designs mittels von Yosys erstellten Grafiken nach den jeweiligen Synthesephasen dargestellt. Die Grafiken werden mit dem Yosys-Befehl: `show -notitle -format svg` erzeugt. Das Listing 5.1 zeigt einen beispielhaften Verilog-Entwurf, der als Basis für die Durchführung der Synthese dienen soll.

Der Entwurf implementiert eine Rollosteuerung. Es existiert ein Taktsignal und ein Resetsignal. `sensor_oben` und `sensor_unten` geben die Information, ob das Rollo die obere oder untere Lichtschranke erreicht hat. `btn_oben`, `btn_unten` und `stop` steuern das Rollo, in dem es nach oben oder nach unten geschickt oder gestoppt wird. `nach_oben` und `nach_unten` sind die Steuersignale, mit denen die Motoren gesteuert werden. Das Signal `sensor` gibt an, ob einer der Sensoren aktiv ist. Im Entwurf gibt es 5 Zustände:

- stop(000) — Rollo steht
- nach_oben(001) — Rollo fährt nach oben
- nach_unten(010) — Rollo fährt nach unten
- oben(011) — Rollo ist oben
- unten(100) — Rollo ist unten

```
1  module rollosteuerung (
2    input clk,
3    input rst,
4    input sensor_oben,
5    input sensor_unten,
6    input btn_oben,
7    input btn_unten,
8    input stop,
9    output reg nach_oben,
10   output reg nach_unten
11 );
12 wire sensor = sensor_oben || sensor_unten;
13 reg [2:0] state;
14 always @(posedge clk) begin
15   if(rst) state <= 3'b000;
16   else case (state)
17     //stop
18     3'b000: if(btn_oben && !sensor_oben) state <= 3'b001;
19             else if(btn_unten && !sensor_unten) state <= 3'b010;
20             else state <= 3'b000;
21     //nach_oben
22     3'b001: if(btn_unten) state <= 3'b010;
23             else if(sensor_oben) state <= 3'b011;
24             else if(stop) state <= 3'b000;
25             else state <= 3'b001;
26     //nach_unten
27     3'b010: if(btn_oben) state <= 3'b001;
28             else if(sensor_unten) state <= 3'b100;
29             else if(stop) state <= 3'b000;
30             else state <= 3'b010;
31     //oben
32     3'b011: if(btn_unten) state <= 3'b010;
33             else if(!sensor) state <= 3'b000;
34             else state <= 3'b011;
35     //unten
36     3'b100: if(btn_oben) state <= 3'b001;
37             else if(!sensor) state <= 3'b000;
38             else state <= 3'b100;
39     default: state <= 3'b000;
40   endcase
41 end
42 always @(posedge clk) begin
43   case (state)
44     3'b000: {nach_oben, nach_unten} <= 2'b00;
45     3'b001: {nach_oben, nach_unten} <= 2'b10;
46     3'b010: {nach_oben, nach_unten} <= 2'b01;
47     3'b011: {nach_oben, nach_unten} <= 2'b00;
48     3'b100: {nach_oben, nach_unten} <= 2'b00;
49     default: {nach_oben, nach_unten} <= 2'b00;
50   endcase
51 end
52 endmodule
```

Listing 5.1: Beispiel Verilog-Modul für eine Rollosteuerung

Das Rollo darf nur weiter nach oben oder unten fahren, wenn es nicht bereits dort ist. Während des Fahrens kann das Rollo mit stop gestoppt werden oder mit einem der anderen Buttons in die andere Richtung geschickt werden. Erkennt ein Sensor, dass das Rollo oben oder unten ist, bleibt es stehen.

Der erste Prozess sorgt für die Zustandsübergänge, währen der zweite die Steuersignale setzt.

Zuerst muss der Entwurf eingelesen werden. Der Befehl zum Einlesen lautet:

```
yosys> read_verilog rollosteuerung.v
1. Executing Verilog-2005 frontend: rollosteuerung.v
Parsing Verilog input from ‘rollosteuerung.v’ to AST
   representation.
Generating RTLIL representation for module ‘\rollosteuerung’.
Successfully finished Verilog frontend.
```

Das damit entstandene Design zeigt die Abbildung 5.1.

