Yosys Manual

Clifford Wolf

Abstract

Most of todays digital design is done in HDL code (mostly Verilog or VHDL) and with the help of HDL synthesis tools.

In special cases such as synthesis for coarse-grain cell libraries or when testing new synthesis algorithms it might be neccessary to write a custom HDL synthesis tool or add new features to an existing one. It this cases the availability of a Free and Open Source (FOSS) synthesis tool that can be used as basis for custom tools would be helpful.

In the absence of such a tool, the Yosys Open SYnthesis Suite (Yosys) was developped. This document covers the design and implementation of this tool. At the moment the main focus of Yosys lies on the high-level aspects of digital synthesis. The pre-existing FOSS logic-synthesis tool ABC is used by Yosys to perform advanced gate-level optimizations.

An evaluation of Yosys based on real-world designs is included. It is shown that Yosys can be used as-is to synthesize such designs. The results produced by Yosys in this tests where successflly verified using formal verification and are compareable in quality to the results produced by a commercial synthesis tool.

This document was originally published as bachelor thesis at the Vienna University of Technology [Wol13].

Abbreviations

AIG	And-Inverter-Graph
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
BDD	Binary Decicion Diagram
BLIF	Berkeley Logic Interchange Format
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
ER Diagram	Entity-Relationship Diagram
FOSS	Free and Open-Source Software
FPGA	Field-Programmable Gate Array
FSM	Finite-state machine
HDL	Hardware Description Language
LPM	Library of Parameterized Modules
RTLIL	RTL Intermediate Language
RTL	Register Transfer Level
SAT	Satisfiability Problem
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit
YOSYS	Yosys Open SYnthesis Suite

Contents

Chapter 1

Introduction

This document presents the Free and Open Source (FOSS) Verilog HDL synthesis tool "Yosys". Its design and implementation as well as its performance on real-world designs is discussed in this document.

1.1 History of Yosys

A Hardware Description Language (HDL) is a computer language used to describe circuits. A HDL synthesis tool is a computer program that takes a formal description of a circuit written in an HDL as input and generates a netlist that implements the given circuit as output.

Currently the most widely used and supported HDLs for digital circuits are Verilog [Ver06][Ver02] and VHDL[1] [VHD09][VHD04]. Both HDLs are used for test and verification purposes as well as logic synthesis, resulting in a set of synthesizable and a set of non-synthesizable language features. In this document we only look at the synthesizable subset of the language features.

In recent work on heterogeneous coarse-grain reconfigurable logic [WGS+12] the need for a custom application-specific HDL synthesis tool emerged. It was soon realised that a synthesis tool that understood Verilog or VHDL would be preferred over a synthesis tool for a custom HDL. Given an existing Verilog or VHDL front end, the work for writing the necessary additional features and integrating them in an existing tool can be estimated to be about the same as writing a new tool with support for a minimalistic custom HDL.

The proposed custom HDL synthesis tool should be licensed under a Free and Open Source Software (FOSS) licence. So an existing FOSS Verilog or VHDL synthesis tool would have been needed as basis to build upon. The main advantages of choosing Verilog or VHDL is the ability to synthesize existing HDL code and to mitigate the requirement for circuit-designers to learn a new language. In order to take full advantage of any existing FOSS Verilog or VHDL tool, such a tool would have to provide a feature-complete implementation of the synthesizable HDL subset.

Basic RTL synthesis is a well understood field [HS96]. Lexing, parsing and processing of computer languages [ASU86] is a thoroughly researched field. All the information required to write such tools has been openly available for a long time, and it is therefore likely that a FOSS HDL synthesis tool with a feature-complete Verilog or VHDL front end must exist which can be used as a basis for a custom RTL synthesis tool.

Due to the authors preference for Verilog over VHDL it has been decided early on to go for Verilog instead of VHDL[2]. So the existing FOSS Verilog synthesis tools were evaluated (see App. E). The results of this evaluation are utterly devastating. Therefore a completely new Verilog synthesis tool was implemented and is recommended as basis for custom synthesis tools. This is the tool that is discussed in this document.

[1] VHDL is an acronym for "VHSIC hardware description language" and VHSIC is an acronym for "Very-High-Speed Integrated Circuits".

[2] A quick investigation into FOSS VHDL tools yielded similar grim results for FOSS VHDL synthesis tools.

1.2 Structure of this Document

The structure of this document is a follows:

Chapter 1 is this introduction.

Chapter 2 covers a short introduction to the world of HDL synthesis. Basic principles and the terminology is outlined in this chapter.

Chapter 3 gives the quickest possible outline to how the problem of implementing a HDL synthesis tool is approached in the case of Yosys.

Chapter 4 contains a more detailed overview of the implementation of Yosys. This chapter covers the data structures used in Yosys to represent a design in detail and is therefore recommended reading for everyone who is interested in understanding the Yosys internals.

Chapter 5 covers the internal cell library used by Yosys. This is especially important knowledge for anyone who wants to understand the intermediate netlists used internally by Yosys.

Chapter 6 gives a tour to the internal APIs of Yosys. This is recommended reading for everyone who actually wants to read or write Yosys source code. The chapter concludes with an example loadable module for Yosys.

Chapters 7, 8, and 9 cover three improtant pieces of the synthesis pileline: The Verilog frontend, the optimization passes and the technology mapping to the target architecture, respectively.

Chapter 10 covers the evaluation of the performance (correctness and quality) of Yosys on real-world input data. The chapter concludes the main part of this document with conclusions and outlook to future work.

Various appendices, including a command reference manual (App. C) and an evaluation of pre-existing FOSS Verilog synthesis tools (App. E) complete this document.

Chapter 2

Basic Principles

This chapter contains a short introduction to the basic principles of digital circuit synthesis.

2.1 Levels of Abstraction

Digital circuits can be represented at different levels of abstraction. During the design process a circuit is usually first specified using a higher level abstraction. Implementation can then be understood as finding a functionally equivalent representation at a lower abstraction level. When this is done automatically using software, the term *synthesis* is used.

So synthesis is the automatic conversion of a high-level representation of a circuit to a functionally equivalent low-level representation of a circuit. Figure 2.1 lists the different levels of abstraction and how they relate to different kinds of synthesis.

Regardless of the way a lower level representation of a circuit is obtained (synthesis or manual design), the lower level representation is usually verified by comparing simulation results of the lower level and the higher level representation [1]. Therefore even if no synthesis is used, there must still be a simulatable representation of the circuit in all levels to allow for verification of the design.

[1] In the last years formal equivalence checking also became an important verification method for validating RTL and lower abstraction representation of the design.

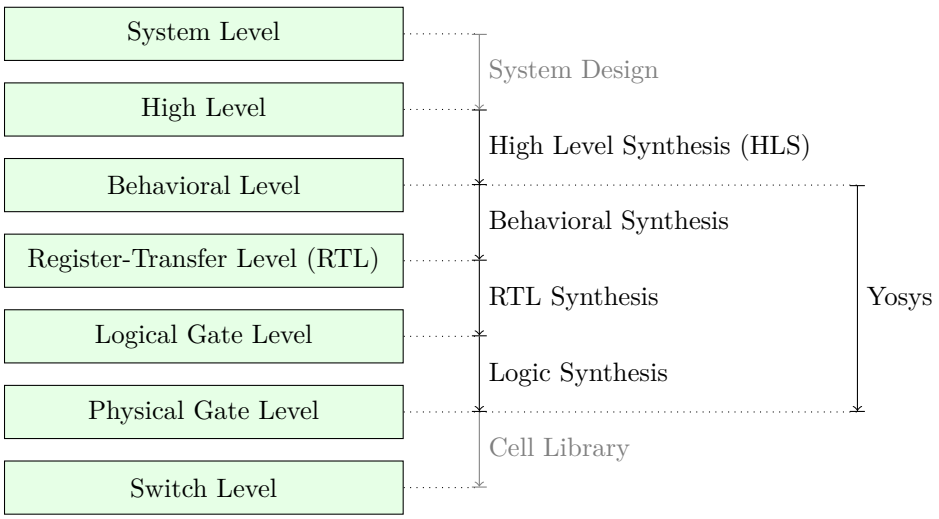

Figure 2.1: Different levels of abstraction and synthesis.

Note: The exact meaning of terminology such as "High-Level" is of course not fixed over time. For example the HDL "ABEL" was first introduced in 1985 as "A High-Level Design Language for Programmable Logic Devices" [LHBB85], but would not be considered a "High-Level Language" today.

2.1.1 System Level

The System Level abstraction of a system only looks at its biggest building blocks like CPUs and computing cores. On this level the circuit is usually described using traditional programming languages like C/C++ or Matlab. Sometimes special software libraries are used that are aimed at simulation circuits on the system level, such as SystemC.

Usually no synthesis tools are used to automatically transform a system level representation of a circuit to a lower-level representation. But system level design tools exist that can be used to connect system level building blocks.

The IEEE 1685-2009 standard defines the IP-XACT file format that can be used to represent designs on the system level and building blocks that can be used in such system level designs. [IP-10]

2.1.2 High Level

The high-level abstraction of a system (sometimes referred to as *algorithmic* level) is also often represented using traditional programming languages, but with a reduced feature set. For example when representing a design at the high level abstraction in C, pointers can only be used to mimic concepts that can be found in hardware, such as memory interfaces. Full featured dynamic memory management is not allowed as it has no corresponding concept in digital circuits.

Tools exist to synthesize high level code (usually in the form of C/C++/SystemC code with additional metadata) to behavioural HDL code (usually in the form of Verilog or VHDL code). Aside from the many commercial tools for high level synthesis there are also a number of FOSS tools for high level synthesis [21] [24].

2.1.3 Behavioural Level

At the behavioural abstraction level a language aimed at hardware description such as Verilog or VHDL is used to describe the circuit, but so-called *behavioural modelling* is used in at least part of the circuit description. In behavioural modelling there must be a language feature that allows for imperative programming to be used to describe data paths and registers. This is the `always`-block in Verilog and the `process`-block in VHDL.

In behavioural modelling, code fragments are provided together with a *sensitivity list*; a list of signals and conditions. In simulation, the code fragment is executed whenever a signal in the sensitivity list changes its value or a condition in the sensitivity list is triggered. A synthesis tool must be able to transfer this representation into an appropriate datapath followed by the appropriate types of register.

For example consider the following verilog code fragment:

```
1 always @(posedge clk)
2         y <= a + b;
```

In simulation the statement `y <= a + b` is executed whenever a positive edge on the signal `clk` is detected. The synthesis result however will contain an adder that calculates the sum `a + b` all the time, followed by a d-type flip-flop with the adder output on its D-input and the signal `y` on its Q-output.

Usually the imperative code fragments used in behavioural modelling can contain statements for conditional execution (**if**- and **case**-statements in Verilog) as well as loops, as long as those loops can be completely unrolled.

Interestingly there seems to be no other FOSS Tool that is capable of performing Verilog or VHDL behavioural syntheses besides Yosys (see App. E).

2.1.4 Register-Transfer Level (RTL)

On the Register-Transfer Level the design is represented by combinatorial data paths and registers (usually d-type flip flops). The following verilog code fragment is equivalent to the previous verilog example, but is in RTL representation:

```
1 assign tmp = a + b;       // combinatorial data path
2
3 always @(posedge clk)     // register
4         y <= tmp;
```

A design in RTL representation is usually stored using HDLs like Verilog and VHDL. But only a very limited subset of features is used, namely minimalistic `always`-blocks (Verilog) or `process`-blocks (VHDL) that model the register type used and unconditional assignments for the datapath logic. The use of HDLs on this level simplifies simulation as no additional tools are required to simulate a design in RTL representation.

Many optimizations and analyses can be performed best at the RTL level. Examples include FSM detection and optimization, identification of memories or other larger building blocks and identification of shareable resources.

Note that RTL is the first abstraction level in which the circuit is represented as a graph of circuit elements (registers and combinatorical cells) and signals. Such a graph, when encoded as list of cells and connections, is called a netlist.

RTL synthesis is easy as each circuit node element in the netlist can simply be replaced with an equivalent gate-level circuit. However, usually the term *RTL synthesis* does not only refer to synthesizing an RTL netlist to a gate level netlist but also to performing a number of highly sophisticated optimizations within the RTL representation, such as the examples listed above.

A number of FOSS tools exist that can perform isolated tasks within the domain of RTL synthesis steps. But there seems to be no FOSS tool that covers a wide range of RTL synthesis operations.

2.1.5 Logical Gate Level

On the logical gate level the design is represented by a netlist that uses only cells from a small number of single-bit cells, such as basic logic gates (AND, OR, NOT, XOR, etc.) and Registers (usually D-Type Flip-flops).

A number of netlist formats exists that can be used on this level, e.g. the Electronic Design Interchange Format (EDIF), but for ease of simulation often a HDL netlist is used. The latter is a HDL file (Verilog or VHDL) that only uses the most basic language constructs for instantiation and connecting of cells.

There are two challenges in logic synthesis: First finding opportunities for optimizations within the gate level netlist and second the optimal (or at least good) mapping of the logic gate netlist to an equivalent netlist of physically available gate types.

The simplest approach to logic synthesis is *two-level logic synthesis*, where a logic function is converted into a sum-of-products representation, e.g. using a karnaugh map. This is a simple approach, but has

exponential worst-case effort and can not make efficient use of physical gates other than AND/NAND-, OR/NOR- and NOT-Gates.

Therefore modern logic synthesis tools utilize much more complicated *multi-level logic synthesis* algorithms [BHSV90]. Most of these algorithms convert the logic function to a Binary-Decision-Diagram (BDD) or And-Inverter-Graph (AIG) and work from that representation. The former has the advantage that it has a unique normalized form. The latter has much better worst case performance and is therefore better suited for the synthesis of large logic functions.

Good FOSS tools exists for multi-level logic synthesis [35] [34] [36].

Yosys contains basic logic synthesis functionality but can also use ABC [35] for the logic synthesis step. Using ABC is recommended.

2.1.6 Physical Gate Level

On the physical gate level only gates are used that are physically available on the target architecture. In some cases this may only be NAND, NOR and NOT gates as well as D-Type registers. In other cases this might include cells that are more complex than the cells used at the logical gate level (e.g. complete half-adders). In the case of an FPGA-based design the physical gate level representation is a netlist of LUTs with optional output registers, as these are the basic building blocks of FPGA logic cells.

For the synthesis tool chain this abstraction is usually the lowest level. In case of an ASIC-based design the cell library might contain further information on how the physical cells map to individual switches (transistors).

2.1.7 Switch Level

A switch level representation of a circuit is a netlist utilizing single transistors as cells. Switch level modelling is possible in Verilog and VHDL, but is seldom used in modern designs, as in modern digital ASIC or FPGA flows the physical gates are considered the atomic build blocks of the logic circuit.

2.1.8 Yosys

Yosys is a Verilog HDL synthesis tool. This means that it takes a behavioural design description as input and generates an RTL, logical gate or physical gate level description of the design as output. Yosys' main strengths are behavioural and RTL synthesis. A wide range of commands (synthesis passes) exist within Yosys that can be used to perform a wide range of synthesis tasks within the domain of behavioural, rtl and logic synthesis. Yosys is designed to be extensible and therefore is a good basis for implementing custom synthesis tools for specialised tasks.

2.2 Features of Synthesizable Verilog

The subset of Verilog [Ver06] that is synthesizable is specified in a separate IEEE standards document, the IEEE standard 1364.1-2002 [Ver02]. This standard also describes how certain language constructs are to be interpreted in the scope of synthesis.

This section provides a quick overview of the most important features of synthesizable Verilog, structured in order of increasing complexity.

2.2.1 Structural Verilog

Structural Verilog (also known as *Verilog Netlists*) is a Netlist in Verilog syntax. Only the following language constructs are used in this case:

- Constant values
- Wire and port declarations
- Static assignments of signals to other signals
- Cell instantiations

Many tools (especially at the back end of the synthesis chain) only support structural verilog as input. ABC is an example of such a tool. Unfortunately there is no standard specifying what *Structural Verilog* actually is, leading to some confusion about what syntax constructs are supported in structural verilog when it comes to features such as attributes or multi-bit signals.

2.2.2 Expressions in Verilog

In all situations where Verilog accepts a constant value or signal name, expressions using arithmetic operations such as `+`, `-` and `*`, boolean operations such as `&` (AND), `|` (OR) and `^` (XOR) and many others (comparison operations, unary operator, etc.) can also be used.

During synthesis these operators are replaced by cells that implement the respective function.

Many FOSS tools that claim to be able to process Verilog in fact only support basic structural verilog and simple expressions. Yosys can be used to convert full featured synthesizable verilog to this simpler subset, thus enabling such applications to be used with a richer set of Verilog features.

2.2.3 Behavioural Modelling

Code that utilizes the Verilog `always` statement is using *Behavioural Modelling*. In behavioural, modelling a circuit is described by means of imperative program code that is executed on certain events, namely any change, a rising edge, or a falling edge of a signal. This is a very flexible construct during simulation but is only synthesizable when one of the following is modelled:

- **Asynchronous or latched logic**
  In this case the sensitivity list must contain all expressions that are used within the `always` block. The syntax `@*` can be used for these cases. Examples of this kind include:

```
1  // asynchronous
2  always @* begin
3          if (add_mode)
4                  y <= a + b;
5          else
6                  y <= a - b;
7  end
8
9  // latched
10 always @* begin
11         if (!hold)
12                 y <= a + b;
13 end
```

Note that latched logic is often considered bad style and in many cases just the result of sloppy HDL design. Therefore many synthesis tools generate warnings whenever latched logic is generated.

- **Synchronous logic (with optional synchronous reset)**
  This is logic with d-type flip-flops on the output. In this case the sensitivity list must only contain the respective clock edge. Example:

```
1 // counter with synchronous reset
2 always @(posedge clk) begin
3         if (reset)
4                 y <= 0;
5         else
6                 y <= y + 1;
7 end
```

- **Synchronous logic with asynchronous reset**
  This is logic with d-type flip-flops with asynchronous resets on the output. In this case the sensitivity list must only contain the respective clock and reset edges. The values assigned in the reset branch must be constant. Example:

```
1 // counter with asynchronous reset
2 always @(posedge clk, posedge reset) begin
3         if (reset)
4                 y <= 0;
5         else
6                 y <= y + 1;
7 end
```

Many synthesis tools support a wider subset of flip-flops that can be modelled using `always`-statements (including Yosys). But only the ones listed above are covered by the Verilog synthesis standard and when writing new designs one should limit herself or himself to these cases.

In behavioural modelling, blocking assignments (=) and non-blocking assignments (<=) can be used. The concept of blocking vs. non-blocking assignment is one of the most misunderstood constructs in Verilog [CI00].