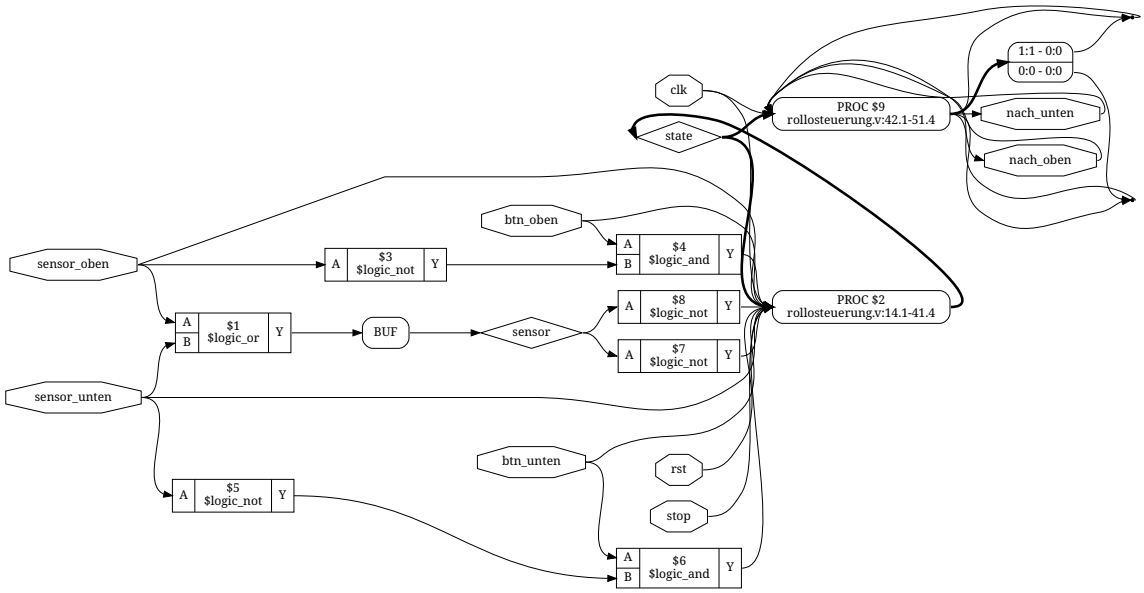


Abbildung 5.1: Die Netzliste nach dem Einlesen des Moduls

Die Prozesse werden aufgeräumt bevor sie umgewandelt werden. Dafür wird zuerst der `proc_clean` -Pass ausgeführt. Dafür muss der folgende Befehl eingegeben werden:

```
yosys> proc_clean

2. Executing PROC_CLEAN pass (remove empty switches from decision
    trees).
Cleaned up 0 empty switches.
```

Der `proc_rmdead`-Pass entfernt wirkungslose Zweige der Entscheidungsbäume und wird mit folgendem Befehl ausgeführt:

```
yosys> proc_rmdead

3. Executing PROC_RMDEAD pass (remove dead branches from decision
    trees).
```

```
Marked 1 switch rules as full_case in process
   $proc$rollosteuerung.v:42$9 in module rollosteuerung.
Marked 14 switch rules as full_case in process
   $proc$rollosteuerung.v:14$2 in module rollosteuerung.
Removed a total of 0 dead cases.
```

Der proc_prune-Pass entfernt wirkungslose Zuweisungen und wird mit folgendem Befehl ausgeführt:

```
yosys> proc_prune

4. Executing PROC_PRUNE pass (remove redundant assignments in
   processes).
Removed 3 redundant assignments.
Promoted 0 assignments to connections.
```

Da das Design über keine ROM-artige Prozesse verfügt, wird der proc_init-Pass übersprungen. Auch der Reset ist synchron gestaltet, was bedeutet, dass der proc_arst-Pass nicht benötigt wird. Mit dem proc_mux-Pass werden nun die switch-case Strukturen aufgelöst und Multiplexer generiert. Dafür soll der Pass mit dem folgendem Befehl ausgeführt werden:

```
yosys> proc_mux

5. Executing PROC_MUX pass (convert decision trees to
   multiplexers).
Creating decoders for process `\rollosteuerung.
   $proc$rollosteuerung.v:42$9'.
        1/1: { $0\nach_oben[0:0] $0\nach_unten[0:0] }
Creating decoders for process `\rollosteuerung.
   $proc$rollosteuerung.v:14$2'.
        1/1: $0\state[2:0]
```

Abbildung 5.2 zeigt den Zustand des Designs nach dem generieren der Multiplexer aus den Prozessen.

Mit dem proc_dff werden die Prozesse aufgelöst und DFFs erstellt. Der folgende Befehl soll ausgeführt werden um die DFFs zu erzeugen:

```
yosys> proc_dff

6. Executing PROC_DFF pass (convert process syncs to FFs).
Creating register for signal `\rollosteuerung.\nach_oben' using
   process `\rollosteuerung.$proc$rollosteuerung.v:42$9'.
        created $dff cell `$procdff$62' with positive edge clock.
Creating register for signal `\rollosteuerung.\nach_unten' using
   process `\rollosteuerung.$proc$rollosteuerung.v:42$9'.
        created $dff cell `$procdff$63' with positive edge clock.
Creating register for signal `\rollosteuerung.\state' using
   process `\rollosteuerung.$proc$rollosteuerung.v:14$2'.
        created $dff cell `$procdff$64' with positive edge clock.
```

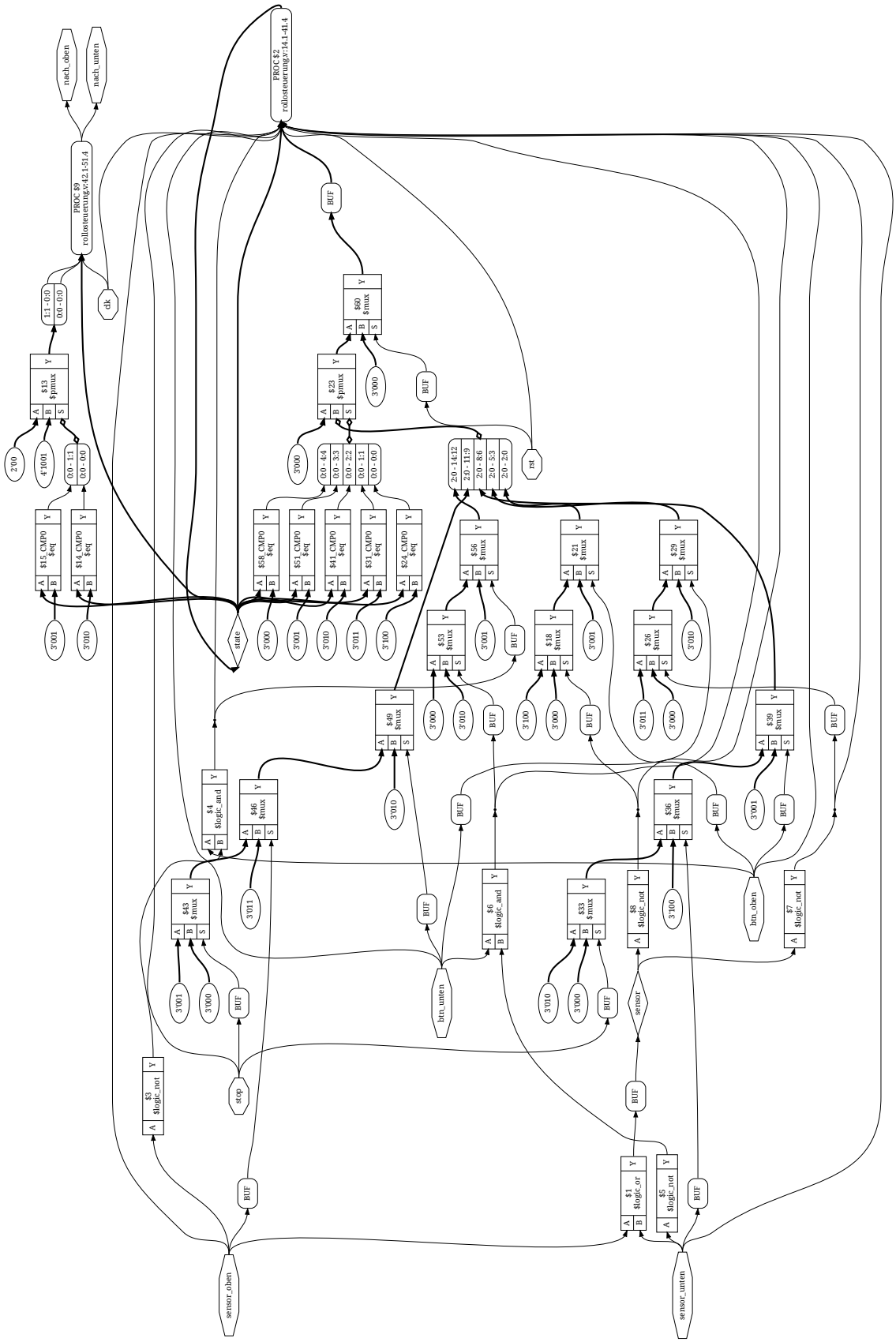


Abbildung 5.2: Die Netzliste nach dem proc_mux-Pass

Es soll der übersichtlichkeits halber direkt im Anschluss der `proc_clean`-Pass ausgeführt werden. Dafür wird der folgende Befehl ausgeführt:

```
yosys> proc_clean

7. Executing PROC_CLEAN pass (remove empty switches from decision
    trees).
Found and cleaned up 1 empty switch in `\rollosteuerung.
   $proc$rollosteuerung.v:42$9'.
Removing empty process `rollosteuerung.$proc$rollosteuerung.v:42
   $9'.
Found and cleaned up 14 empty switches in `\rollosteuerung.
   $proc$rollosteuerung.v:14$2'.
Removing empty process `rollosteuerung.$proc$rollosteuerung.v:14
   $2'.
Cleaned up 15 empty switches.
```

Abbildung 5.3 zeigt den Zustand des Designs nach der Erzeugung der DFFs.