The blocking assignment behaves exactly like an assignment in any imperative programming language, while with the non-blocking assignment the right hand side of the assignment is evaluated immediately but the actual update of the left hand side register is delayed until the end of the time-step. For example the Verilog code `a <= b; b <= a;` exchanges the values of the two registers. See Sec. E.1 for a more detailed description of this behaviour.

2.2.4 Functions and Tasks

Verilog supports *Functions* and *Tasks* to bundle statements that are used in multiple places (similar to *Procedures* in imperative programming). Both constructs can be implemented easily by substituting the function/task-call with the body of the function or task.

2.2.5 Conditionals, Loops and Generate-Statements

Verilog supports **`if-else`**-statements and **`for`**-loops inside **`always`**-statements.

It also supports both features in **`generate`**-statements on the module level. This can be used to selectively enable or disable parts of the module based on the module parameters (**`if-else`**) or to generate a set of similar subcircuits (**`for`**).

While the **if-else**-statement inside an always-block is part of behavioural modelling, the three other cases are (at least for a synthesis tool) part of a built-in macro processor. Therefore it must be possible for the synthesis tool to completely unroll all loops and evaluate the condition in all **if-else**-statement in **generate**-statements using const-folding.

Examples for this can be found in Fig. E.4 and Fig. E.5 in App. E.

2.2.6 Arrays and Memories

Verilog supports arrays. This is in general a synthesizable language feature. In most cases arrays can be synthesized by generating addressable memories. However, when complex or asynchronous access patterns are used, it is not possible to model an array as memory. In these cases the array must be modelled using individual signals for each word and all accesses to the array must be implemented using large multiplexers.

In some cases it would be possible to model an array using memories, but it is not desired. Consider the following delay circuit:

```
1  module (clk, in_data, out_data);
2
3  parameter BITS = 8;
4  parameter STAGES = 4;
5
6  input clk;
7  input [BITS-1:0] in_data;
8  output [BITS-1:0] out_data;
9  reg [BITS-1:0] ffs [STAGES-1:0];
10
11 integer i;
12 always @(posedge clk) begin
13         ffs[0] <= in_data;
14         for (i = 1; i < STAGES; i = i+1)
15                 ffs[i] <= ffs[i-1];
16 end
17
18 assign out_data = ffs[STAGES-1];
19
20 endmodule
```

This could be implemented using an addressable memory with `STAGES` input and output ports. A better implementation would be to use a simple chain of flip-flops (a so-called shift register). This better implementation can either be obtained by first creating a memory-based implementation and then optimizing it based on the static address signals for all ports or directly identifying such situations in the language front end and converting all memory accesses to direct accesses to the correct signals.

2.3 Challenges in Digital Circuit Synthesis

This section summarizes the most important challenges in digital circuit synthesis. Tools can be characterized by how well they address these topics.

2.3.1 Standards Compliance

The most important challenge is compliance with the HDL standards in question (in case of Verilog the IEEE Standards 1364.1-2002 and 1364-2005). This can be broken down in two items:

- Completeness of implementation of the standard
- Correctness of implementation of the standard

Completeness is mostly important to guarantee compatibility with existing HDL code. Once a design has been verified and tested, HDL designers are very reluctant regarding changes to the design, even if it is only about a few minor changes to work around a missing feature in a new synthesis tool.

Correctness is crucial. In some areas this is obvious (such as correct synthesis of basic behavioural models). But it is also crucial for the areas that concern minor details of the standard, such as the exact rules for handling signed expressions, even when the HDL code does not target different synthesis tools. This is because (different to software source code that is only processed by compilers), in most design flows HDL code is not only processed by the synthesis tool but also by one or more simulators and sometimes even a formal verification tool. It is key for this verification process that all these tools use the same interpretation for the HDL code.

2.3.2 Optimizations

Generally it is hard to give a one-dimensional description of how well a synthesis tool optimizes the design. First of all because not all optimizations are applicable to all designs and all synthesis tasks. Some optimizations work (best) on a coarse grain level (with complex cells such as adders or multipliers) and others work (best) on a fine grain level (single bit gates). Some optimizations target area and others target speed. Some work well on large designs while others don't scale well and can only be applied to small designs.

A good tool is capable of applying a wide range of optimizations at different levels of abstraction and gives the designer control over which optimizations are performed (or skipped) and what the optimization goals are.

2.3.3 Technology Mapping

Technology mapping is the process of converting the design into a netlist of cells that are available in the target architecture. In an ASIC flow this might be the process-specific cell library provided by the fab. In an FPGA flow this might be LUT cells as well as special function units such as dedicated multipliers. In a coarse-grain flow this might even be more complex special function units.

An open and vendor independent tool is especially of interest if it supports a wide range of different types of target architectures.

2.4 Script-Based Synthesis Flows

A digital design is usually started by implementing a high-level or system-level simulation of the desired function. This description is then manually transformed (or re-implemented) into a synthesizable lower-level description (usually at the behavioural level) and the equivalence of the two representations is verified by simulating both and comparing the simulation results.

Then the synthesizable description is transformed to lower-level representations using a series of tools and the results are again verified using simulation. This process is illustrated in Fig. 2.2.

In this example the System Level Model and the Behavioural Model are both manually written design files. After the equivalence of system level model and behavioural model has been verified, the lower level representations of the design can be generated using synthesis tools. Finally the RTL Model and the Gate-Level Model are verified and the design process is finished.

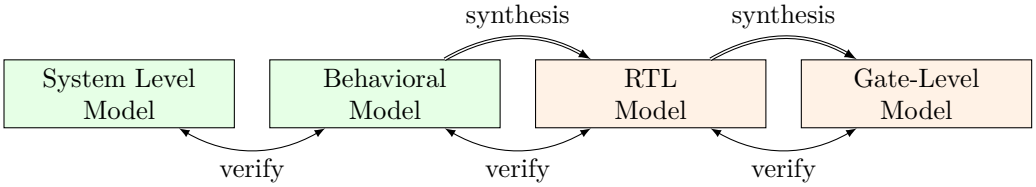

Figure 2.2: Typical design flow. Green boxes represent manually created models. Orange boxes represent models generated by synthesis tools.

However, in any real-world design effort there will be multiple iterations for this design process. The reason for this can be the late change of a design requirement or the fact that the analysis of a low-abstraction model (e.g. gate-level timing analysis) revealed that a design change is required in order to meet the design requirements (e.g. maximum possible clock speed).

Whenever the behavioural model or the system level model is changed their equivalence must be re-verified by re-running the simulations and comparing the results. Whenever the behavioural model is changed the synthesis must be re-run and the synthesis results must be re-verified.

In order to guarantee reproducibility it is important to be able to re-run all automatic steps in a design project with a fixed set of settings easily. Because of this, usually all programs used in a synthesis flow can be controlled using scripts. This means that all functions are available via text commands. When such a tool provides a gui, this is complementary to, and not instead of, a command line interface.

Usually a synthesis flow in an UNIX/Linux environment would be controlled by a shell script that calls all required tools (synthesis and simulation/verification in this example) in the correct order. Each of these tools would be called with a script file containing commands for the respective tool. All settings required for the tool would be provided by these script files so that no manual interaction would be necessary. These script files are considered design sources and should be kept under version control just like the source code of the system level and the behavioural model.

2.5 Methods from Compiler Design

Some parts of synthesis tools involve problem domains that are traditionally known from compiler design. This section addresses some of these domains.

2.5.1 Lexing and Parsing

The best known concepts from compiler design are probably *lexing* and *parsing*. These are two methods that together can be used to process complex computer languages easily. [ASU86]

A *lexer* consumes single characters from the input and generates a stream of *lexical tokens* that consist of a *type* and a *value*. For example the Verilog input "`assign foo = bar + 42;`" might be translated by the lexer to the list of lexical tokens given in Tab. 2.1.

The lexer is usually generated by a lexer generator (e.g. `flex` [22]) from a description file that is using regular expressions to specify the text pattern that should match the individual tokens.

The lexer is also responsible for skipping ignored characters (such as white spaces outside string constants and comments in the case of Verilog) and converting the original text snippet to a token value.

Note that individual keywords use different token types (instead of a keyword type with different token values). This is because the parser usually can only use the Token-Type to make a decision on the grammatical role of a token.

Token-Type	Token-Value
TOK_ASSIGN	-
TOK_IDENTIFIER	"foo"
TOK_EQ	-
TOK_IDENTIFIER	"bar"
TOK_PLUS	-
TOK_NUMBER	42
TOK_SEMICOLON	-

Table 2.1: Exemplary token list for the statement "**assign** foo = bar + 42;".

The parser then transforms the list of tokens into a parse tree that closely resembles the productions from the computer languages grammar. As the lexer, the parser is also typically generated by a code generator (e.g. `bison` [23]) from a grammar description in Backus-Naur Form (BNF).

Let's consider the following BNF (in Bison syntax):

```
1 assign_stmt: TOK_ASSIGN TOK_IDENTIFIER TOK_EQ expr TOK_SEMICOLON;
2 expr: TOK_IDENTIFIER | TOK_NUMBER | expr TOK_PLUS expr;
```

The parser converts the token list to the parse tree in Fig. 2.3. Note that the parse tree never actually exists as a whole as data structure in memory. Instead the parser calls user-specified code snippets (so-called *reduce-functions*) for all inner nodes of the parse tree in depth-first order.

In some very simple applications (e.g. code generation for stack machines) it is possible to perform the task at hand directly in the reduce functions. But usually the reduce functions are only used to build an in-memory data structure with the relevant information from the parse tree. This data structure is called an *abstract syntax tree* (AST).

The exact format for the abstract syntax tree is application specific (while the format of the parse tree and token list are mostly dictated by the grammar of the language at hand). Figure 2.4 illustrates what an AST for the parse tree in Fig. 2.3 could look like.

Usually the AST is then converted into yet another representation that is more suitable for further processing. In compilers this is often an assembler-like three-address-code intermediate representation. [ASU86]

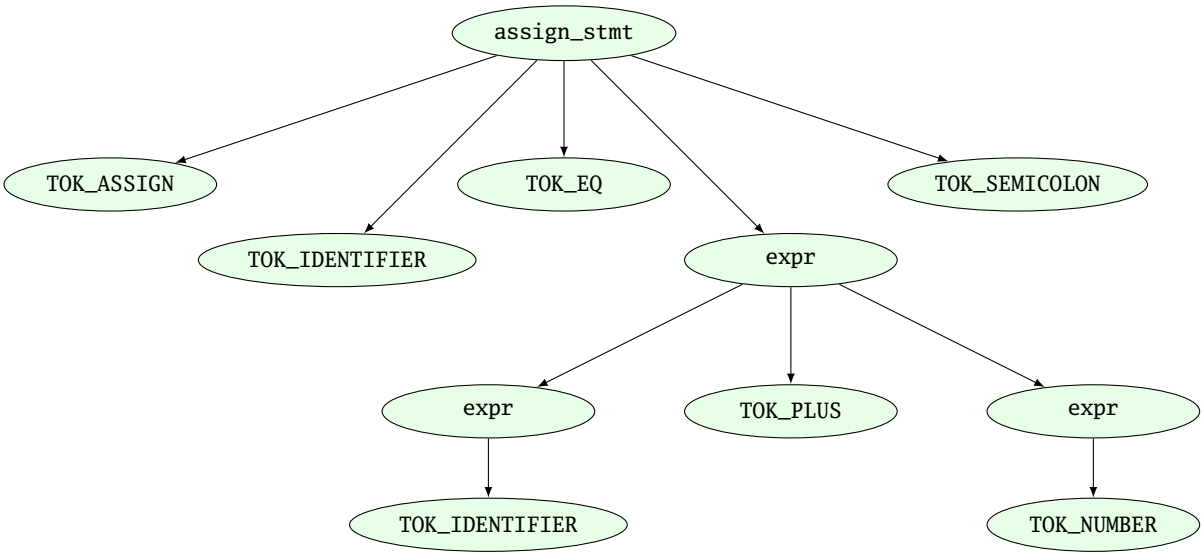

Figure 2.3: Example parse tree for the Verilog expression "**assign** foo = bar + 42;".

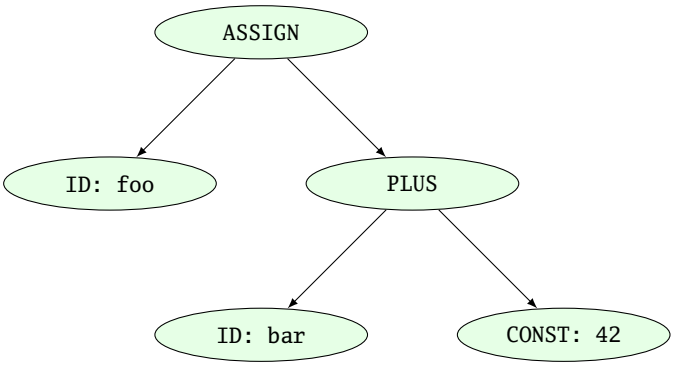

Figure 2.4: Example abstract syntax tree for the Verilog expression "`assign foo = bar + 42;`".

2.5.2 Multi-Pass Compilation

Complex problems are often best solved when split up into smaller problems. This is certainly true for compilers as well as for synthesis tools. The components responsible for solving the smaller problems can be connected in two different ways: through *Single-Pass Pipelining* and by using *Multiple Passes*.

Traditionally a parser and lexer are connected using the pipelined approach: The lexer provides a function that is called by the parser. This function reads data from the input until a complete lexical token has been read. Then this token is returned to the parser. So the lexer does not first generate a complete list of lexical tokens and then passes it to the parser. Instead they are running concurrently and the parser can consume tokens as the lexer produces them.

The single-pass pipelining approach has the advantage of lower memory footprint (at no time the complete design must be kept in memory) but has the disadvantage of tighter coupling between the interacting components.

Therefore single-pass pipelining should only be used when the lower memory footprint is required or the components are also conceptually tightly coupled. The latter certainly is the case for a parser and its lexer. But when data is passed between two conceptually loosely coupled components it is often beneficial to use a multi-pass approach.

In the multi-pass approach the first component processes all the data and the result is stored in a in-memory data structure. Then the second component is called with this data. This reduces complexity, as only one component is running at a time. It also improves flexibility as components can be exchanged easier.

Most modern compilers are multi-pass compilers.

Chapter 3

Approach

Yosys is a tool for synthesising (behavioural) Verilog HDL code to target architecture netlists. Yosys aims at a wide range of application domains and thus must be flexible and easy to adapt to new tasks. This chapter covers the general approach followed in the effort to implement this tool.

3.1 Data- and Control-Flow

The data- and control-flow of a typical synthesis-tool is very similar to the data- and control-flow of a typical compiler: different subsystems are called in a predetermined order, each consuming the data generated by the last subsystem and generating the data for the next subsystem (see Fig. 3.1).

The first subsystem to be called is usually called a *frontend*. It does not process the data generated by another subsystem but instead reads the user input; in the case of a HDL synthesis tool the behavioural HDL code.

The subsystems that consume data from previous subsystems and produces data for the next subsystems (usually in the same or a similar format) are called *passes*.

The last subsystem that is executed transforms the data generated by the last pass into a suitable output format and writes it to a disk file. This subsystem is usually called the *backend*.

In Yosys all frontends, passes and backends are directly available as commands in the synthesis script. Thus the user can easily create a custom synthesis flow just by calling passes in the right order in a synthesis script.

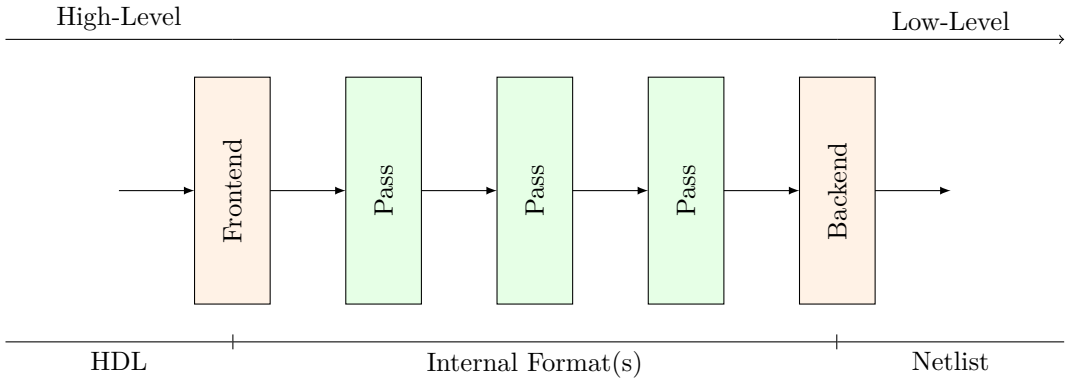

Figure 3.1: General data- and control-flow of a synthesis tool

3.2 Internal Formats in Yosys

Yosys uses two different internal formats. The first is used to store an abstract syntax tree (AST) of a verilog input file. This format is simply called *AST* and is generated by the Verilog Frontend. This data structure is then consumed by a subsystem called *AST Frontend*[1]. This AST Frontend then generates a design in Yosys' main internal format, the Register-Transfer-Level-Intermediate-Language (RTLIL) representation. It does that by first performing a number of simplifications within the AST representation and then generating RTLIL from the simplified AST data structure.

The RTLIL representation is used by all passes as input and outputs. This has the following advantages over using different representational formats between different passes:

- The passes can be re-arranged in a different order and passes can be removed or inserted.
- Passes can simply pass-thru the parts of the design they don't change without the need to convert between formats. In fact Yosys passes output the same data structure they received as input and perform all changes in place.
- All passes use the same interface, thus reducing the effort required to understand a pass when reading the Yosys source code, e.g. when adding additional features.

The RTLIL representation is basically a netlist representation with the following additional features:

- An internal cell library with fixed-function cells to represent RTL datapath and register cells as well as logical gate-level cells (single-bit gates and registers).
- Support for multi-bit values that can use individual bits from wires as well as constant bits to represent coarse-grain netlists.
- Support for basic behavioural constructs (if-then-else structures and multi-case switches with a sensitivity list for updating the outputs).
- Support for multi-port memories.

The use of RTLIL also has the disadvantage of having a very powerful format between all passes, even when doing gate-level synthesis where the more advanced features are not needed. In order to reduce complexity for passes that operate on a low-level representation, these passes check the features used in the input RTLIL and fail to run when non-supported high-level constructs are used. In such cases a pass that transforms the higher-level constructs to lower-level constructs must be called from the synthesis script first.

3.3 Typical Use Case

The following example script may be used in a synthesis flow to convert the behavioural Verilog code from the input file `design.v` to a gate-level netlist `synth.v` using the cell library described by the Liberty file [30] `cells.lib`:

```
1 # read input file tpo internal representation
2 read_verilog design.v
3
4 # convert high-level behavioral parts ("processes") to d-type flip-flops and muxes
5 proc
```

[1] In Yosys the term *pass* is only used to refer to commands that operate on the RTLIL data structure.

```
6
7  # perform some simple optimizations
8  opt
9
10 # convert high-level memory constructs to d-type flip-flops and multiplexers
11 memory
12
13 # perform some simple optimizations
14 opt
15
16 # convert design to (logical) gate-level netlists
17 techmap
18
19 # perform some simple optimizations
20 opt
21
22 # map internal register types to the ones from the cell library
23 dfflibmap -liberty cells.lib
24
25 # use ABC to map remaining logic to cells from the cell library
26 abc -liberty cells.lib
27
28 # cleanup
29 opt
30
31 # write results to output file
32 write_verilog synth.v
```

A detailed description of the commands available in Yosys can be found in App. C.

Chapter 4

Implementation Overview

Yosys is an extensible open source hardware synthesis tool. It is aimed at designers who are looking for an easy accessible, universal, and vendor independent synthesis tool, and scientists who do research in electronic design automation (EDA) and are looking for an open synthesis framework that can be used to test algorithms on complex real-world designs.

Yosys can synthesize a large subset of Verilog 2005 and has been tested with a wide range of real-world designs, including the OpenRISC 1200 CPU [28], the openMSP430 CPU [27], the OpenCores I^2C master [25] and the k68 CPU [26].

As of this writing a Yosys VHDL frontend is in development.

Yosys is written in C++ (using some features from the new C++11 standard). This chapter describes some of the fundamental Yosys data structures. For the sake of simplicity the C++ type names used in the Yosys implementation are used in this chapter, even though the chapter only explains the conceptual idea behind it and can be used as reference to implement a similar system in any language.

4.1 Simplified Data Flow

Figure 4.1 shows the simplified data flow within Yosys. Rectangles in the figure represent program modules and ellipses internal data structures that are used to exchange design data between the program modules.

Design data is read in using one of the frontend modules. The high-level HDL frontends for Verilog and VHDL code generate an abstract syntax tree (AST) that is then passed to the AST frontend. Note that both HDL frontends use the same AST representation that is powerful enough to cover the Verilog HDL and VHDL language.

The AST Frontend then compiles the AST to Yosys's main internal data format, the RTL Intermediate Language (RTLIL). A more detailed description of this format is given in the next section.

There is also a text representation of the RTLIL data structure that can be parsed using the ILANG Frontend.

The design data may then be transformed using a series of passes that all operate on the RTLIL representation of the design.

Finally the design in RTLIL representation is converted back to text by one of the backends, namely the Verilog Backend for generating Verilog netlists and the ILANG Backend for writing the RTLIL data in the same format that is understood by the ILANG Frontend.

With the exception of the AST Frontend, that is called by the high-level HDL frontends and can't be called directly by the user, all program modules are called by the user (usually using a synthesis script that contains text commands for Yosys).

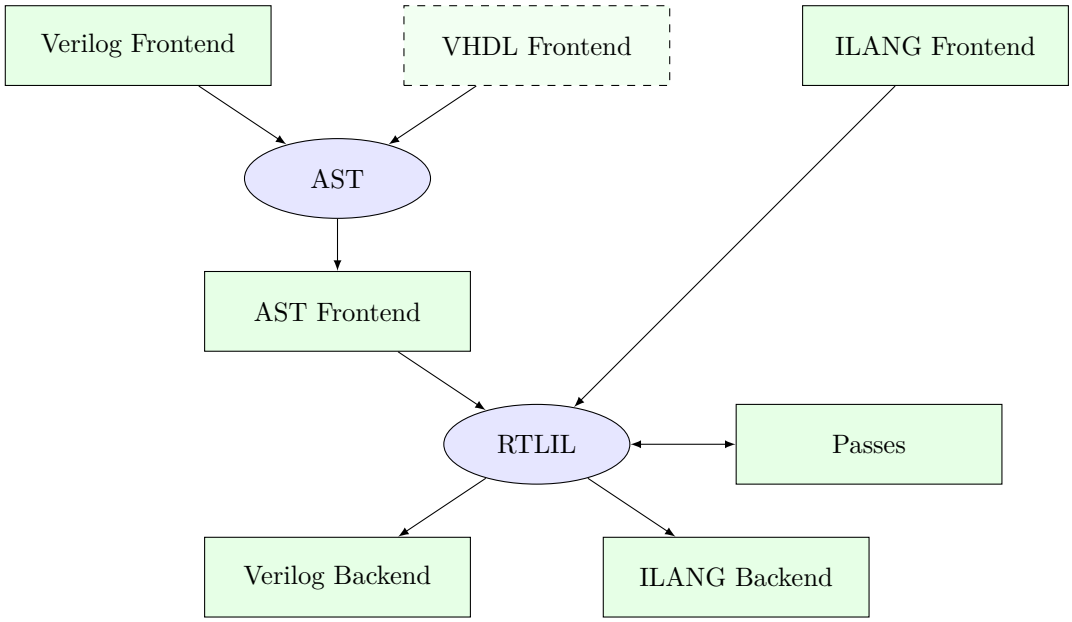

Figure 4.1: Yosys simplified data flow (ellipses: data structures, rectangles: program modules)

By combining passes in different ways and/or adding additional passes to Yosys it is possible to adapt Yosys to a wide range of applications. For this to be possible it is key that (1) all passes operate on the same data structure (RTLIL) and (2) that this data structure is powerful enough represent the design in different stages of the synthesis.

4.2 The RTL Intermediate Language

All frontends, passes and backends in Yosys operate on a design in RTLIL[1] representation. The only exception are the high-level frontends that use the AST representation as an intermediate step before generating RTLIL data.

In order to avoid re-inventing names for the RTLIL classes, they are simply referred to by their full C++ name, i.e. including the `RTLIL::` namespace prefix, in this document.

Figure 4.2 shows a simplified Entity-Relationship Diagram (ER Diagram) of RTLIL. In $1 : N$ relationships the arrow points from the N side to the 1. For example one RTLIL::Design contains N (zero to many) instances of RTLIL::Module. A two-pointed arrow indicates a $1 : 1$ relationship.

The RTLIL::Design is the root object of the RTLIL data structure. There is always one "current design" in memory on which passes operate, frontends add data to it and backends convert to exportable formats. But in some cases passes internally generate additional RTLIL::Design objects. For example when a pass is reading an auxiliary Verilog file such as a cell library, it might create an additional RTLIL::Design object and call the Verilog frontend with this other object to parse the cell library.

There is only one active RTLIL::Design object that is used by all frontends, passes and backends called by the user, e.g. using a synthesis script. The RTLIL::Design then contains zero to many RTLIL::Module objects. This corresponds to modules in Verilog or entities in VHDL. Each module in turn contains objects from three different categories:

- RTLIL::Cell and RTLIL::Wire objects represent classical netlist data.

[1] The *Language* in *RTL Intermediate Language* refers to the fact, that RTLIL also has a text representation, usually referred to as *Intermediate Language* (ILANG).

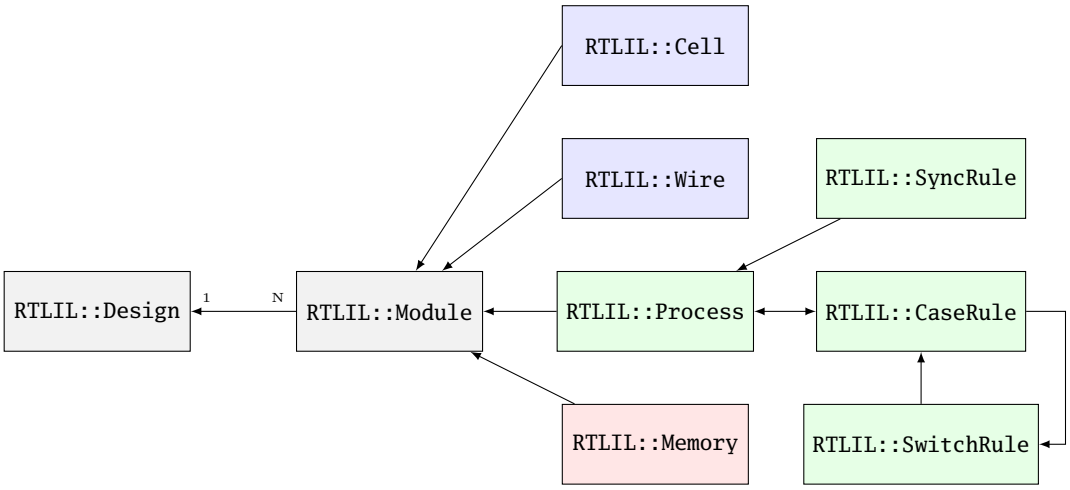

Figure 4.2: Simplified RTLIL Entity-Relationship Diagram

- RTLIL::Process objects represent the decision trees (if-then-else statements, etc.) and synchronization declarations (clock signals and sensitivity) from Verilog `always` and VHDL `process` blocks.
- RTLIL::Memory objects represent addressable memories (arrays).

Usually the output of the synthesis procedure is a netlist, i.e. all RTLIL::Process and RTLIL::Memory objects must be replaced by RTLIL::Cell and RTLIL::Wire objects by synthesis passes.

All features of the HDL that cannot be mapped directly to these RTLIL classes must be transformed to an RTLIL-compatible representation by the HDL frontend. This includes Verilog-features such as generate-blocks, loops and parameters.

The following sections contain a more detailed description of the different parts of RTLIL and rationales behind some of the design decisions.

4.2.1 RTLIL Identifiers

All identifiers in RTLIL (such as module names, port names, signal names, cell types, etc.) follow the following naming convention: They must either start with a backslash (\) or a dollar sign ($).

Identifiers starting with a backslash are public visible identifiers. Usually they originate from one of the HDL input files. For example the signal name "`\sig42`" is most likely a signal that was declared using the name "`sig42`" in an HDL input file. On the other hand the signal name "`$sig42`" is an auto-generated signal name. The backends convert all identifiers that start with a dollar sign to identifiers that do not collide with identifiers that start with a backslash.

This has three advantages:

- Firstly it is impossible that an auto-generated identifier collides with an identifier that was provided by the user.
- Secondly the information about which identifiers were originally provided by the user is always available which can help guide some optimizations. For example the "opt_rmunused" is trying to preserve signals with a user-provided name but doesn't hesitate to delete signals that have auto-generated names when they just duplicate other signals.

- Thirdly the delicate job of finding suitable auto-generated public visible names is deferred to one central location. Internally auto-generated names that may hold important information for Yosys developers can be used without disturbing external tools. For example the Verilog backend assigns names in the form *_integer_*.

In order to avoid programming errors, the RTLIL data structures check if all identifiers start with either a backslash or a dollar sign and generate a runtime error if this rule is violated.

All RTLIL identifiers are case sensitive.

4.2.2 RTLIL::Design and RTLIL::Module

The RTLIL::Design object is basically just a container for RTLIL::Module objects. In addition to a list of RTLIL::Module objects the RTLIL::Design also keeps a list of *selected objects*, i.e. the objects that passes should operate on. In most cases the whole design is selected and therefore passes operate on the whole design. But this mechanism can be useful for more complex synthesis jobs in which only parts of the design should be affected by certain passes.

Besides the objects shown in the ER diagram in Fig. 4.2 an RTLIL::Module object contains the following additional properties:

- The module name
- A list of attributes
- A list of connections between wires
- An optional frontend callback used to derive parametrized variations of the module

The attributes can be Verilog attributes imported by the Verilog frontend or attributes assigned by passes. They can be used to store additional metadata about modules or just mark them to be used by certain part of the synthesis script but not by others.

Verilog and VHDL both support parametric modules (known as "generic entities" in VHDL). The RTLIL format does not support parametric modules itself. Instead each module contains a callback function into the AST frontend to generate a parametrized variation of the RTLIL::Module as needed. This callback then returns the auto-generated name of the parametrized variation of the module. (A hash over the parameters and the module name is used to prohibit the same parametrized variation to be generated twice. For modules with only a few parameters, a name directly containing all parameters is generated instead of a hash string.)

4.2.3 RTLIL::Cell and RTLIL::Wire

A module contains zero to many RTLIL::Cell and RTLIL::Wire objects. Objects of these types are used to model netlists. Usually the goal of all synthesis efforts is to convert all modules to a state where the functionality of the module is implemented only by cells from a given cell library and wires to connect these cells with each other. Note that module ports are just wires with a special property.

An RTLIL::Wire object has the following properties:

- The wire name
- A list of attributes
- A width (busses are just wires with a width > 1)

- If the wire is a port: port number and direction (input/output/inout)

As with modules, the attributes can be Verilog attributes imported by the Verilog frontend or attributes assigned by passees.

In Yosys, busses (signal vectors) are represented using a single wire object with a width > 1. So Yosys does not convert signal vectors to individual signals. This makes some aspects of RTLIL more complex but enables Yosys to be used for coarse grain synthesis where the cells of the target architecture operate on entire signal vectors instead of single bit wires.

An RTLIL::Cell object has the following properties:

- The cell name and type
- A list of attributes
- A list of parameters (for parametric cells)
- Cell ports and the connections of ports to wires and constants

The connections of ports to wires are coded by assigning an RTLIL::SigSpec to each cell ports. The RTLIL::SigSpec data type is described in the next section.

4.2.4 RTLIL::SigSpec

A "signal" is everything that can be applied to a cell port. I.e.

- Any constant value of arbitrary bit-width
  For example: `1337, 16'b0000010100111001, 1'b1, 1'bx`
- All bits of a wire or a selection of bits from a wire
  For example: `mywire, mywire[24], mywire[15:8]`
- Concatenations of the above
  For example: `{16'd1337, mywire[15:8]}`

The RTLIL::SigSpec data type is used to represent signals. The RTLIL::Cell object contains one RTLIL::SigSpec for each cell port.

In addition, connections between wires are represented using a pair of RTLIL::SigSpec objects. Such pairs are needed in different locations. Therefore the type name RTLIL::SigSig was defined for such a pair.

4.2.5 RTLIL::Process

When a high-level HDL frontend processes behavioural code it splits it up into data path logic (e.g. the expression `a + b` is replaced by the output of an adder that takes `a` and `b` as inputs) and an RTLIL::Process that models the control logic of the behavioural code. Let's consider a simple example:

```
1 module ff_with_en_and_async_reset(clock, reset, enable, d, q);
2 input clock, reset, enable, d;
3 output reg q;
4 always @(posedge clock, posedge reset)
5         if (reset)
6                 q <= 0;
7         else if (enable)
8                 q <= d;
9 endmodule
```

In this example there is no data path and therefore the RTLIL::Module generated by the frontend only contains a few RTLIL::Wire objects and an RTLIL::Process. The RTLIL::Process in ILANG syntax:

```
1  process $proc$ff_with_en_and_async_reset.v:4$1
2          assign $0\q[0:0] \q
3          switch \reset
4                  case 1'1
5                          assign $0\q[0:0] 1'0
6                  case
7                          switch \enable
8                                  case 1'1
9                                          assign $0\q[0:0] \d
10                                 case
11                         end
12         end
13         sync posedge \clock
14                 update \q $0\q[0:0]
15         sync posedge \reset
16                 update \q $0\q[0:0]
17 end
```

This RTLIL::Process contains two RTLIL::SyncRule objects, two RTLIL::SwitchRule objects and five RTLIL::CaseRule objects. The wire `$0\q[0:0]` is an automatically created wire that holds the next value of `\q`. The lines 2 . . . 12 describe how `$0\q[0:0]` should be calculated. The lines 13 . . . 16 describe how the value of `$0\q[0:0]` is used to update `\q`.

An RTLIL::Process is a container for zero or more RTLIL::SyncRule objects and exactly one RTLIL::CaseRule object, which is called the *root case*.

An RTLIL::SyncRule object contains an (optional) synchronization condition (signal and edge-type) and zero or more assignments (RTLIL::SigSig).

An RTLIL::CaseRule is a container for zero or more assignments (RTLIL::SigSig) and zero or more RTLIL::SwitchRule objects. An RTLIL::SwitchRule objects is a container for zero or more RTLIL::CaseRule objects.

In the above example the lines 2 . . . 12 are the root case. Here `$0\q[0:0]` is first assigned the old value `\q` as default value (line 2). The root case also contains an RTLIL::SwitchRule object (lines 3 . . . 12). Such an object is very similar to the C `switch` statement as it uses a control signal (`\reset` in this case) to determine which of its cases should be active. The RTLIL::SwitchRule object then contains one RTLIL::CaseRule object per case. In this example there is a case[2] for `\reset == 1` that causes `$0\q[0:0]` to be set (lines 4 and 5) and a default case that in turn contains a switch that sets `$0\q[0:0]` to the value of `\d` if `\enable` is active (lines 6 . . . 11).

The lines 13 . . . 16 then cause `\q` to be updated whenever there is a positive clock edge on `\clock` or `\reset`.

In order to generate such a representation, the language frontend must be able to handle blocking and nonblocking assignments correctly. However, the language frontend does not need to identify the correct type of storage element for the output signal or generate multiplexers for the decision tree. This is done by passes that work on the RTLIL representation. Therefore it is relatively easy to substitute these steps with other algorithms that target different target architectures or perform optimizations or other transformations on the decision trees before further processing them.

One of the first actions performed on a design in RTLIL representation in most synthesis scripts is identifying asynchronous resets. This is usually done using the `proc_arst` pass. This pass transforms the above example to the following RTLIL::Process:

[2] The syntax `1'1` in the ILANG code specifies a constant with a length of one bit (the first "1"), and this bit is a one (the second "1").

```
process $proc$ff_with_en_and_async_reset.v:4$1
        assign $0\q[0:0] \q
        switch \enable
                case 1'1
                        assign $0\q[0:0] \d
                case
        end
        sync posedge \clock
                update \q $0\q[0:0]
        sync high \reset
                update \q 1'0
end
```

This pass has transformed the outer RTLIL::SwitchRule into a modified RTLIL::SyncRule object for the `\reset` signal. Further processing converts the RTLIL::Process e.g. into a d-type flip-flop with asynchronous reset and a multiplexer for the enable signal:

```
cell $adff $procdff$6
        parameter \ARST_POLARITY 1'1
        parameter \ARST_VALUE 1'0
        parameter \CLK_POLARITY 1'1
        parameter \WIDTH 1
        connect \ARST \reset
        connect \CLK \clock
        connect \D $0\q[0:0]
        connect \Q \q
end
cell $mux $procmux$3
        parameter \WIDTH 1
        connect \A \q
        connect \B \d
        connect \S \enable
        connect \Y $0\q[0:0]
end
```

Different combinations of passes may yield different results. Note that `$adff` and `$mux` are internal cell types that still need to be mapped to cell types from the target cell library.