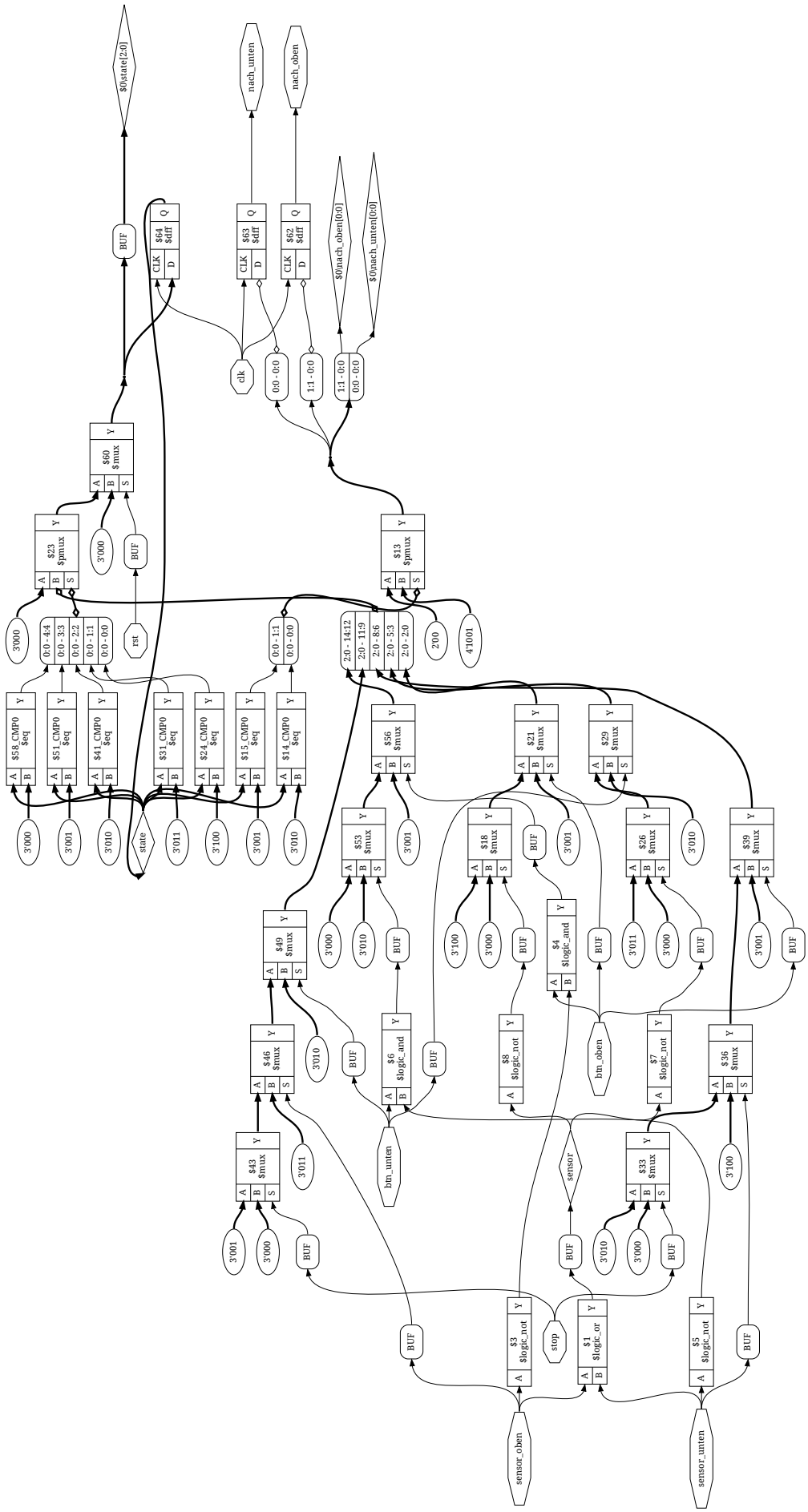


Abbildung 5.3: Die Netzliste nach dem `proc_dff`-Pass

Unser Entwurf besitzt keinen Speicher, also ist der memory-Pass überflüssig. Bevor das Technologiemapping durchgeführt wird, wird das Design mit dem opt-Pass optimiert. Dies ist nicht unbedingt notwendig, allerdings kann es die Größe des Designs verringern. Führen Sie die folgenden Befehle aus. Alternativ kann auch der gesamte opt-Pass mit dem Befehl opt ausgeführt werden mit der selben Wirkung.

```
yosys> opt_expr

8. Executing OPT_EXPR pass (perform const folding).
Optimizing module rollosteuerung.

yosys> opt_merge -nomux

9. Executing OPT_MERGE pass (detect identical cells).
Finding identical cells in module `\rollosteuerung'.
Removed a total of 3 cells.

yosys> opt_muxtree

10. Executing OPT_MUXTREE pass (detect dead branches in mux trees
   ).
Running muxtree optimizer on module \rollosteuerung..
        Creating internal representation of mux trees.
        Evaluating internal representation of mux trees.
        Analyzing evaluation results.
Removed 0 multiplexer ports.

yosys> opt_reduce

11. Executing OPT_REDUCE pass (consolidate $*mux and $reduce_*
   inputs).
        Optimizing cells in module \rollosteuerung.
Performed a total of 0 changes.

yosys> opt_merge

12. Executing OPT_MERGE pass (detect identical cells).
Finding identical cells in module `\rollosteuerung'.
Removed a total of 0 cells.

yosys> opt_dff -nodffe -nosdff

13. Executing OPT_DFF pass (perform DFF optimizations).

yosys> opt_clean

14. Executing OPT_CLEAN pass (remove unused cells and wires).
```

```
Finding unused cells or wires in module \rollosteuerung..
Removed 1 unused cells and 20 unused wires.

yosys> opt_expr

15. Executing OPT_EXPR pass (perform const folding).
Optimizing module rollosteuerung.
```

Wie man sieht, konnten bereits die opt_expr und opt_merge-Pässe das Design optimieren. Abbildung 5.4 zeigt den Zustand des Designs nach Durchlauf des opt-Passes.