Some passes refuse to operate on modules that still contain RTLIL::Process objects as the presence of these objects in a module increases the complexity. Therefore the passes to translate processes to a netlist of cells are usually called early in a synthesis script. The `proc` pass calls a series of other passes that together perform this conversion in a way that is suitable for most synthesis taks.

4.2.6 RTLIL::Memory

For every array (memory) in the HDL code an RTLIL::Memory object is created. A memory object has the following properties:

- The memory name
- A list of attributes
- The width of an addressable word

- The size of the memory in number of words

All read accesses to the memory are transformed to `$memrd` cells and all write accesses to `$memwr` cells by the language frontend. These cells consist of independent read- and write-ports to the memory. The `\MEMID` parameter on these cells is used to link them together and to the RTLIL::Memory object they belong to.

The rationale behind using separate cells for the individual ports versus creating a large multiport memory cell right in the language frontend is that the separate `$memrd` and `$memwr` cells can be consolidated using resource sharing. As resource sharing is a non-trivial optimization problem where different synthesis tasks can have different requirements it lends itself to do the optimisation in separate passes and merge the RTLIL::Memory objects and `$memrd` and `$memwr` cells to multiport memory blocks after resource sharing is completed.

The `memory` pass performs this conversion and can (depending on the options passed to it) transform the memories directly to d-type flip-flops and address logic or yield multiport memory blocks (represented using `$mem` cells).

See Sec. 5.1.5 for details on the memory cell types.

4.3 Command Interface and Synthesis Scripts

Yosys reads and processes commands from synthesis scripts, command line arguments and an interactive command prompt. Yosys commands consist of a command name and an optional whitespace sparated list of arguments. Commands are terminated using the newline character or a semicolon (`;`). Empty lines and lines starting with the hash sign (`#`) are ignored. See Sec. 3.3 for an example synthesis script.

The command `help` can be used to access the command reference manual.

Most commands can operate not only on the entire design but also only on *selected* parts of the design. For example the command `dump` will print all selected objects in the current design while `dump foobar` will only print the module `foobar` and `dump *` will print the entire design regardless of the current selection.

The selection mechanism is very powerful. For example the command `dump */t:$add %x:+[A] */w:* %i` will print all wires that are connected to the `\A` port of a `$add` cell. A detailed documentation of the select framework can be found in the command reference for the `select` command.

4.4 Source Tree and Build System

The Yosys source tree is organized in the following top-level directories:

- `backends/`
  This directory contains a subdirectory for each of the backend modules.
- `frontends/`
  This directory contains a subdirectory for each of the frontend modules.
- `kernel/`
  This directory contains all the core functionality of Yosys. This includes the functions and definitions for working with the RTLIL data structures (`rtlil.h` and `rtlil.cc`), the main() function (`driver.cc`), the internal framework for generating log messages (`log.h` and `log.cc`), the internal framework for registering and calling passes (`register.h` and `register.cc`), some core commands that are not really passes (`select.cc`, `show.cc`, ...) and a couple of other small utility libraries.

- `passes/`
  This directory contains a subdirectory for each pass or group of passes. For example as of this writing the directory `passes/opt/` contains the code for seven passes: `opt`, `opt_const`, `opt_muxtree`, `opt_reduce`, `opt_rmdff`, `opt_rmunused` and `opt_share`.

- `techlibs/`
  This directory contains simulation models and standard implementations for the cells from the internal cell library.

- `tests/`
  This directory contains a couple of test cases. Most of the smaller tests are executed automatically when `make test` is called. The larger tests must be executed manually. Most of the larger tests require downloading external HDL source code and/or external tools. The tests range from comparing simulation results of the synthesized design to the original sources to logic equivalence checking of entire CPU cores.

The top-level Makefile includes `frontends/*/Makefile.inc`, `passes/*/Makefile.inc` and `backends/*/Makefile.inc`. So when extending Yosys it is enough to create a new directory in `frontends/`, `passes/` or `backends/` with your sources and a `Makefile.inc`. The Yosys kernel automatically detects all commands linked with Yosys. So it is not needed to add additional commands to a central list of commands.

A good starting point for reading example source code for learning how to write passes are `passes/opt/opt_rmdff.cc` and `passes/opt/opt_share.cc`.

See the top-level README file for a quick *Getting Started* guide and build instructions. Yosys is a pure Makefile based project.

Users of the Qt Creator IDE can generate a QT Creator project file using `make qtcreator`. Users of the Eclipse IDE can use the "Makefile Project with Existing Code" project type in the Eclipse "New Project" dialog (only available after the CDT plugin has been installed) to create an Eclipse Project for programming extensions to Yosys or just browsing the Yosys code base.

Chapter 5

Internal Cell Library

Most of the passes in Yosys operate on netlists, i.e. they only care about the RTLIL::Wire and RTLIL::Cell objects in an RTLIL::Module. This chapter discusses the cell types used by Yosys to represent a behavioural design internally.

This chapter is split in two parts. In the first part the internal RTL cells are covered. These cells are used to represent the design on a coarse grain level. Like in the original HDL code on this level the cells operate on vectors of signals and complex cells like adders exist. In the second part the internal gate cells are covered. These cells are used to represent the design on a fine-grain gate-level. All cells from this category operate on single bit signals.

5.1 RTL Cells

Most of the RTL cells closely resemble the operators available in HDLs such as Verilog or VHDL. Therefore Verilog operators are used in the following sections to define the behaviour of the RTL cells.

Note that all RTL cells have parameters indicating the size of inputs and outputs. When passes modify RTL cells they must always keep the values of these parameters in sync with the size of the signals connected to the inputs and outputs.

Simulation models for the RTL cells can be found in the file `techlibs/simlib.v` in the Yosys source tree.

5.1.1 Unary Operators

All unary RTL cells have one input port `\A` and one output port `\Y`. They also have the following parameters:

- `\A_SIGNED`
  Set to a non-zero value if the input `\A` is signed and therefore should be sign-extended when needed.
- `\A_WIDTH`
  The width of the input port `\A`.
- `\Y_WIDTH`
  The width of the output port `\Y`.

Table 5.1 lists all cells for unary RTL operators.

Note that `$reduce_or` and `$reduce_bool` actually represent the same logic function. But the HDL frontends generate them in different situations. A `$reduce_or` cell is generated when the prefix `|` operator is being used. A `$reduce_bool` cell is generated when a bit vector is used as a condition in an `if`-statement or `?:`-expression.

Verilog	Cell Type
Y = ~A	$not
Y = +A	$pos
Y = -A	$neg
Y = &A	$reduce_and
Y = |A	$reduce_or
Y = ^A	$reduce_xor
Y = ~^A	$reduce_xnor
Y = |A	$reduce_bool
Y = !A	$logic_not

Table 5.1: Cell types for unary operators with their corresponding Verilog expressions.

5.1.2 Binary Operators

All binary RTL cells have two input ports \A and \B and one output port \Y. They also have the following parameters:

- \A_SIGNED
  Set to a non-zero value if the input \A is signed and therefore should be sign-extended when needed.
- \A_WIDTH
  The width of the input port \A.
- \B_SIGNED
  Set to a non-zero value if the input \B is signed and therefore should be sign-extended when needed.
- \B_WIDTH
  The width of the input port \B.
- \Y_WIDTH
  The width of the output port \Y.

Table 5.2 lists all cells for binary RTL operators.

5.1.3 Multiplexers

Multiplexers are generated by the Verilog HDL frontend for ?:-expressions. Multiplexers are also generated by the proc pass to map the decision trees from RTLIL::Process objects to logic.

The simplest multiplexer cell type is $mux. Cells of this type have a \WIDTH parameter and data inputs \A and \B and a data ouput \Y, all of the specified width. This cell also has a single bit control input \S. If \S is 0 the value from the \A input is sent to the output, if it is 1 the value from the \B input is sent to the output. So the $mux cell implements the function `Y = S ? B : A`.

The $pmux cell is used to multiplex between many inputs using a one-hot select signal. Cells of this type have a \WIDTH and a \S_WIDTH parameter and inputs \A, \B, and \S and an output \Y. The \S input is \S_WIDTH bits wide. The \A input and the output are both \WIDTH bits wide and the \B input is \WIDTH*\S_WIDTH bits wide. When all bits of \S are zero, the value from \A input is sent to the output. If the n'th bit from \S is set, the value n'th \WIDTH bits wide slice of the \B input is sent to the output. When more than one bit from \S is set the output is undefined. Cells of this type are used to model "parallel cases" (defined by using the parallel_case attribute or detected by an optimization).

The $safe_pmux behaves similarly to the $pmux cell type. But when more than one bit of \S is set, it is guaranteed that this cell type will output the value of the \A input instead of an undefined value.

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

Verilog	Cell Type
`Y = A < B`	`$lt`
`Y = A <= B`	`$le`
`Y = A == B`	`$eq`
`Y = A != B`	`$ne`
`Y = A >= B`	`$ge`
`Y = A > B`	`$gt`
`Y = A + B`	`$add`
`Y = A - B`	`$sub`
`Y = A * B`	`$mul`
`Y = A / B`	`$div`
`Y = A % B`	`$mod`
`Y = A ** B`	`$pow`

Table 5.2: Cell types for binary operators with their corresponding Verilog expressions.

Behavioural code with cascaded `if-then-else`- and `case`-statements usually results in trees of multiplexer cells. Many passes (from various optimizations to FSM extraction) heavily depend on these multiplexer trees to understand dependencies between signals. Therefore optimizations should not break these multiplexer trees (e.g. by replacing a multiplexer between a calculated signal and a constant zero with an `$and` gate).

5.1.4 Registers

D-Type Flip-Flops are represented by `$dff` cells. These cells have a clock port `\CLK`, an input port `\D` and an output port `\Q`. The following parameters are available for $dff cells:

- `\WIDTH`
  The width of input `\D` and output `\Q`.
- `\CLK_POLARITY`
  Clock is active on the positive edge if this parameter has the value `1'b1` and on the negative edge if this parameter is `1'b0`.

D-Type Flip-Flops with asynchronous resets are represented by `$adff` cells. As the `$dff` cells they have `\CLK`, `\D` and `\Q` ports. In addition they also have a single-bit `\ARST` input port for the reset pin and the following additional two parameters:

- `\ARST_POLARITY`
  The asynchronous reset is high-active if this parameter has the value `1'b1` and low-active if this parameter is `1'b0`.
- `\ARST_VALUE`
  The state of `\Q` will be set to this value when the reset is active.

Note that the `$adff` cell can only be used when the reset value is constant.

Usually these cells are generated by the `proc` pass using the information in the designs RTLIL::Process objects.

FIXME:
Add information about `$sr` cells (set-reset flip-flops) and d-type latches.

5.1.5 Memories

Memories are either represented using RTLIL::Memory objects and `$memrd` and `$memwr` cells or simply by using `$mem` cells.

In the first alternative the RTLIL::Memory objects hold the general metadata for the memory (bit width, size in number of words, etc.) and for each port a `$memrd` (read port) or `$memwr` (write port) cell is created. Having individual cells for read and write ports has the advantage that they can be consolidated using resource sharing passes. In some cases this drastically reduces the number of required ports on the memory cell.

The `$memrd` cells have a clock input `\CLK`, an address input `\ADDR` and a data output `\DATA`. They also have the following parameters:

- `\MEMID`
  The name of the RTLIL::Memory object that is associated with this read port.
- `\ABITS`
  The number of address bits (width of the `\ADDR` input port).
- `\WIDTH`
  The number of data bits (width of the `\DATA` output port).
- `\CLK_ENABLE`
  When this parameter is non-zero, the clock is used. Otherwise this read port is asynchronous and the `\CLK` input is not used.
- `\CLK_POLARITY`
  Clock is active on the positive edge if this parameter has the value `1'b1` and on the negative edge if this parameter is `1'b0`.

The `$memwr` cells have a clock input `\CLK`, an enable input `\EN`, an address input `\ADDR` and a data input `\DATA`. They also have the following parameters:

- `\MEMID`
  The name of the RTLIL::Memory object that is associated with this read port.
- `\ABITS`
  The number of address bits (width of the `\ADDR` input port).
- `\WIDTH`
  The number of data bits (width of the `\DATA` output port).
- `\CLK_ENABLE`
  When this parameter is non-zero, the clock is used. Otherwise this read port is asynchronous and the `\CLK` input is not used.
- `\CLK_POLARITY`
  Clock is active on positive edge if this parameter has the value `1'b1` and on the negative edge if this parameter is `1'b0`.

The HDL frontend models a memory using RTLIL::Memory objects and asynchronous `$memrd` and `$memwr` cells. The `memory` pass (i.e. its various sub-passes) migrates `$dff` cells into the `$memrd` and `$memwr` cells making them synchronous, then converts them to a single `$mem` cell and (optionally) maps this cell type to `$dff` cells for the individual words and multiplexer-based address decoders for the read and write interfaces. When the last step is disabled or not possible, a `$mem` cell is left in the design.

The `$mem` cell provides the following parameters:

- \MEMID
  The name of the original RTLIL::Memory object that became this $mem cell.
- \SIZE
  The number of words in the memory.
- \ABITS
  The number of address bits.
- \WIDTH
  The number of data bits per word.
- \RD_PORTS
  The number of read ports on this memory cell.
- \RD_CLK_ENABLE
  This parameter is \RD_PORTS bits wide, containing a clock enable bit for each read port.
- \RD_CLK_POLARITY
  This parameter is \RD_PORTS bits wide, containing a clock polarity bit for each read port.
- \WR_PORTS
  The number of write ports on this memory cell.
- \WR_CLK_ENABLE
  This parameter is \WR_PORTS bits wide, containing a clock enable bit for each write port.
- \WR_CLK_POLARITY
  This parameter is \WR_PORTS bits wide, containing a clock polarity bit for each write port.

The $mem cell has the following ports:

- \RD_CLK
  This input is \RD_PORTS bits wide, containing all clock signals for the read ports.
- \RD_ADDR
  This input is \RD_PORTS*\ABITS bits wide, containing all address signals for the read ports.
- \RD_DATA
  This input is \RD_PORTS*\WIDTH bits wide, containing all data signals for the read ports.
- \WR_CLK
  This input is \WR_PORTS bits wide, containing all clock signals for the write ports.
- \WR_EN
  This input is \WR_PORTS bits wide, containing all enable signals for the write ports.
- \WR_ADDR
  This input is \WR_PORTS*\ABITS bits wide, containing all address signals for the write ports.
- \WR_DATA
  This input is \WR_PORTS*\WIDTH bits wide, containing all data signals for the write ports.

The techmap pass can be used to manually map $mem cells to specialized memory cells on the target architecture, such as block ram resources on an FPGA.

5.1.6 Finite State Machines

FIXME:
Add a brief description of the $fsm cell type.

Verilog	Cell Type
`Y = ~A`	`$_INV_`
`Y = A & B`	`$_AND_`
`Y = A | B`	`$_OR_`
`Y = A ^ B`	`$_XOR_`
`Y = S ? B : A`	`$_MUX_`
always @(**negedge** C) Q <= D	`$_DFF_N_`
always @(**posedge** C) Q <= D	`$_DFF_P_`

ClkEdge	*RstLvl*	*RstVal*	Cell Type
negedge	0	0	`$_DFF_NN0_`
negedge	0	1	`$_DFF_NN1_`
negedge	1	0	`$_DFF_NP0_`
negedge	1	1	`$_DFF_NP1_`
posedge	0	0	`$_DFF_PN0_`
posedge	0	1	`$_DFF_PN1_`
posedge	1	0	`$_DFF_PP0_`
posedge	1	1	`$_DFF_PP1_`

Table 5.3: Cell types for gate level logic networks

5.2 Gates

For gate level logic networks, fixed function single bit cells are used that do not provide any parameters.

Simulation models for these cells can be found in the file `techlibs/stdcells_sim.v` in the Yosys source tree.

Table 5.3 lists all cell types used for gate level logic. The cell types `$_INV_`, `$_AND_`, `$_OR_`, `$_XOR_` and `$_MUX_` are used to model combinatorial logic. The cell types `$_DFF_N_` and `$_DFF_P_` represent d-type flip-flops.

The cell types `$_DFF_NN0_`, `$_DFF_NN1_`, `$_DFF_NP0_`, `$_DFF_NP1_`, `$_DFF_PN0_`, `$_DFF_PN1_`, `$_DFF_PP0_` and `$_DFF_PP1_` implement d-type flip-flops with asynchronous resets. The values in the table for these cell types relate to the following verilog code template, where *RstEdge* is **posedge** if *RstLvl* if `1`, and **negedge** otherwise.

```
always @(ClkEdge C, RstEdge R)
        if (R == RstLvl)
                Q <= RstVal;
        else
                Q <= D;
```

In most cases gate level logic networks are created from RTL networks using the `techmap` pass. The flip-flop cells from the gate level logic network can be mapped to physical flip-flop cells from a Liberty file using the `dfflibmap` pass. The combinatorial logic cells can be mapped to physical cells from a Liberty file via ABC [35] using the `abc` pass.

Chapter 6

Programming Yosys Extensions

FIXME:
This chapter will contain a guided tour to the Yosys APIs and conclude with an example module.

6.1 Programming with RTLIL

6.2 Internal Utility Libraries

6.3 Loadable Modules

Chapter 7

The Verilog and AST Frontends

This chapter provides an overview of the implementation of the Yosys Verilog and AST frontends. The Verilog frontend reads Verilog-2005 code and creates an abstract syntax tree (AST) representation of the input. This AST representation is then passed to the AST frontend that converts it to RTLIL data, as illustrated in Fig. 7.1.

7.1 Transforming Verilog to AST

The *Verilog frontend* converts the Verilog sources to an internal AST representation that closely resembles the structure of the original Verilog code. The Verilog frontend consists of three components, the *Preprocessor*, the *Lexer* and the *Parser*.

The source code to the Verilog frontend can be found in `frontends/verilog/` in the Yosys source tree.