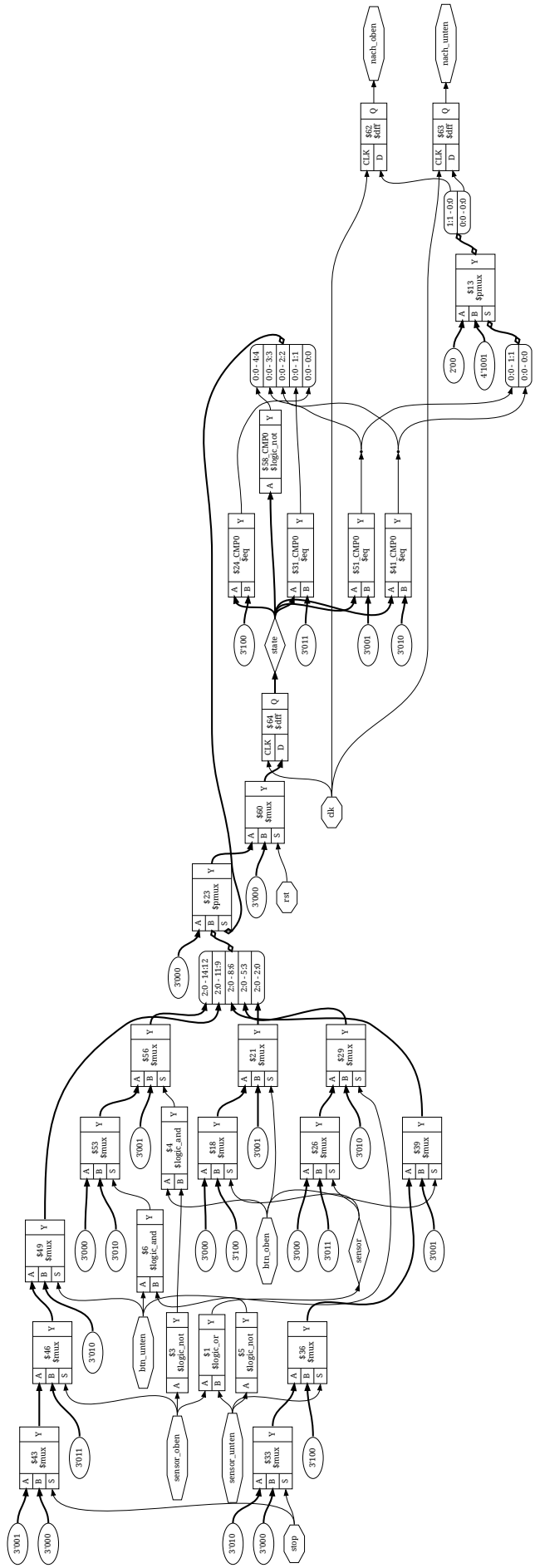


Abbildung 5.4: Die Netzliste nach dem opt-Pass

Nun kann der Zustandsautomat optimiert werden. Dafür wird der `fsm`-Pass ausgeführt. Mit den `fsm_detect` und `fsm_extract` Pässen wird die Logik temporär durch eine `$fsm`-Zelle dargestellt. Führen Sie die folgenden Befehle aus:

```
        yosys> fsm_detect

16. Executing FSM_DETECT pass (finding FSMs in design).
Found FSM state register rollosteuerung.state.

yosys> fsm_extract

17. Executing FSM_EXTRACT pass (extracting FSM from design).
Extracting FSM `\state' from module `\rollosteuerung'.
  found $dff cell for state register: $procdff$64
  root of input selection tree: $0\state[2:0]
  found reset state: 3'000 (guessed from mux tree)
  found ctrl input: \rst
  found ctrl input: $procmux$24_CMP
  found ctrl input: $procmux$31_CMP
  found ctrl input: $procmux$14_CMP
  found ctrl input: $procmux$15_CMP
  found ctrl input: $procmux$58_CMP
  found state code: 3'000
  found ctrl input: \btn_oben
  found ctrl input: \sensor
  found state code: 3'100
  found state code: 3'001
  found ctrl input: \btn_unten
  found state code: 3'011
  found state code: 3'010
  found ctrl input: \sensor_unten
  found ctrl input: \stop
  found ctrl input: \sensor_oben
  found ctrl input: $logic_and$rollosteuerung.v:18$4_Y
  found ctrl input: $logic_and$rollosteuerung.v:19$6_Y
  found ctrl output: $procmux$24_CMP
  found ctrl output: $procmux$31_CMP
  found ctrl output: $procmux$14_CMP
  found ctrl output: $procmux$15_CMP
  found ctrl output: $procmux$58_CMP
  ctrl inputs: { $logic_and$rollosteuerung.v:19$6_Y
     $logic_and$rollosteuerung.v:18$4_Y \sensor \stop \btn_unten
     \btn_oben \sensor_unten \sensor_oben \rst }
  ctrl outputs: { $procmux$58_CMP $procmux$31_CMP $procmux$24_CMP
      $procmux$15_CMP $procmux$14_CMP $0\state[2:0] }
  transition:      3'000 9'00------0 ->      3'000 8'10000000
  transition:      3'000 9'10------0 ->      3'010 8'10000010
```

```
  transition:      3'000 9'-1------0 ->      3'001 8'10000001
  transition:      3'000 9'--------1 ->      3'000 8'10000000