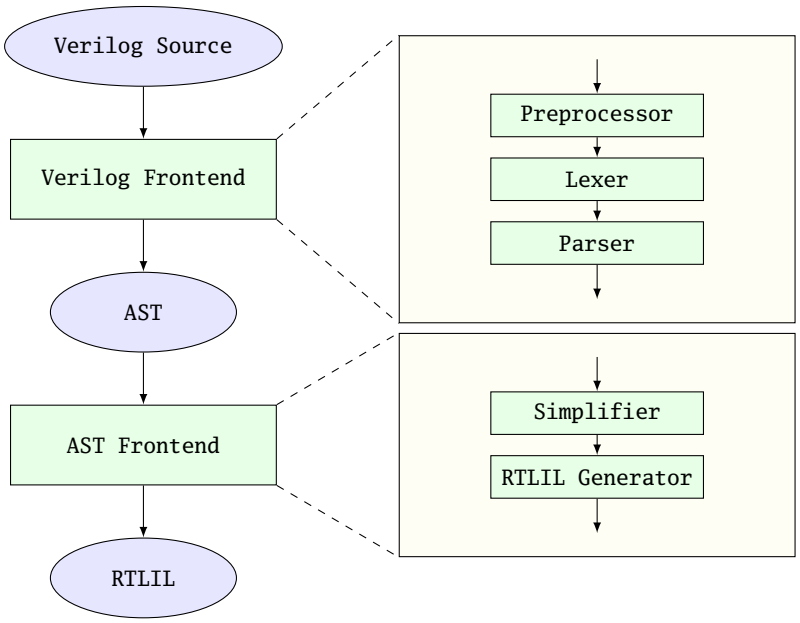

Figure 7.1: Simplified Verilog to RTLIL data flow

7.1.1 The Verilog Preprocessor

The Verilog preprocessor scans over the Verilog source code and interprets some of the Verilog compiler directives such as **`‘include`**, **`‘define`** and **`‘ifdef`**.

It is implemented as a C++ function that is passed a file descriptor as input and returns the pre-processed Verilog code as a `std::string`.

The source code to the Verilog Preprocessor can be found in `frontends/verilog/preproc.cc` in the Yosys source tree.

7.1.2 The Verilog Lexer

The Verilog Lexer is written using the lexer generator *flex* [22]. Its source code can be found in `frontends/verilog/lexer.l` in the Yosys source tree. The lexer does little more than identifying all keywords and literals recognised by the Yosys Verilog frontend.

The lexer keeps track of the current location in the verilog source code using some global variables. These variables are used by the constructor of AST nodes to annotate each node with the source code location it originated from.

Finally the lexer identifies and handles special comments such as "*`//synopsys translate_off`*" and "*`//synopsys full_case`*". (It is recommended to use **`‘ifdef`** constructs instead of the Synsopsys translate_on/off comments and attributes such as `(* full_case *)` over "*`//synopsys full_case`*" whenever possible.)

7.1.3 The Verilog Parser

The Verilog Parser is written using the parser generator *bison* [23]. Its source code can be found in `frontends/verilog/parser.y` in the Yosys source tree.

It generates an AST using the `AST::AstNode` data structure defined in `frontends/ast/ast.h`. An `AST::AstNode` object has the following properties:

- **The node type**
  This enum (`AST::AstNodeType`) specifies the role of the node. Table 7.1 contains a list of all node types.
- **The child nodes**
  This is a list of pointers to all children in the abstract syntax tree.
- **Attributes**
  As almost every AST node might have Verilog attributes assigned to it, the `AST::AstNode` has direct support for attributes. Note that the attribute values are again AST nodes.
- **Node content**
  Each node might have additional content data. A series of member variables exist to hold such data. For example the member `std::string str` can hold a string value and is used e.g. in the `AST_IDENTIFIER` node type to store the identifier name.
- **Source code location**
  Each `AST::AstNode` is automatically annotated with the current source code location by the `AST::AstNode` constructor. It is stored in the `std::string filename` and `int linenum` member variables.

The `AST::AstNode` constructor can be called with up to two child nodes that are automatically added to the list of child nodes for the new object. This simplifies the creation of AST nodes for simple expressions a bit. For example the bison code for parsing multiplications:

```
1       basic_expr '*' attr basic_expr {
2               $$ = new AstNode(AST_MUL, $1, $4);
3               append_attr($$, $3);
4       } |
```

The generated AST data structure is then passed directly to the AST frontend that performs the actual conversion to RTLIL.

Note that the Yosys command `read_verilog` provides the options `-yydebug` and `-dump_ast` that can be used to print the parse tree or abstract syntax tree respectively.

7.2 Transforming AST to RTLIL

The *AST Frontend* converts a set of modules in AST representation to modules in RTLIL representation and adds them to the current design. This is done in two steps: *simplification* and *RTLIL generation*.

The source code to the AST frontend can be found in `frontends/ast/` in the Yosys source tree.

7.2.1 AST Simplification

A full-featured AST is too complex to be transformed into RTLIL directly. Therefore it must first be brought into a simpler form. This is done by calling the `AST::AstNode::simplify()` method of all `AST_MODULE` nodes in the AST. This initiates a recursive process that performs the following transformations on the AST data structure:

AST Node Type	Corresponding Verilog Construct
`AST_NONE`	This Node type should never be used.
`AST_DESIGN`	This node type is used for the top node of the AST tree. It has no corresponding Verilog construct.
`AST_MODULE`, `AST_TASK`, `AST_FUNCTION`	**`module`**, **`task`** and **`function`**
`AST_WIRE`	**`input`**, **`output`**, **`wire`**, **`reg`** and **`integer`**
`AST_MEMORY`	Verilog Arrays
`AST_AUTOWIRE`	Created by the simplifier when an undeclared signal name is used.
`AST_PARAMETER`, `AST_LOCALPARAM`	**`parameter`** and **`localparam`**
`AST_PARASET`	Parameter set in cell instanciation
`AST_ARGUMENT`	Port connection in cell instanciation
`AST_RANGE`	Bit-Index in a signal or element index in array
`AST_CONSTANT`	A literal value
`AST_CELLTYPE`	The type of cell in cell instanciation
`AST_IDENTIFIER`	An Identifier (signal name in expression or cell/task/etc. name in other contexts)
`AST_PREFIX`	Construct an identifier in the form `<prefix>[<index>].<suffix>` (used only in advanced generate constructs)
`AST_FCALL`, `AST_TCALL`	Call to function or task
`AST_TO_SIGNED`, `AST_TO_UNSIGNED`	The `$signed()` and `$unsigned()` functions

Table 7.1: AST node types with their corresponding Verilog constructs.
(continued on next page)

AST Node Type	Corresponding Verilog Construct
AST_CONCAT AST_REPLICATE	The {...} and {...{...}} operators
AST_BIT_NOT, AST_BIT_AND, AST_BIT_OR, AST_BIT_XOR, AST_BIT_XNOR	The bitwise operators ~, &, |, ^ and ~^
AST_REDUCE_AND, AST_REDUCE_OR, AST_REDUCE_XOR, AST_REDUCE_XNOR	The unary reduction operators ~, &, |, ^ and ~^
AST_REDUCE_BOOL	Conversion from multi-bit value to boolian value (equivialent to AST_REDUCE_OR)
AST_SHIFT_LEFT, AST_SHIFT_RIGHT, AST_SHIFT_SLEFT, AST_SHIFT_SRIGHT	The shift operators <<, >>, <<< and >>>
AST_LT, AST_LE, AST_EQ, AST_NE, AST_GE, AST_GT	The relational operators <, <=, ==, !=, >= and >
AST_ADD, AST_SUB, AST_MUL, AST_DIV, AST_MOD, AST_POW	The binary operators +, -, *, /, % and **
AST_POS, AST_NEG	The prefix operators + and -
AST_LOGIC_AND, AST_LOGIC_OR, AST_LOGIC_NOT	The logic operators &&, || and !
AST_TERNARY	The ternary ?:-operator
AST_MEMRD AST_MEMWR	Read and write memories. These nodes are generated by the AST simplifier for writes/reads to/from Verilog arrays.
AST_ASSIGN	An **assign** statement
AST_CELL	A cell instanciation
AST_PRIMITIVE	A primitive cell (**and**, **nand**, **or**, etc.)
AST_ALWAYS, AST_INITIAL	Verilog **always**- and **initial**-blocks
AST_BLOCK	A **begin**-**end**-block
AST_ASSIGN_EQ. AST_ASSIGN_LE	Blocking (=) and nonblocking (<=) assignments within an **always**- or **initial**-block
AST_CASE. AST_COND, AST_DEFAULT	The **case** (**if**) statements, conditions within a case and the default case respectively
AST_FOR	A **for**-loop witn an **always**- or **initial**-block
AST_GENVAR, AST_GENBLOCK, AST_GENFOR, AST_GENIF	The **genvar** and **generate** keywords and **for** and **if** within a generate block.
AST_POSEDGE, AST_NEGEDGE, AST_EDGE	Event conditions for **always** blocks.

Table 7.1: AST node types with their corresponding Verilog constructs.
(continuation from previous page)

- Inline all task and function calls.
- Evaluate all **generate**-statements and unroll all **for**-loops.
- Perform const folding where it is neccessary (e.g. in the value part of AST_PARAMETER, AST_LOCALPARAM, AST_PARASET and AST_RANGE nodes).
- Replace AST_PRIMITIVE nodes with appropriate AST_ASSIGN nodes.
- Replace dynamic bit ranges in the left-hand-side of assignments with AST_CASE nodes with AST_COND children for each possible case.
- Detect array access patterns that are too complicated for the RTLIL::Memory abstraction and replace them with a set of signals and cases for all reads and/or writes.
- Otherwise replace array accesses with AST_MEMRD and AST_MEMWR nodes.

In addition to these transformations, the simplifier also annotates the AST with additional information that is needed for the RTLIL generator, namely:

- All ranges (width of signals and bit selections) are not only const folded but (when a constant value is found) are also written to member variables in the `AST_RANGE` node.
- All identifiers are resolved and all `AST_IDENTIFIER` nodes are annotated with a pointer to the AST node that contains the declaration of the identifier. If no declaration has been found, an `AST_AUTOWIRE` node is created and used for the annotation.

This produces an AST that is fairly easy to convert to the RTLIL format.

7.2.2 Generating RTLIL

After AST simplification, the `AST::AstNode::genRTLIL()` method of each `AST_MODULE` node in the AST is called. This initiates a recursive process that generates equivialent RTLIL data for the AST data.

The `AST::AstNode::genRTLIL()` method returns an `RTLIL::SigSpec` structure. For nodes that represent expressions (operators, constants, signals, etc.), the cells needed to implement the calculation described by the expression are created and the resulting signal is returned. That way it is easy to generate the circuits for large expressions using depth-first recursion. For nodes that do not represent an expression (such as `AST_CELL`), the corresponding circuit is generated and an empty `RTLIL::SigSpec` is returned.

7.3 Synthesizing Verilog always Blocks

For behavioural Verilog code (code utilizing **`always`**- and **`initial`**-blocks) it is necessary to also generate `RTLIL::Process` objects. This is done in the following way:

- Whenever `AST::AstNode::genRTLIL()` encounters an **`always`**- or **`initial`**-block, it creates an instance of `AST_INTERNAL::ProcessGenerator`. This object then generates the `RTLIL::Process` object for the block. It also calls `AST::AstNode::genRTLIL()` for all right-hand-side expressions contained within the block.
- First the `AST_INTERNAL::ProcessGenerator` creates a list of all signals assigned within the block. It then creates a set of temporary signals using the naming scheme `$`<*number*>`\`<*original_name*> for each of the assigned signals.
- Then an `RTLIL::Process` is created that assigns all intermediate values for each left-hand-side signal to the temporary signal in its `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree.
- Finally a `RTLIL::SyncRule` is created for the `RTLIL::Process` that assigns the temporary signals for the final values to the actual signals.
- Calls to `AST::AstNode::genRTLIL()` are generated for right hand sides as needed. When blocking assignments are used, `AST::AstNode::genRTLIL()` is configured using global variables to use the temporary signals that hold the correct intermediate values whenever one of the previously assigned signals is used in an expression.

Unfortunately the generation of a correct `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree for behavioural code is a non-trivial task. The AST frontend solves the problem using the approach described on the following pages. The following example illustrates what the algorithm is supposed to do. Consider the following Verilog code:

```
always @(posedge clock) begin
        out1 = in1;
        if (in2)
                out1 = !out1;
        out2 <= out1;
        if (in3)
                out2 <= out2;
        if (in4)
                if (in5)
                        out3 <= in6;
                else
                        out3 <= in7;
        out1 = out1 ^ out2;
end
```

This is translated by the Verilog and AST frontends into the following RTLIL code (attributes, cell parameters and wire declarations not included):

```
cell $logic_not $logic_not$<input>:4$2
  connect \A \in1
  connect \Y $logic_not$<input>:4$2_Y
end
cell $xor $xor$<input>:13$3
  connect \A $1\out1[0:0]
  connect \B \out2
  connect \Y $xor$<input>:13$3_Y
end
process $proc$<input>:1$1
  assign $0\out3[0:0] \out3
  assign $0\out2[0:0] $1\out1[0:0]
  assign $0\out1[0:0] $xor$<input>:13$3_Y
  switch \in2
    case 1'1
      assign $1\out1[0:0] $logic_not$<input>:4$2_Y
    case
      assign $1\out1[0:0] \in1
  end
  switch \in3
    case 1'1
      assign $0\out2[0:0] \out2
    case
  end
  switch \in4
    case 1'1
      switch \in5
        case 1'1
          assign $0\out3[0:0] \in6
        case
          assign $0\out3[0:0] \in7
      end
    case
  end
  sync posedge \clock
    update \out1 $0\out1[0:0]
```

```
37      update \out2 $0\out2[0:0]
38      update \out3 $0\out3[0:0]
39  end
```

Note that the two operators are translated into separate cells outside the generated process. The signal `out1` is assigned using blocking assignments and therefore `out1` has been replaced with a different signal in all expressions after the initial assignment. The signal `out2` is assigned using nonblocking assignments and therefore is not substituted on the right-hand-side expressions.

The `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree must be interpreted the following way:

- On each case level (the body of the process is the *root case*), first the actions on this level are evaluated and then the switches within the case are evaluated. (Note that the last assignment on line 13 of the Verilog code has been moved to the beginning of the RTLIL process to line 13 of the RTLIL listing.)

  I.e. the special cases deeper in the switch hierarchy override the defaults on the upper levels. The assignments in lines 12 and 22 of the RTLIL code serve as an example for this.

  Note that in contrast to this, the order within the `RTLIL::SwitchRule` objects within a `RTLIL::CaseRule` is preserved with respect to the original AST and Verilog code.

- The whole `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree describes an asynchronous circuit. I.e. the decision tree formed by the switches can be seen independently for each assigned signal. Whenever one assigned signal changes, all signals that depend on the changed signals are to be updated. For example the assignments in lines 16 and 18 in the RTLIL code in fact influence the assignment in line 12, even though they are in the "wrong order".

The only synchronous part of the process is in the `RTLIL::SyncRule` object generated at line 35 in the RTLIL code. The sync rule is the only part of the process where the original signals are assigned. The synchronization event from the original Verilog code has been translated into the synchronization type (`posedge`) and signal (`\clock`) for the `RTLIL::SyncRule` object. In the case of this simple example the `RTLIL::SyncRule` object is later simply transformed into a set of d-type flip-flops and the `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree to a decision tree using multiplexers.

In more complex examples (e.g. asynchronous resets) the part of the `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree that describes the asynchronous reset must first be transformed to the correct `RTLIL::SyncRule` objects. This is done by the `proc_adff` pass.

7.3.1 The ProcessGenerator Algorithm

The `AST_INTERNAL::ProcessGenerator` uses the following internal state variables:

- `subst_rvalue_from` and `subst_rvalue_to`
  These two variables hold the replacement pattern that should be used by `AST::AstNode::genRTLIL()` for signals with blocking assignments. After initialization of `AST_INTERNAL::ProcessGenerator` these two variables are empty.

- `subst_lvalue_from` and `subst_lvalue_to`
  These two variables contain the mapping from left-hand-side signals (`\`<*name*>) to the current temporary signal for the same thing (initially `$0\`<*name*>).

- `current_case`
  A pointer to a `RTLIL::CaseRule` object. Initially this is the root case of the generated `RTLIL::Process`.

As the algorithm runs these variables are continously modified as well as pushed to the stack and later restored to their earlier values by popping from the stack.

On startup the ProcessGenerator generates a new `RTLIL::Process` object with an empty root case and initializes its state variables as described above. Then the `RTLIL::SyncRule` objects are created using the synchronization events from the `AST_ALWAYS` node and the initial values of `subst_lvalue_from` and `subst_lvalue_to`. Then the AST for this process is evaluated recursively.

During this recursive evaluation, three different relevant types of AST nodes can be discovered: `AST_ASSIGN_LE` (nonblocking assignments), `AST_ASSIGN_EQ` (blocking assignments) and `AST_CASE` (**`if`** or **`case`** statement).

7.3.1.1 Handling of Nonblocking Assignments

When an `AST_ASSIGN_LE` node is discovered, the following actions are performed by the ProcessGenerator:

- The left-hand-side is evaluated using `AST::AstNode::genRTLIL()` and mapped to a temporary signal name using `subst_lvalue_from` and `subst_lvalue_to`.
- The right-hand-side is evaluated using `AST::AstNode::genRTLIL()`. For this call, the values of `subst_rvalue_from` and `subst_rvalue_to` are used to map blocking-assigned signals correctly.
- Remove all assignments to the same left-hand-side as this assignment from the `current_case` and all cases within it.
- Add the new assignment to the `current_case`.

7.3.1.2 Handling of Blocking Assignments

When an `AST_ASSIGN_EQ` node is discovered, the following actions are performed by the ProcessGenerator:

- Perform all the steps that would be performed for a nonblocking assignment (see above).
- Remove the found left-hand-side (before lvalue mapping) from `subst_rvalue_from` and also remove the respective bits from `subst_rvalue_to`.
- Append the found left-hand-side (before lvalue mapping) to `subst_rvalue_from` and append the found right-hand-side to `subst_rvalue_to`.

7.3.1.3 Handling of Cases and if-Statements

When an `AST_CASE` node is discovered, the following actions are performed by the ProcessGenerator:

- The values of `subst_rvalue_from`, `subst_rvalue_to`, `subst_lvalue_from` and `subst_lvalue_to` are pushed to the stack.
- A new `RTLIL::SwitchRule` object is generated, the selection expression is evaluated using `AST::AstNode::genRTLIL()` (with the use of `subst_rvalue_from` and `subst_rvalue_to`) and added to the `RTLIL::SwitchRule` object and the obect is added to the `current_case`.
- All lvalues assigned to within the `AST_CASE` node using blocking assignments are collected and saved in the local variable `this_case_eq_lvalue`.
- New temporary signals are generated for all signals in `this_case_eq_lvalue` and stored in `this_case_eq_ltemp`.

- The signals in `this_case_eq_lvalue` are mapped using `subst_rvalue_from` and `subst_rvalue_to` and the resulting set of signals is stored in `this_case_eq_rvalue`.