  transition:      3'100 9'--0--0--0 ->      3'000 8'00100000
  transition:      3'100 9'--1--0--0 ->      3'100 8'00100100
  transition:      3'100 9'-----1--0 ->      3'001 8'00100001
  transition:      3'100 9'--------1 ->      3'000 8'00100000
  transition:      3'010 9'---0-00-0 ->      3'010 8'00001010
  transition:      3'010 9'---1-00-0 ->      3'000 8'00001000
  transition:      3'010 9'-----01-0 ->      3'100 8'00001100
  transition:      3'010 9'-----1--0 ->      3'001 8'00001001
  transition:      3'010 9'--------1 ->      3'000 8'00001000
  transition:      3'001 9'---00--00 ->      3'001 8'00010001
  transition:      3'001 9'---10--00 ->      3'000 8'00010000
  transition:      3'001 9'----0--10 ->      3'011 8'00010011
  transition:      3'001 9'----1---0 ->      3'010 8'00010010
  transition:      3'001 9'--------1 ->      3'000 8'00010000
  transition:      3'011 9'--0-0---0 ->      3'000 8'01000000
  transition:      3'011 9'--1-0---0 ->      3'011 8'01000011
  transition:      3'011 9'----1---0 ->      3'010 8'01000010
  transition:      3'011 9'--------1 ->      3'000 8'01000000
```

Die neue Netzliste befindet sich in Abbildung 5.5. Der Zustandsautomat wurde erkannt und kann jetzt optimiert werden. Führen Sie dafür die folgenden Befehle aus:

```
yosys> fsm_opt

18. Executing FSM_OPT pass (simple optimizations of FSMs).
Optimizing FSM `$fsm$\state$65' from module `\rollosteuerung'.

yosys> opt_clean

19. Executing OPT_CLEAN pass (remove unused cells and wires).
Finding unused cells or wires in module \rollosteuerung..
Removed 20 unused cells and 20 unused wires.

yosys> fsm_opt

20. Executing FSM_OPT pass (simple optimizations of FSMs).
Optimizing FSM `$fsm$\state$65' from module `\rollosteuerung'.
  Removing unused output signal $0\state[2:0] [0].
  Removing unused output signal $0\state[2:0] [1].
  Removing unused output signal $0\state[2:0] [2].
  Removing unused output signal $procmux$24_CMP.
  Removing unused output signal $procmux$31_CMP.
  Removing unused output signal $procmux$58_CMP.
```

Die Ergebnisse der obigen Pässe sind in der Abbildung 5.6 dargestellt.

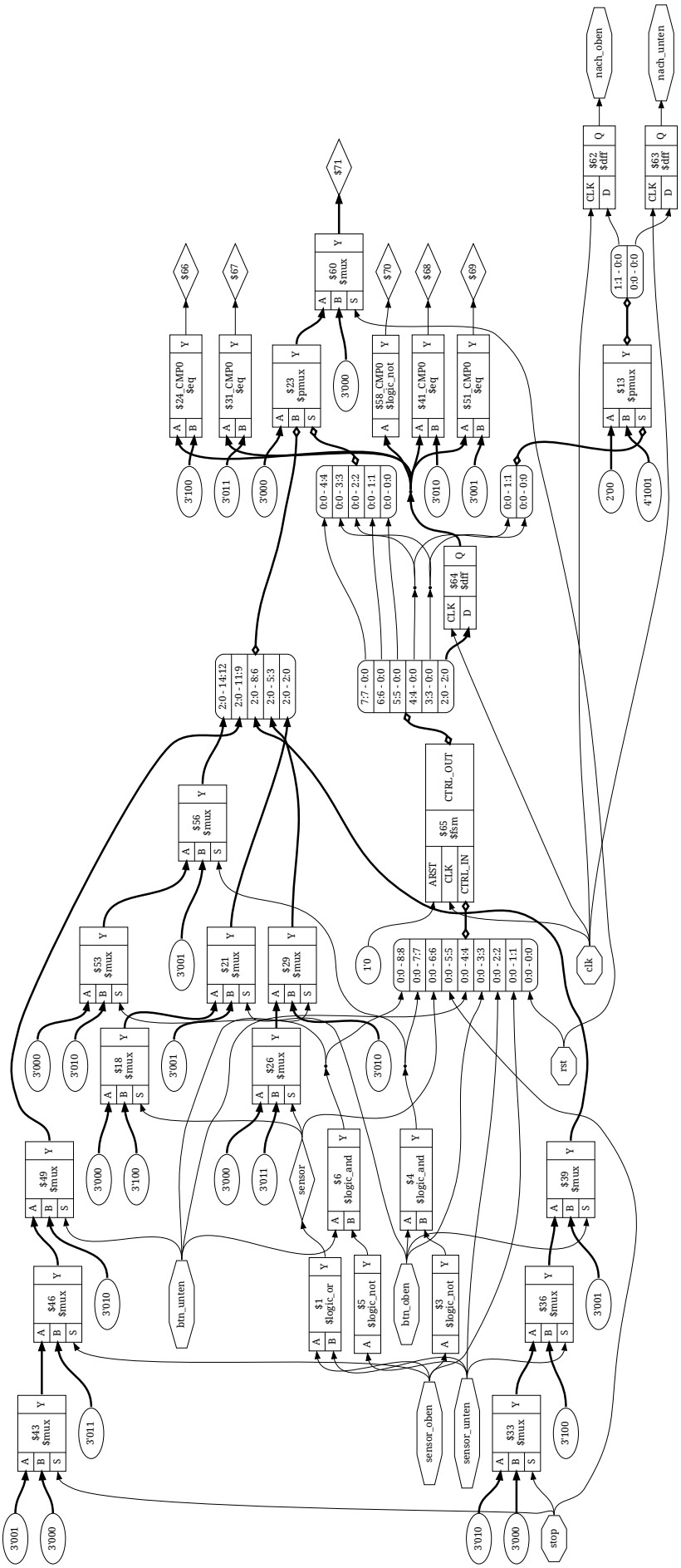


Abbildung 5.5: Die Netzliste nach dem `fsm_extract`-Pass

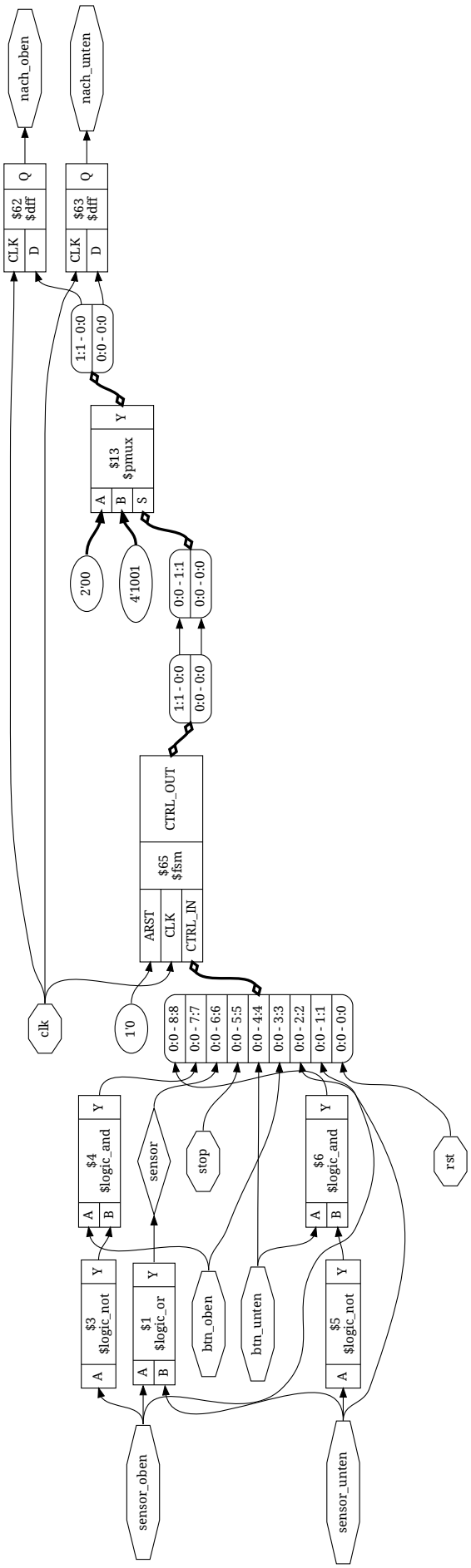


Abbildung 5.6: Die Netzliste nach dem `fsm_opt`-Pass

Die Zustandsautomatenoptimierung hat das Design signifikant verkleinert. Mit den restlichen `fsm`-Pässen wird das Design wieder als RTL dargestellt. Führen Sie die folgenden Befehle aus:

```
yosys> fsm_recode

21. Executing FSM_RECODE pass (re-assigning FSM state encoding).
Recoding FSM `$fsm$\state$65' from module `\rollosteuerung' using
    `auto' encoding:
        mapping auto encoding to `one-hot` for this FSM.
        000 -> ----1
        100 -> ---1-
        010 -> --1--
        001 -> -1---
        011 -> 1----

yosys> fsm_map

22. Executing FSM_MAP pass (mapping FSMs to basic logic).
Mapping FSM `$fsm$\state$65' from module `\rollosteuerung'.
```