Then the following steps are performed for each `AST_COND` node within the `AST_CASE` node:

- Set `subst_rvalue_from`, `subst_rvalue_to`, `subst_lvalue_from` and `subst_lvalue_to` to the values that have been pushed to the stack.
- Remove `this_case_eq_lvalue` from `subst_lvalue_from`/`subst_lvalue_to`.
- Append `this_case_eq_lvalue` to `subst_lvalue_from` and append `this_case_eq_ltemp` to `subst_lvalue_to`.
- Push the value of `current_case`.
- Create a new `RTLIL::CaseRule`. Set `current_case` to the new object and add the new object to the `RTLIL::SwitchRule` created above.
- Add an assignment from `this_case_eq_rvalue` to `this_case_eq_ltemp` to the new `current_case`.
- Evaluate the compare value for this case using `AST::AstNode::genRTLIL()` (with the use of `subst_rvalue_from` and `subst_rvalue_to`) modify the new `current_case` accordingly.
- Recursion into the children of the `AST_COND` node.
- Restore `current_case` by popping the old value from the stack.

Finally the following steps are performed:

- The values of `subst_rvalue_from`, `subst_rvalue_to`, `subst_lvalue_from` and `subst_lvalue_to` are popped from the stack.
- The signals from `this_case_eq_lvalue` are removed from the `subst_rvalue_from`/`subst_rvalue_to`-pair.
- The value of `this_case_eq_lvalue` is appended to `subst_rvalue_from` and the value of `this_case_eq_ltemp` is appended to `subst_rvalue_to`.
- Map the signals in `this_case_eq_lvalue` using `subst_lvalue_from`/`subst_lvalue_to`.
- Remove all assignments to signals in `this_case_eq_lvalue` in `current_case` and all cases within it.
- Add an assignment from `this_case_eq_ltemp` to `this_case_eq_lvalue` to `current_case`.

7.3.1.4 Further Analysis of the Algorithm for Cases and if-Statements

With respect to nonblocking assignments the algorithm is easy: later assignments invalidate earlier assignments. For each signal assigned using nonblocking assignments exactly one temporary variable is generated (with the `$0`-prefix) and this variable is used for all assignments of the variable.

Note how all the `_eq_`-variables become empty when no blocking assignments are used and many of the steps in the algorithm can then be ignored as a result of this.

For a variable with blocking assignments the algorithm shows the following behaviour: First a new temporary variable is created. This new temporary variable is then registered as the assignment target for all assignments for this variable within the cases for this `AST_CASE` node. Then for each case the new temporary variable is first assigned the old temporary variable. This assignment is overwritten if the variable is actually assigned in this case and is kept as a default value otherwise.

This yields an `RTLIL::CaseRule` that assigns the new temporary variable in all branches. So when all cases have been processed a final assignment is added to the containing block that assigns the new temporary variable to the old one. Note how this step always overrides a previous assignment to the old temporary variable. Other than nonblocking assignments, the old assignment could still have an effect somewhere in the design, as there have been calls to `AST::AstNode::genRTLIL()` with a `subst_rvalue_from`/`subst_rvalue_to`-tuple that contained the right-hand-side of the old assignment.

7.3.2 The proc pass

The ProcessGenerator converts a behavioural model in AST representation to a behavioural model in `RTLIL::Process` representation. The actual conversion from a behavioural model to an RTL representation is performed by the `proc` pass and the passes it launches:

- `proc_clean` and `proc_rmdead`
  These two passes just clean up the `RTLIL::Process` structure. The `proc_clean` pass removes empty parts (eg. empty assignments) from the process and `proc_rmdead` detects and removes unreachable branches from the process's decision trees.
- `proc_arst`
  This pass detects processes that describe d-type flip-flops with asynchronous resets and rewrites the process to better reflect what they are modelling: Before this pass, an asynchronous reset has two edge-sensitive sync rules and one top-level `RTLIL::SwitchRule` for the reset path. After this pass the sync rule for the reset is level-sensitive and the top-level `RTLIL::SwitchRule` has been removed.
- `proc_mux`
  This pass converts the `RTLIL::CaseRule`/`RTLIL::SwitchRule`-tree to a tree of multiplexers per written signal. After this, the `RTLIL::Process` structure only contains the `RTLIL::SyncRule`s that describe the output registers.
- `proc_dff`
  This pass replaces the `RTLIL::SyncRule`s to d-type flip-flops (with asynchronous resets if neccessary).
- `proc_clean`
  A final call to `proc_clean` removes the now empty `RTLIL::Process` objects.

Performing these last processing steps in passes instead of in the Verilog frontend has two important benefits:

First it improves the transparency of the process. Everything that happens in a seperate pass is easier to debug, as the RTLIL data structures can be easily investigated before and after each of the steps.

Second it improves flexibility. This scheme can easily be extended to support other types of storage-elements, such as sr-latches or d-latches, without having to extend the actual Verilog frontend.

7.4 Synthesizing Verilog Arrays

FIXME:
Add some information on the generation of `$memrd` and `$memwr` cells and how they are processsed in the `memory` pass.

7.5 Synthesizing Parametric Designs

FIXME:
Add some information on the `RTLIL::Module::derive()` method and how it is used to synthesize parametric modules via the `hierarchy` pass.

Chapter 8

Optimizations

Yosys employs a number of optimizations to generate better and cleaner results. This chapter outlines these optimizations.

8.1 Simple Optimizations

The Yosys pass `opt` runs a number of simple optimizations. This includes removing unused signals and cells and const folding. It is recommended to run this pass after each major step in the synthesis script. At the time of this writing the `opt` pass executes the following passes that each perform a simple optimization:

- Once at the beginning of `opt`:
  - `opt_const`
  - `opt_share -nomux`
- Repeat until result is stable:
  - `opt_muxtree`
  - `opt_reduce`
  - `opt_share`
  - `opt_rmdff`
  - `opt_clean`
  - `opt_const`

The following section describes each of the `opt_*` passes.

8.1.1 The opt_const pass

This pass performs const folding on the internal combinational cell types described in Chap. 5. This means a cell with all constant inputs is replaced with the constant value this cell drives. In some cases this pass can also optimize cells with some constant inputs.

Table 8.1 shows the replacement rules used for optimizing an `$_AND_` gate. The first three rules implement the obvious const folding rules. Note that 'any' might include dynamic values calculated by other parts of the circuit. The following three lines propagate undef (X) states. These are the only three cases in which it is allowed to propagate an undef according to Sec. 5.1.10 of IEEE Std. 1364-2005 [Ver06].

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

Table 8.1: Const folding rules for `$_AND_` cells as used in `opt_const`.

The next two lines assume the value 0 for undef states. These two rules are only used if no other subsitutions are possible in the current module. If other substitutions are possible they are performed first, in the hope that the 'any' will change to an undef value or a 1 and therefore the output can be set to undef.

The last two lines simply replace an `$_AND_` gate with one constant-1 input with a buffer.

Besides this basic const folding the `opt_const` pass can replace 1-bit wide `$eq` and `$ne` cells with buffers or not-gates if one input is constant.

The `opt_const` pass is very conservative regarding optimizing `$mux` cells, as these cells are often used to model decision-trees and breaking these trees can interfere with other optimizations.

8.1.2 The opt_muxtree pass

This pass optimizes trees of multiplexer cells by analyzing the select inputs. Consider the following simple example:

```
1 module uut(a, y);
2 input a;
3 output [1:0] y = a ? (a ? 1 : 2) : 3;
4 endmodule
```

The output can never be 2, as this would require `a` to be 1 for the outer multiplexer and 0 for the inner multiplexer. The `opt_muxtree` pass detects this contradiction and replaces the inner multiplexer with a constant 1, yielding the logic for `y = a ? 1 : 3`.

8.1.3 The opt_reduce pass

This is a simple optimization pass that identifies and consolidates identical input bits to `$reduce_and` and `$reduce_or` cells. It also sorts the input bits to ease identification of shareable `$reduce_and` and `$reduce_or` cells in other passes.

This pass also identifies and consolidates identical inputs to multiplexer cells. In this case the new shared select bit is driven using a `$reduce_or` cell that combines the original select bits.

Lastly this pass consolidates trees of `$reduce_and` cells and trees of `$reduce_or` cells to single large `$reduce_and` or `$reduce_or` cells.

These three simple optimizations are performed in a loop until a stable result is produced.

8.1.4 The opt_rmdff pass

This pass identifies single-bit d-type flip-flops (`$_DFF_*`, `$dff`, and `$adff` cells) with a constant data input and replaces them with a constant driver.

8.1.5 The opt_clean pass

This pass identifies unused signals and cells and removes them from the design. It also creates an `\unused_bits` attribute on wires with unused bits. This attribute can be used for debugging or by other optimization passes.

8.1.6 The opt_share pass

This pass performs trivial resource sharing. This means that this pass identifies cells with identical inputs and replaces them with a single instance of the cell.

The option `-nomux` can be used to disable resource sharing for multiplexer cells (`$mux`, `$pmux`, and `$safe_pmux`). This can be useful as it prevents multiplexer trees to be merged, which might prevent `opt_muxtree` to identify possible optimizations.

8.2 FSM Extraction and Encoding

The `fsm` pass performs finite-state-machine (FSM) extraction and recoding. The `fsm` pass simply executes the following other passes:

- Identify and extract FSMs:
  - `fsm_detect`
  - `fsm_extract`
- Basic optimizations:
  - `fsm_opt`
  - `opt_clean`
  - `fsm_opt`
- Expanding to nearby gate-logic (if called with `-expand`):
  - `fsm_expand`
  - `opt_clean`
  - `fsm_opt`
- Re-code FSM states (unless called with `-norecode`):
  - `fsm_recode`
- Print information about FSMs:
  - `fsm_info`
- Export FSMs in KISS2 file format (if called with `-export`):
  - `fsm_export`

- Map FSMs to RTL cells (unless called with `-nomap`):
  - `fsm_map`

The `fsm_detect` pass identifies FSM state registers and marks them using the `\fsm_encoding= "auto"` attribute. The `fsm_extract` extracts all FSMs marked using the `\fsm_encoding` attribute (unless `\fsm_encoding` is set to `"none"`) and replaces the corresponding RTL cells with a `$fsm` cell. All other `fsm_*` passes operate on these `$fsm` cells. The `fsm_map` call finally replaces the `$fsm` cells with RTL cells.

Note that these optimizations operate on an RTL netlist. I.e. the `fsm` pass should be executed after the `proc` pass has transformed all `RTLIL::Process` objects to RTL cells.

The algorithms used for FSM detection and extraction are influenced by a more general reported technique [STGR10].

8.2.1 FSM Detection

The `fsm_detect` pass identifies FSM state registers. It sets the `\fsm_encoding= "auto"` attribute on any (multi-bit) wire that matches the following description:

- Does not already have the `\fsm_encoding` attribute.
- Is not an output of the containing module.
- Is driven by single `$dff` or `$adff` cell.
- The `\D`-Input of this `$dff` or `$adff` cell is driven by a multiplexer tree that only has constants or the old state value on its leaves.
- The state value is only used in the said multiplexer tree or by simple relational cells that compare the state value to a constant (usually `$eq` cells).

This heuristic has proven to work very well. It is possible to overwrite it by setting `\fsm_encoding= "auto"` on registers that should be considered FSM state registers and setting `\fsm_encoding= "none"` on registers that match the above criteria but should not be considered FSM state registers.

8.2.2 FSM Extraction

The `fsm_extract` pass operates on all state signals marked with the `\fsm_encoding` (`!= "none"`) attribute. For each state signal the following information is determined:

- The state registers
- The asynchronous reset state if the state registers use asynchronous reset
- All states and the control input signals used in the state transition functions
- The control output signals calculated from the state signals and control inputs
- A table of all state transitions and corresponding control inputs- and outputs

The state registers (and asynchronous reset state, if applicable) is simply determined by identifying the driver for the state signal.

From there the `$mux`-tree driving the state register inputs is recursively traversed. All select inputs are control signals and the leaves of the `$mux`-tree are the states. The algorithm fails if a non-constant leaf that is not the state signal itself is found.

The list of control outputs is initialized with the bits from the state signal. It is then extended by adding all values that are calculated by cells that compare the state signal with a constant value.

In most cases this will cover all uses of the state register, thus rendering the state encoding arbitrary. If however a design uses e.g. a single bit of the state value to drive a control output directly, this bit of the state signal will be transformed to a control output of the same value.

Finally, a transition table for the FSM is generated. This is done by using the `ConstEval` C++ helper class (defined in `kernel/consteval.h`) that can be used to evaluate parts of the design. The `ConstEval` class can be asked to calculate a given set of result signals using a set of signal-value assignments. It can also be passed a list of stop-signals that abort the `ConstEval` algorithm if the value of a stop-signal is needed in order to calculate the result signals.

The `fsm_extract` pass uses the `ConstEval` class in the following way to create a transition table. For each state:

1. Create a `ConstEval` object for the module containing the FSM
2. Add all control inputs to the list of stop signals
3. Set the state signal to the current state
4. Try to evaluate the next state and control output
5. If step 4 was not successful:
   - Recursively goto step 4 with the offending stop-signal set to 0.
   - Recursively goto step 4 with the offending stop-signal set to 1.
6. If step 4 was successful: Emit transition

Finally a `$fsm` cell is created with the generated transition table and added to the module. This new cell is connected to the control signals and the old drivers for the control outputs are disconnected.

8.2.3 FSM Optimization

The `fsm_opt` pass performs basic optimizations on `$fsm` cells (not including state recoding). The following optimizations are performed (in this order):

- Unused control outputs are removed from the `$fsm` cell. The attribute `\unused_bits` (that is usually set by the `opt_clean` pass) is used to determine which control outputs are unused.
- Control inputs that are connected to the same driver are merged.
- When a control input is driven by a control output, the control input is removed and the transition table altered to give the same performance without the external feedback path.
- Entries in the transition table that yield the same output and only differ in the value of a single control input bit are merged and the different bit is removed from the sensitivity list (turned into a don't-care bit).
- Constant inputs are removed and the transition table is alterered to give an unchanged behaviour.
- Unused inputs are removed.

8.2.4 FSM Recoding

The `fsm_recode` pass assigns new bit pattern to the states. Usually this also implies a change in the width of the state signal. At the moment of this writing only one-hot encoding with all-zero for the reset state is supported.

The `fsm_recode` pass can also write a text file with the changes performed by it that can be used when verifying designs synthesized by Yosys using Synopsys Formality [29].

8.3 Logic Optimization

Yosys can perform multi-level combinational logic optimization on gate-level netlists using the external program ABC [35]. The `abc` pass extracts the combinational gate-level parts of the design, passes it through ABC, and re-integrates the results. The `abc` pass can also be used to perform other operations using ABC, such as technology mapping (see Sec. 9.3 for details).

Chapter 9

Technology Mapping

Previous chapters outlined how HDL code is transformed into an RTL netlist. The RTL netlist is still based on abstract coarse-grain cell types like arbitrary width adders and even multipliers. This chapter covers how an RTL netlist is transformed into a functionally equivialent netlist utililizing the cell types available in the target architecture.

Technology mapping is often performed in two phases. In the first phase RTL cells are mapped to an internal library of single-bit cells (see Sec. 5.2). In the second phase this netlist of internal gate types is transformed to a netlist of gates from the target technology library.

When the target architecture provides coarse-grain cells (such as block ram or ALUs), these must be mapped to directly form the RTL netlist, as information on the coarse-grain structure of the design is lost when it is mapped to bit-width gate types.

9.1 Cell Substitution

The simplest form of technology mapping is cell substitution, as performed by the `techmap` pass. This pass, when provided with a Verilog file that implements the RTL cell types using simpler cells, simply replaces the RTL cells with the provided implementation.

When no map file is provided, `techmap` uses a built-in map file that maps the Yosys RTL cell types to the internal gate library used by Yosys. The curious reader may find this map file as `techlibs/stdcells.v` in the Yosys source tree.

Additional features have been added to `techmap` to allow for conditional mapping of cells (see `help techmap` or Sec. C.50). This can for example be usefull if the target architecture supports hardware multipliers for certain bit-widths but not for others.

A usual synthesis flow would first use the `techmap` pass to directly map some RTL cells to coarse-grain cells provided by the target architecture (if any) and then use techmap with the built-in default file to map the remaining RTL cells to gate logic.

9.2 Subcircuit Substitution

Sometimes the target architecture provides cells that are more powerful than the RTL cells used by Yosys. For example a cell in the target architecture that can calculate the absolute-difference of two numbers does not match any single RTL cell type but only combinations of cells.

For these cases Yosys provides the `extract` pass that can match a given set of modules against a design and identify the portions of the design that are identical (i.e. isomorphic subcircuits) to any of the given modules. These matched subcircuits are then replaced by instances of the given modules.

The `extract` pass also finds basic variations of the given modules, such as swapped inputs on commutative cell types.

In addition to this the `extract` pass also has limited support for frequent subcircuit mining, i.e. the process of finding recurring subcircuits in the design. This has a few applications, including the design of new coarse-grain architectures [GW13].

The hard algorithmic work done by the `extract` pass (solving the isomorphic subcircuit problem and frequent subcircuit mining) is performed using the SubCircuit library that can also be used stand-alone without Yosys (see Sec. A.3).

9.3 Gate-Level Technology Mapping

On the gate-level the target architecture is usually described by a "Liberty file". The Liberty file format is an industry standard format that can be used to describe the behaviour and other properties of standard library cells [30].

Mapping a design utilizing the Yosys internal gate library (e.g. as a result of mapping it to this representation using the `techmap` pass) is performed in two phases.

First the register cells must be mapped to the registers that are available on the target architectures. The target architecture might not provide all variations of d-type flip-flops with positive and negative clock edge, high-active and low-active asynchronous set and/or reset, etc. Therefore the process of mapping the registers might add additional inverters to the design and thus it is important to map the register cells first.

Mapping of the register cells may be performed by using the `dfflibmap` pass. This pass expects a Liberty file as argument (using the `-liberty` option) and only uses the register cells from the Liberty file.

Secondly the combinational logic must be mapped to the target architecture. This is done using the external program ABC [35] via the `abc` pass by using the `-liberty` option to the pass. Note that in this case only the combinatorial cells are used from the cell library.

Occasionally Liberty files contain trade secrets (such as sensitive timing information) that cannot be shared freely. This complicates processes such as reporting bugs in the tools involved. When the information in the Liberty file used by Yosys and ABC are not part of the sensitive information, the additional tool `yosys-filterlib` (see Sec. B.2) can be used to strip the sensitive information from the Liberty file.

Chapter 10

Evaluation, Conclusion, Future Work

The Yosys source tree contains over 200 test cases[1] which are used in the `make test` make-target. Besides these there is an external Yosys benchmark and test case package that contains a few larger designs [43]. This package contains the designs listed in Tab. 10.1.

10.1 Correctness of Synthesis Results

The following measures were taken to increase the confidence in the correctness of the Yosys synthesis results:

- Yosys comes with a large selection[3] of small test cases that are evaluated when the command `make test` is executed. During development of Yosys it was shown that this collection of test cases is sufficient to catch most bugs. The following more sophisticated test procedures only caught a few additional bugs. Whenever this happend, an appropiate test case was added to the collection of small test cases for `make test` to ensure better testability of the feature in question in the future.
- The designs listed in Tab. 10.1 where validated using the formal verification tool Synopsys Formality[29]. The Yosys synthesis scripts used to synthesize the individual designs for this test are slightly different

[1] Most of this test cases are copied from HANA [32] or the ASIC-WORLD website [31].

[2] Number of gates determined using the Yosys synthesis script "`hierarchy -top $top; proc; opt; memory; opt; techmap; opt; abc; opt; flatten $top; hierarchy -top $top; abc; opt; select -count */c:*`".

[3] At the time of this writing 269 test cases.

Test-Design	Source	Gates[2]	Description / Comments
`aes_core`	IWLS2005	41,837	AES Cipher written by Rudolf Usselmann
`i2c`	IWLS2005	1,072	WISHBONE compliant I2C Master by Richard Herveille
`openmsp430`	OpenCores	7,173	MSP430 compatible CPU by Olivier Girard
`or1200`	OpenCores	42,675	The OpenRISC 1200 CPU by Damjan Lampret
`sasc`	IWLS2005	456	Simple Async. Serial Comm. Device by Rudolf Usselmann
`simple_spi`	IWLS2005	690	MC68HC11E based SPI interface by Richard Herveille
`spi`	IWLS2005	2,478	SPI IP core by Simon Srot
`ss_pcm`	IWLS2005	279	PCM IO Slave by Rudolf Usselmann
`systemcaes`	IWLS2005	6,893	AES core (using SystemC to Verilog) by Javier Castillo
`usb_phy`	IWLS2005	515	USB 1.1 PHY by Rudolf Usselmann

Table 10.1: Tests included in the yosys-tests package.

per design in order to broaden the coverage of Yosys features. The large majority of all errors encountered using these tests are false-negatives, mostly related to FSM encoding or signal naming in large array logic (such as in memory blocks). Therefore the `fsm_recode` pass was extended so it can be used to generate TCL commands for Synopsys Formality that describe the relationship between old and new state encodings. Also the method used to generate signal and cell names in the Verilog backend was slightly modified in order to improve the automatic matching of net names in Synopsys Formality. With these changes in place all designs in Tab. 10.1 validate successfully using Formality.

- VlogHammer [42] is a set of scripts that auto-generate a large collection of test cases[4] and synthesize them using Yosys and the following freely available propritary synthesis tools.
  - Xilinx Vivado WebPack (2013.2) [44]
  - Xilinx ISE (XST) WebPack (14.5) [44]
  - Altera Quartus II Web Edition (13.0) [33]

  The built-in SAT solver of Yosys is used to formally verify the Yosys RTL- and Gate-Level netlists against the netlists generated by this other tools.[5] When differences are found, the input pattern that result in different outputs are used for simulating the original Verilog code as well as the synthesis results using the following Verilog simulators.
  - Xilinx ISIM (from Xilinx ISE 14.5 [44])
  - Modelsim 10.1d (from Quartus II 13.0 [33])
  - Icarus Verilog (no specific version)

  The set of tests performed by VlogHammer systematically verify the correct behaviour of
  - Yosys Verilog Frontend and RTL generation
  - Yosys Gate-Level Technology Mapping
  - Yosys SAT Models for RTL- and Gate-Level cells
  - Yosys Constant Evaluator Models for RTL- and Gate-Level cells

  against the reference provided by the other tools. A few bugs related to sign extensions and bit-width extensions where found (and have been fixed meanwhile) using this approach. This test also revealed a small number of bugs in the other tools (i.e. Vivado, XST, Quartus, ISIM and Icarus Verilog; no bugs where found in Modelsim using vlogHammer so far).

Although complex software can never be expected to be fully bug-free [Kli67], it has been shown that Yosys is mature and feature-complete enough to handle most real-world cases correctly.

10.2 Quality of synthesis results

In this section an attempt to evaluate the quality of Yosys synthesis results is made. To this end the synthesis results of a commercial FPGA synthesis tool when presented with the original HDL code vs. when presented with the Yosys synthesis result are compared.

The OpenMSP430 and the OpenRISC 1200 test cases were synthesized using the following Yosys synthesis script:

```
1 hierarchy -check
2 proc; opt; fsm; opt; memory; opt
3 techmap; opt; abc; opt
```

[4] At the time of this writing over 6600 test cases.

[5] A SAT solver is a program that can solve the boolean satisfiability problem. The built-in SAT solver in Yosys can be used for formal equivalence checking, amongst other things. See Sec. C.40 for details.

The original RTL and the Yosys output where both passed to the Xilinx XST 14.5 FPGA synthesis tool. The following setting where used for XST:

```
1 -p artix7
2 -use_dsp48 NO
3 -iobuf NO
4 -ram_extract NO
5 -rom_extract NO
6 -fsm_extract YES
7 -fsm_encoding Auto
```

The results of this comparison is summarized in Tab. 10.2. The used FPGA resources (registers and LUTs) and performance (maximum frequency as reported by XST) are given per module (indentation indicates module hierarchy, the numbers are including all contained modules).

For most modules the results are very similar between XST and Yosys. XST is used in both cases for the final mapping of logic to LUTs. So this comparison only compares the high-level synthesis functions (such as FSM extraction and encoding) of Yosys and XST.

10.3 Conclusion and Future Work

Yosys is capable of correctly synthesizing real-world Verilog designs. The generated netlists are of a decent quality. However, in cases where dedicated hardware resources should be used for certain functions it is of course necessary to implement proper technology mapping for these functions in Yosys. This can be as easy as calling the `techmap` pass with an architecture-specific mapping file in the synthesis script. As no such thing has been done in the above tests, it is only natural that the resulting designs cannot benefit from these dedicated hardware resources.

Therefore future work includes the implementation of architecture-specific technology mappings besides additional frontends (VHDL), backends (EDIF), and above all else, application specific passes. After all, this was the main motivation for the development of Yosys in the first place.

	Without Yosys			With Yosys		
Module	Regs	LUTs	Max. Freq.	Regs	LUTs	Max. Freq.
openMSP430	689	2210	71 MHz	719	2779	53 MHz
omsp_clock_module	21	30	645 MHz	21	30	644 MHz
omsp_sync_cell	2	—	1542 MHz	2	—	1542 MHz
omsp_sync_reset	2	—	1542 MHz	2	—	1542 MHz
omsp_dbg	143	344	292 MHz	149	430	353 MHz
omsp_dbg_uart	76	135	377 MHz	79	139	389 MHz
omsp_execution_unit	266	911	80 MHz	266	1034	137 MHz
omsp_alu	—	202	—	—	263	—
omsp_register_file	231	478	285 MHz	231	506	293 MHz
omsp_frontend	115	340	178 MHz	118	527	206 MHz
omsp_mem_backbone	38	141	1087 MHz	38	144	1087 MHz
omsp_multiplier	73	397	129 MHz	102	1053	55 MHz
omsp_sfr	6	18	1023 MHz	6	20	1023 MHz
omsp_watchdog	24	53	362 MHz	24	70	360 MHz
or1200_top	7148	9969	135 MHz	7173	10238	108 MHz
or1200_alu	—	681	—	—	641	—
or1200_cfgr	—	11	—	—	11	—
or1200_ctrl	175	186	464 MHz	174	279	377 MHz
or1200_except	241	451	313 MHz	241	353	301 MHz
or1200_freeze	6	18	507 MHz	6	16	515 MHz
or1200_if	68	143	806 MHz	68	139	790 MHz
or1200_lsu	8	138	—	12	205	1306 MHz
or1200_mem2reg	—	60	—	—	66	—
or1200_reg2mem	—	29	—	—	29	—
or1200_mult_mac	394	2209	240 MHz	394	2230	241 MHz
or1200_amultp2_32x32	256	1783	240 MHz	256	1770	241 MHz
or1200_operandmuxes	65	129	1145 MHz	65	129	1145 MHz
or1200_rf	1041	1722	822 MHz	1042	1722	581 MHz
or1200_sprs	18	432	724 MHz	18	469	722 MHz
or1200_wbmux	33	93	—	33	78	—
or1200_dc_top	—	5	—	—	5	—
or1200_dmmu_top	2445	1004	—	2445	1043	—
or1200_dmmu_tlb	2444	975	—	2444	1013	—
or1200_du	67	56	859 MHz	67	56	859 MHz
or1200_ic_top	39	100	527 MHz	41	136	514 MHz
or1200_ic_fsm	40	42	408 MHz	40	75	484 MHz
or1200_pic	38	50	1169 MHz	38	50	1177 MHz
or1200_tt	64	112	370 MHz	64	186	437 MHz

Table 10.2: Synthesis results (as reported by XST) for OpenMSP430 and OpenRISC 1200

Appendix A

Auxilary Libraries

The Yosys source distribution contains some auxilary libraries that are bundled with Yosys.

A.1 SHA1

The files in `libs/sha1/` provide a SHA1 implementation written by Micael Hildenborg [37]. It is used for generating unique names when specializing parameterized modules.

A.2 BigInt

The files in `libs/bigint/` provide a library for performing arithmetic with arbitrary length integers. It is written by Matt McCutchen [38].

The BigInt library is used for evaluating constant expressions, e.g. using the `ConstEval` class provided in `kernel/consteval.h`.

A.3 SubCircuit

The files in `libs/subcircuit` provide a library for solving the subcircuit isomorphism problem. It is written by Clifford Wolf and based on the Ullmann Subgraph Isomorphism Algorithm [Ull76]. It is used by the `extract` pass (see `help extract` or Sec. C.6).

A.4 ezSAT

The files in `libs/ezsat` provide a library for simplifying generating CNF formulas for SAT solvers. It also contains bindings of MiniSAT. The ezSAT library is written by Clifford Wolf. It is used by the `sat` pass (see `help sat` or Sec. C.40).

Appendix B

Auxilary Programs

Besides the main `yosys` executable, the Yosys distribution contains a set of additional helper programs.

B.1 yosys-config

The `yosys-config` tool (an auto-generated shell-script) can be used to query compiler options and other information needed for building loadable modules for Yosys. FIXME: See Sec. 6 for details.

B.2 yosys-filterlib

The `yosys-filterlib` tool is a small utility that can be used to strip or extract information from a Liberty file. See Sec. 9.3 for details.

B.3 yosys-svgviewer

The `yosys-svgviewer` tool is a small Qt program that can be used to view SVG files. This tool is automatically launched by the `show` command when no `-format` and no `-viewer` option is passed to the command. See `help show` or Sec. C.46 for details.

Appendix C

Command Reference Manual

C.1 abc – use ABC for technology mapping

```
    abc [options] [selection]

This pass uses the ABC tool [1] for technology mapping of yosys's internal gate
library to a target architecture.

    -exe <command>
        use the specified command name instead of "yosys-abc" to execute ABC.
        This can e.g. be used to call a specific version of ABC or a wrapper.

    -script <file>
        use the specified ABC script file instead of the default script.

    -liberty <file>
        generate netlists for the specified cell library (using the liberty
        file format). Without this option, ABC is used to optimize the netlist
        but keeps using yosys's internal gate library. This option is ignored if
        the -script option is also used.

    -nocleanup
        when this option is used, the temporary files created by this pass
        are not removed. this is useful for debugging.

This pass does not operate on modules with unprocessed processes in it.
(I.e. the 'proc' pass should be used first to convert processes to netlists.)

[1] http://www.eecs.berkeley.edu/~alanmi/abc/
```

C.2 cd – a shortcut for 'select -module <name>'

```
    cd <modname>

This is just a shortcut for 'select -module <modname>'.

```

```
    cd <cellname>

When no module with the specified name is found, but there is a cell
with the specified name in the current module, then this is equivialent
to 'cd <celltype>'.

    cd ..

This is just a shortcut for 'select -clear'.
```

C.3 dfflibmap – technology mapping of flip-flops

```
    dfflibmap -liberty <file> [selection]

Map internal flip-flop cells to the flip-flop cells in the technology
library specified in the given liberty file.

This pass may add inverters as needed. Therefore it is recommended to
first run this pass and then map the logic paths to the target technology.
```

C.4 dump – print parts of the design in ilang format

```
    dump [options] [selection]

Write the selected parts of the design to the console or specified file in
ilang format.

    -outfile <filename>
        Write to the specified file.
```

C.5 eval – evaluate the circuit given an input

```
    eval [options] [selection]

This command evaluates the value of a signal given the value of all required
inputs.

    -set <signal> <value>
        set the specified signal to the specified value.

    -show <signal>
        show the value for the specified signal. if no -show option is passed
        then all output ports of the current module are used.
```

C.6 extract – find subcircuits and replace them with cells

```
    extract -map <map_file> [options] [selection]
    extract -mine <out_file> [options] [selection]

This pass looks for subcircuits that are isomorphic to any of the modules
in the given map file and replaces them with instances of this modules. The
map file can be a verilog source file (*.v) or an ilang file (*.il).

    -map <map_file>
        use the modules in this file as reference

    -verbose
        print debug output while analyzing

    -constports
        also find instances with constant drivers. this may be much
        slower than the normal operation.

    -nodefaultswaps
        normally builtin port swapping rules for internal cells are used per
        default. This turns that off, so e.g. 'a^b' does not match 'b^a'
        when this option is used.

    -compat <needle_type> <haystack_type>
        Per default, the cells in the map file (needle) must have the
        type as the cells in the active design (haystack). This option
        can be used to register additional pairs of types that should
        match. This option can be used multiple times.

    -swap <needle_type> <port1>,<port2>[,...]
        Register a set of swapable ports for a needle cell type.
        This option can be used multiple times.

    -perm <needle_type> <port1>,<port2>[,...] <portA>,<portB>[,...]
        Register a valid permutation of swapable ports for a needle
        cell type. This option can be used multiple times.

    -cell_attr <attribute_name>
        Attributes on cells with the given name must match.

    -wire_attr <attribute_name>
        Attributes on wires with the given name must match.

This pass does not operate on modules with uprocessed processes in it.
(I.e. the 'proc' pass should be used first to convert processes to netlists.)

This pass can also be used for mining for frequent subcircuits. In this mode
the following options are to be used instead of the -map option.

    -mine <out_file>
        mine for frequent subcircuits and write them to the given ilang file

    -mine_cells_span <min> <max>
```

```
        only mine for subcircuits with the specified number of cells
        default value: 3 5

    -mine_min_freq <num>
        only mine for subcircuits with at least the specified number of matches
        default value: 10

    -mine_limit_matches_per_module <num>
        when calculating the number of matches for a subcircuit, don't count
        more than the specified number of matches per module

    -mine_max_fanout <num>
        don't consider internal signals with more than <num> connections

The modules in the map file may have the attribute 'extract_order' set to an
integer value. Then this value is used to determine the order in which the pass
tries to map the modules to the design (ascending, default value is 0).

See 'help techmap' for a pass that does the opposite thing.
```

C.7 flatten – flatten design

```
    flatten [selection]

This pass flattens the design by replacing cells by their implementation. This
pass is very simmilar to the 'techmap' pass. The only difference is that this
pass is using the current design as mapping library.
```

C.8 fsm – extract and optimize finite state machines

```
    fsm [options] [selection]

This pass calls all the other fsm_* passes in a useful order. This performs
FSM extraction and optimiziation. It also calls opt_clean as needed:

    fsm_detect          unless got option -nodetect
    fsm_extract

    fsm_opt
    opt_clean
    fsm_opt

    fsm_expand          if got option -expand
    opt_clean           if got option -expand
    fsm_opt             if got option -expand

    fsm_recode          unless got option -norecode

    fsm_info

```

```
    fsm_export          if got option -export
    fsm_map             unless got option -nomap

Options:

    -expand, -norecode, -export, -nomap
        enable or disable passes as indicated above

    -encoding tye
    -fm_set_fsm_file file
        passed through to fsm_recode pass
```

C.9 fsm_detect – finding FSMs in design

```
    fsm_detect [selection]

This pass detects finite state machines by identifying the state signal.
The state signal is then marked by setting the attribute 'fsm_encoding'
on the state signal to "auto".

Existing 'fsm_encoding' attributes are not changed by this pass.

Signals can be protected from being detected by this pass by setting the
'fsm_encoding' attribute to "none".
```

C.10 fsm_expand – expand FSM cells by merging logic into it

```
    fsm_expand [selection]

The fsm_extract pass is conservative about the cells that belong to a finite
state machine. This pass can be used to merge additional auxiliary gates into
the finate state machine.
```

C.11 fsm_export – exporting FSMs to KISS2 files

```
    fsm_export [-noauto] [-o filename] [-origenc] [selection]

This pass creates a KISS2 file for every selected FSM. For FSMs with the
'fsm_export' attribute set, the attribute value is used as filename, otherwise
the module and cell name is used as filename. If the parameter '-o' is given,
the first exported FSM is written to the specified filename. This overwrites
the setting as specified with the 'fsm_export' attribute. All other FSMs are
exported to the default name as mentioned above.

    -noauto
        only export FSMs that have the 'fsm_export' attribute set

```

```
    -o filename
        filename of the first exported FSM

    -origenc
        use binary state encoding as state names instead of s0, s1, ...
```

C.12 fsm_extract – extracting FSMs in design

```
    fsm_extract [selection]

This pass operates on all signals marked as FSM state signals using the
'fsm_encoding' attribute. It consumes the logic that creates the state signal
and uses the state signal to generate control signal and replaces it with an
FSM cell.

The generated FSM cell still generates the original state signal with its
original encoding. The 'fsm_opt' pass can be used in combination with the
'opt_clean' pass to eliminate this signal.
```

C.13 fsm_info – print information on finite state machines

```
    fsm_info [selection]

This pass dumps all internal information on FSM cells. It can be useful for
analyzing the synthesis process and is called automatically by the 'fsm'
pass so that this information is included in the synthesis log file.
```

C.14 fsm_map – mapping FSMs to basic logic

```
    fsm_map [selection]

This pass translates FSM cells to flip-flops and logic.
```

C.15 fsm_opt – optimize finite state machines

```
    fsm_opt [selection]

This pass optimizes FSM cells. It detects which output signals are actually
not used and removes them from the FSM. This pass is usually used in
combination with the 'opt_clean' pass (see also 'help fsm').
```

C.16 fsm_recode – recoding finite state machines

```
    fsm_recode [-encoding type] [-fm_set_fsm_file file] [selection]

This pass reassign the state encodings for FSM cells. At the moment only
one-hot encoding and binary encoding is supported. The option -encoding
can be used to specify the encoding scheme used for FSMs without the
‘fsm_encoding’ attribute (or with the attribute set to ‘auto’.

The option -fm_set_fsm_file can be used to generate a file containing the
mapping from old to new FSM encoding in form of Synopsys Formality set_fsm_*
commands.
```