Führen Sie zuletzt den `opt`-Pass, um die Erzeugnisse des `fsm`-Passes zu optimieren, mit dem folgendem Befehl aus:

```
yosys> opt

23. Executing OPT pass (performing simple optimizations).
```

Dei Abbildung 5.7 zeigt das Endergebnis.

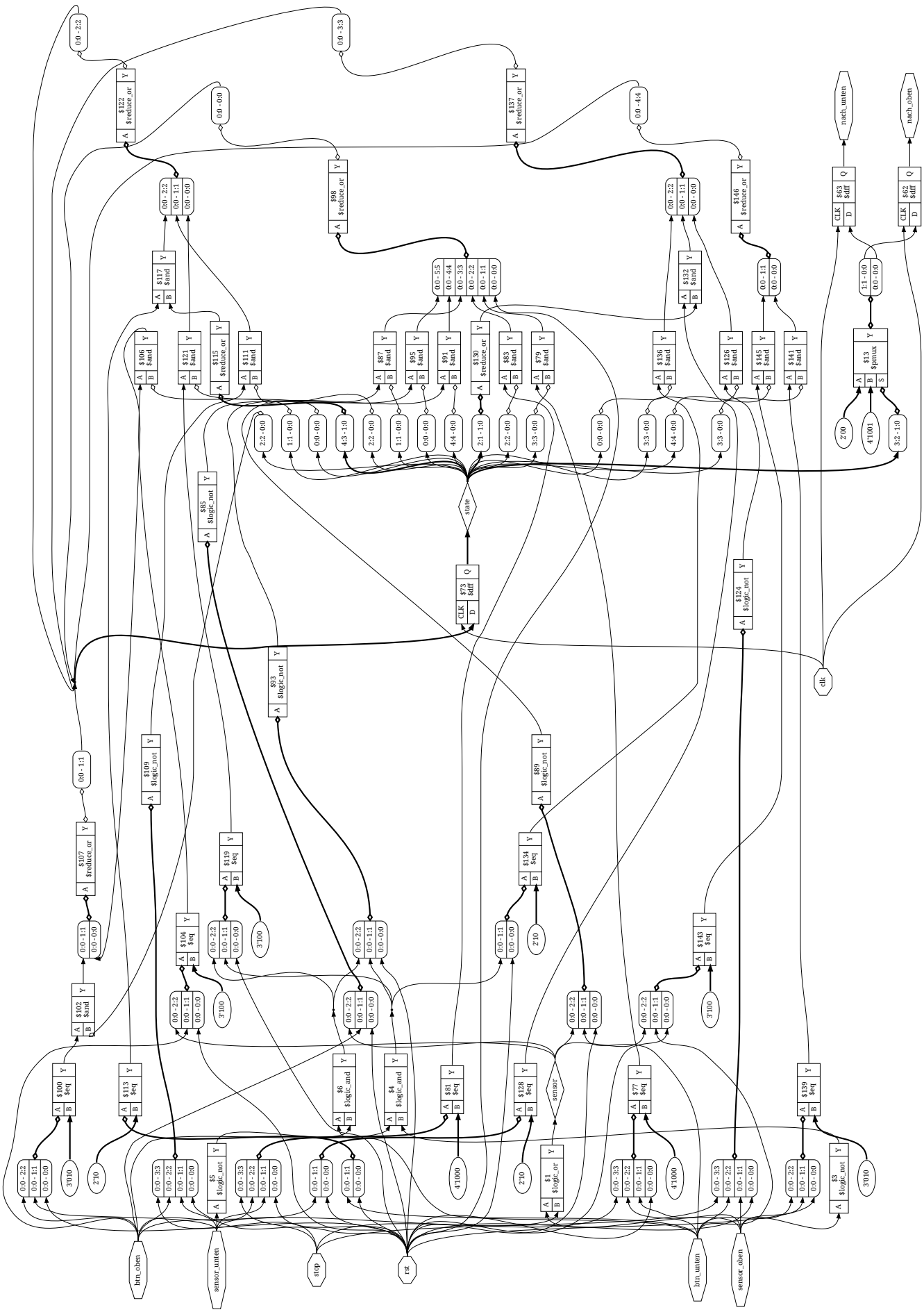


Abbildung 5.7: Die Netzliste nach dem `fsm`-Pass mit ausgeführtem `opt`-Pass