C.17 help – display help messages

```
    help  .............  list all commands
    help <command>  ...  print help message for given command
    help -all  ........  print complete command reference
```

C.18 hierarchy – check, expand and clean up design hierarchy

```
    hierarchy [-check] [-top <module>]
    hierarchy -generate <cell-types> <port-decls>

In parametric designs, a module might exists in serveral variations with
different parameter values. This pass looks at all modules in the current
design an re-runs the language frontends for the parametric modules as
needed.

    -check
        also check the design hierarchy. this generates an error when
        an unknown module is used as cell type.

    -top <module>
        use the specified top module to built a design hierarchy. modules
        outside this tree (unused modules) are removed.

In -generate mode this pass generates placeholder modules for the given cell
types (wildcards supported). For this the design is searched for cells that
match the given types and then the given port declarations are used to
determine the direction of the ports. The syntax for a port declaration is:

    {i|o|io}[@<num>]:<portname>

Input ports are specified with the 'i' prefix, output ports with the 'o'
prefix and inout ports with the 'io' prefix. The optional <num> specifies
the position of the port in the parameter list (needed when instanciated
using positional arguments). When <num> is not specified, the <portname> can
also contain wildcard characters.
```

```
29
30 This pass ignores the current selection and always operates on all modules
31 in the current design.
```

C.19 ls – list modules or objects in modules

```
1     ls
2
3  When no active module is selected, this prints a list of all module.
4
5  When an active module is selected, this prints a list of objects in the module.
```

C.20 memory – translate memories to basic cells

```
1     memory [-nomap] [selection]
2
3  This pass calls all the other memory_* passes in a useful order:
4
5     memory_dff
6     memory_collect
7     memory_map          (skipped if called with -nomap)
8
9  This converts memories to word-wide DFFs and address decoders
10 or moultiport memory blocks if called with the -nomap option.
```

C.21 memory_collect – creating multi-port memory cells

```
1     memory_collect [selection]
2
3  This pass collects memories and memory ports and creates generic multiport
4  memory cells.
```

C.22 memory_dff – merge input/output DFFs into memories

```
1     memory_dff [selection]
2
3  This pass detects DFFs at memory ports and merges them into the memory port.
4  I.e. it consumes an asynchronous memory port and the flip-flops at its
5  interface and yields a synchronous memory port.
```

C.23 memory_map – translate multiport memories to basic cells

```
    memory_map [selection]

This pass converts multiport memory cells as generated by the memory_collect
pass to word-wide DFFs and address decoders.
```

C.24 opt – perform simple optimizations

```
    opt [selection]

This pass calls all the other opt_* passes in a useful order. This performs
a series of trivial optimizations and cleanups. This pass executes the other
passes in the following order:

    opt_const
    opt_share -nomux

    do
        opt_muxtree
        opt_reduce
        opt_share
        opt_rmdff
        opt_clean
        opt_const
    while [changed design]
```

C.25 opt_clean – remove unused cells and wires

```
    opt_clean [options] [selection]

This pass identifies wires and cells that are unused and removes them. Other
passes often remove cells but leave the wires in the design or reconnect the
wires but leave the old cells in the design. This pass can be used to clean up
after the passes that do the actual work.

This pass only operates on completely selected modules without processes.

    -purge
        also remove internal nets if they have a public name
```

C.26 opt_const – perform const folding

```
    opt_const [selection]

This pass performs const folding on internal cell types with constant inputs.
```

C.27 opt_muxtree – eliminate dead trees in multiplexer trees

```
    opt_muxtree [selection]

This pass analyzes the control signals for the multiplexer trees in the design
and identifies inputs that can never be active. It then removes this dead
branches from the multiplexer trees.

This pass only operates on completely selected modules without processes.
```

C.28 opt_reduce – simplify large MUXes and AND/OR gates

```
    opt_reduce [selection]

This pass performs two interlinked optimizations:

1. it consolidates trees of large AND gates or OR gates and eliminates
duplicated inputs.

2. it identifies duplicated inputs to MUXes and replaces them with a single
input with the original control signals OR'ed together.
```

C.29 opt_rmdff – remove DFFs with constant inputs

```
    opt_rmdff [selection]

This pass identifies flip-flops with constant inputs and replaces them with
a constant driver.
```

C.30 opt_share – consolidate identical cells

```
    opt_share [-nomux] [selection]

This pass identifies cells with identical type and input signals. Such cells
are then merged to one cell.

    -nomux
        Do not merge MUX cells.
```

C.31 proc – translate processes to netlists

```
    proc [selection]

This pass calls all the other proc_* passes in the most common order.
```

```
4
5       proc_clean
6       proc_rmdead
7       proc_arst
8       proc_mux
9       proc_dff
10      proc_clean
11
12  This replaces the processes in the design with multiplexers and flip-flops.
```

C.32 proc_arst – detect asynchronous resets

```
1       proc_arst [selection]
2
3   This pass identifies asynchronous resets in the processes and converts them
4   to a different internal representation that is suitable for generating
5   flip-flop cells with asynchronous resets.
```

C.33 proc_clean – remove empty parts of processes

```
1       proc_clean [selection]
2
3   This pass removes empty parts of processes and ultimately removes a process
4   if it contains only empty structures.
```

C.34 proc_dff – extract flip-flops from processes

```
1       proc_dff [selection]
2
3   This pass identifies flip-flops in the processes and converts them to
4   d-type flip-flop cells.
```

C.35 proc_mux – convert decision trees to multiplexers

```
1       proc_mux [selection]
2
3   This pass converts the decision trees in processes (originating from if-else
4   and case statements) to trees of multiplexer cells.
```

C.36 proc_rmdead – eliminate dead trees in decision trees

```
1       proc_rmdead [selection]
2
3   This pass identifies unreachable branches in decision trees and removes them.
```

C.37 read_ilang – read modules from ilang file

```
    read_ilang [filename]

Load modules from an ilang file to the current design. (ilang is a text
representation of a design in yosys's internal format.)
```

C.38 read_verilog – read modules from verilog file

```
    read_verilog [filename]

Load modules from a verilog file to the current design. A large subset of
Verilog-2005 is supported.

    -dump_ast
        dump abstract syntax tree (after simplification)

    -dump_ast_diff
        dump ast differences before and after simplification

    -dump_vlog
        dump ast as verilog code (after simplification)

    -yydebug
        enable parser debug output

    -nolatches
        usually latches are synthesized into logic loops
        this option prohibits this and sets the output to 'x'
        in what would be the latches hold condition

        this behavior can also be achieved by setting the
        'nolatches' attribute on the respective module or
        always block.

    -nomem2reg
        under certain conditions memories are converted to registers
        early during simplification to ensure correct handling of
        complex corner cases. this option disables this behavior.

        this can also be achieved by setting the 'nomem2reg'
        attribute on the respective module or register.

    -mem2reg
        always convert memories to registers. this can also be
        achieved by setting the 'mem2reg' attribute on the respective
        module or register.

    -ppdump
        dump verilog code after pre-processor

    -nopp
```

```
        do not run the pre-processor

    -lib
        only create empty placeholder modules

    -noopt
        don't perform basic optimizations (such as const folding) in the
        high-level front-end.

    -Dname[=definition]
        define the preprocessor symbol 'name' and set its optional value
        'definition'
```

C.39 rename – rename object in the design

```
    rename old_name new_name

Rename the specified object. Note that selection patterns are not supported
by this command.
```

C.40 sat – solve a SAT problem in the circuit

```
    sat [options] [selection]

This command solves a SAT problem defined over the currently selected circuit
and additional constraints passed as parameters.

    -all
        show all solutions to the problem (this can grow exponentially, use
        -max <N> instead to get <N> solutions)

    -max <N>
        like -all, but limit number of solutions to <N>

    -set <signal> <value>
        set the specified signal to the specified value.

    -show <signal>
        show the model for the specified signal. if no -show option is
        passed then a set of signals to be shown is automatically selected.

The following options can be used to set up a sequential problem:

    -seq <N>
        set up a sequential problem with <N> time steps. The steps will
        be numbered from 1 to N.

    -set-at <N> <signal> <value>
    -unset-at <N> <signal>
        set or unset the specified signal to the specified value in the
```

```
        given timestep. this has priority over a -set for the same signal.

The following additional options can be used to set up a proof. If also -seq
is passed, a temporal induction proof is performed.

    -prove <signal> <value>
        Attempt to proof that <signal> is always <value>. In a temporal
        induction proof it is proven that the condition holds forever after
        the number of time steps passed using -seq.

    -maxsteps <N>
        Set a maximum length for the induction.

    -timeout <N>
        Maximum number of seconds a single SAT instance may take.

    -verify
        Return an error and stop the synthesis script if the proof fails.

    -verify-no-timeout
        Like -verify but do not return an error for timeouts.
```

C.41 scatter – add additional intermediate nets

```
    scatter [selection]

This command adds additional intermediate nets on all cell ports. This is used
for testing the correct use of the SigMap halper in passes. If you don't know
what this means: don't worry -- you only need this pass when testing your own
extensions to Yosys.

Use the opt_clean command to get rid of the additional nets.
```

C.42 scc – detect strongly connected components (logic loops)

```
    scc [options] [selection]

This command identifies strongly connected components (aka logic loops) in the
design.

    -max_depth <num>
        limit to loops not longer than the specified number of cells. This can
        e.g. be useful in identifying local loops in a module that turns out
        to be one gigantic SCC.

    -all_cell_types
        Usually this command only considers internal non-memory cells. With
        this option set, all cells are considered. For unkown cells all ports
        are assumed to be bidirectional 'inout' ports.

```

```
    -set_attr <name> <value>
    -set_cell_attr <name> <value>
    -set_wire_attr <name> <value>
        set the specified attribute on all cells and/or wires that are part of
        a logic loop. the special token {} in the value is replaced with a
        unique identifier for the logic loop.

    -select
        replace the current selection with a selection of all cells and wires
        that are part of a found logic loop
```

C.43 script – execute commands from script file

```
    script <filename>

This command executes the yosys commands in the specified file.
```

C.44 select – modify and view the list of selected objects

```
    select [ -add | -del | -set <name> ] <selection>
    select [ -list | -write <filename> | -count | -clear ]
    select -module <modname>

Most commands use the list of currently selected objects to determine which part
of the design to operate on. This command can be used to modify and view this
list of selected objects.

Note that many commands support an optional [selection] argument that can be
used to override the global selection for the command. The syntax of this
optional argument is identical to the syntax of the <selection> argument
described here.

    -add, -del
        add or remove the given objects to the current selection.
        without this options the current selection is replaced.

    -set <name>
        do not modify the current selection. instead save the new selection
        under the given name (see @<name> below).

    -list
        list all objects in the current selection

    -write <filename>
        like -list but write the output to the specified file

    -count
        count all objects in the current selection

    -clear
```

```
        clear the current selection. this effectively selects the
        whole design.

    -module <modname>
        limit the current scope to the specified module.
        the difference between this and simply selecting the module
        is that all object names are interpreted relative to this
        module after this command until the selection is cleared again.

When this command is called without an argument, the current selection
is displayed in a compact form (i.e. only the module name when a whole module
is selected).

The <selection> argument itself is a series of commands for a simple stack
machine. Each element on the stack represents a set of selected objects.
After this commands have been executed, the union of all remaining sets
on the stack is computed and used as selection for the command.

Pushing (selecting) object when not in -module mode:

    <mod_pattern>
        select the specified module(s)

    <mod_pattern>/<obj_pattern>
        select the specified object(s) from the module(s)

Pushing (selecting) object when in -module mode:

    <obj_pattern>
        select the specified object(s) from the current module

A <mod_pattern> can be a module name or wildcard expression (*, ?, [..])
matching module names.

An <obj_pattern> can be an object name, wildcard expression, or one of
the following:

    w:<pattern>
        all wires with a name matching the given wildcard pattern

    m:<pattern>
        all memories with a name matching the given pattern

    c:<pattern>
        all cells with a name matching the given pattern

    t:<pattern>
        all cells with a type matching the given pattern

    p:<pattern>
        all processes with a name matching the given pattern

    a:<pattern>
        all objects with an attribute name matching the given pattern
```

```
86
87      a:<pattern>=<pattern>
88          all objects with a matching attribute name-value-pair
89
90      n:<pattern>
91          all objects with a name matching the given pattern
92          (i.e. 'n:' is optional as it is the default matching rule)
93
94      @<name>
95          push the selection saved prior with 'select -set <name> ...'
96
97  The following actions can be performed on the top sets on the stack:
98
99      %
100         push a copy of the current selection to the stack
101
102     %%
103         replace the stack with a union of all elements on it
104
105     %n
106         replace top set with its invert
107
108     %u
109         replace the two top sets on the stack with their union
110
111     %i
112         replace the two top sets on the stack with their intersection
113
114     %d
115         pop the top set from the stack and subtract it from the new top
116
117     %x[<num1>|*][.<num2>][:<rule>[:<rule>..]]
118         expand top set <num1> num times according to the specified rules.
119         (i.e. select all cells connected to selected wires and select all
120         wires connected to selected cells) The rules specify which cell
121         ports to use for this. the syntax for a rule is a '-' for exclusion
122         and a '+' for inclusion, followed by an optional comma seperated
123         list of cell types followed by an optional comma separated list of
124         cell ports in square brackets. a rule can also be just a cell or wire
125         name that limits the expansion (is included but does not go beyond).
126         select at most <num2> objects. a warning message is printed when this
127         limit is reached. When '*' is used instead of <num1> then the process
128         is repeated until no further object are selected.
129
130     %ci[<num1>|*][.<num2>][:<rule>[:<rule>..]]
131     %co[<num1>|*][.<num2>][:<rule>[:<rule>..]]
132         simmilar to %x, but only select input (%ci) or output cones (%co)
133
134 Example: the following command selects all wires that are connected to a
135 'GATE' input of a 'SWITCH' cell:
136
137     select */t:SWITCH %x:+[GATE] */t:SWITCH %d
```

C.45 shell – enter interactive command mode

```
    shell

This command enters the interactive command mode. This can be useful
in a script to interrupt the script at a certain point and allow for
interactive inspection or manual synthesis of the design at this point.

The command prompt of the interactive shell indicates the current
selection (see 'help select'):

    yosys>
        the entire design is selected

    yosys*>
        only part of the design is selected

    yosys [modname]>
        the entire module 'modname' is selected using 'select -module modname'

    yosys [modname]*>
        only part of current module 'modname' is selected

When in interactive shell, some errors (e.g. invalid command arguments)
do not terminate yosys but return to the command prompt.

This command is the default action if nothing else has been specified
on the command line.

Press Ctrl-D or type 'exit' to leave the interactive shell.
```

C.46 show – generate schematics using graphviz

```
    show [options] [selection]

Create a graphviz DOT file for the selected part of the design and compile it
to a graphics file (usually SVG or PostScript).

    -viewer <viewer>
        Run the specified command with the graphics file as parameter.

    -format <format>
        Generate a graphics file in the specified format.
        Usually <format> is 'svg' or 'ps'.

    -lib <verilog_or_ilang_file>
        Use the specified library file for determining whether cell ports are
        inputs or outputs. This option can be used multiple times to specify
        more than one library.

    -prefix <prefix>
        generate <prefix>.dot and <prefix>.ps instead of ~/.yosys_show.{dot,ps}
```

```
20
21      -color <color> <wire>
22          assign the specified color to the specified wire. The object can be
23          a single selection wildcard expressions or a saved set of objects in
24          the @<name> syntax (see "help select" for details).
25
26      -colors <seed>
27          Randomly assign colors to the wires. The integer argument is the seed
28          for the random number generator. Change the seed value if the colored
29          graph still is ambigous. A seed of zero deactivates the coloring.
30
31      -width
32          annotate busses with a label indicating the width of the bus.
33
34      -stretch
35          stretch the graph so all inputs are on the left side and all outputs
36          (including inout ports) are on the right side.
37
38  When no <format> is specified, SVG is used. When no <format> and <viewer> is
39  specified, 'yosys-svgviewer' is used to display the schematic.
40
41  The generated output files are '~/.yosys_show.dot' and '~/.yosys_show.<format>',
42  unless another prefix is specified using -prefix <prefix>.
```

C.47 splitnets – split up multi-bit nets

```
1       splitnets [options] [selection]
2
3   This command splits multi-bit nets into single-bit nets.
4
5       -ports
6           also split module ports. per default only internal signals are split.
```

C.48 submod – moving part of a module to a new submodule

```
1       submod [selection]
2
3   This pass identifies all cells with the 'submod' attribute and moves them to
4   a newly created module. The value of the attribute is used as name for the
5   cell that replaces the group of cells with the same attribute value.
6
7   This pass can be used to create a design hierarchy in flat design. This can
8   be useful for analyzing or reverse-engineering a design.
9
10  This pass only operates on completely selected modules with no processes
11  or memories.
12
13
14      submod -name <name> [selection]
15
```

```
As above, but don't use the 'submod' attribute but instead use the selection.
Only objects from one module might be selected. The value of the -name option
is used as the value of the 'submod' attribute above.
```

C.49 tcl – execute a TCL script file

```
    tcl <filename>

This command executes the tcl commands in the specified file.
Use 'yosys cmd' to run the yosys command 'cmd' from tcl.

The tcl command 'yosys -import' can be used to import all yosys
commands directly as tcl commands to the tcl shell. The yosys
command 'proc' is wrapped using the tcl command 'procs' in order
to avoid a name collision with the tcl builting command 'proc'.
```

C.50 techmap – simple technology mapper

```
    techmap [-map filename] [selection]

This pass implements a very simple technology mapper that replaces cells in
the design with implementations given in form of a verilog or ilang source
file.

    -map filename
        the library of cell implementations to be used.
        without this parameter a builtin library is used that
        transforms the internal RTL cells to the internal gate
        library.

When a module in the map file has the 'celltype' attribute set, it will match
cells with a type that match the text value of this attribute.

When a module in the map file contains a wire with the name 'TECHMAP_FAIL' (or
one matching '*.TECHMAP_FAIL') then no substitution will be performed. The
modules in the map file are tried in alphabetical order.

When a module in the map file has a parameter where the according cell in the
design has a port, the module from the map file is only used if the port in
the design is connected to a constant value. The parameter is then set to the
constant value.

See 'help extract' for a pass that does the opposite thing.

See 'help flatten' for a pass that does flatten the design (which is
esentially techmap but using the design itself as map library).
```

C.51 write_autotest – generate simple test benches

```
    write_autotest [filename]

Automatically create primitive verilog test benches for all modules in the
design. The generated testbenches toggle the input pins of the module in
a semi-random manner and dumps the resulting output signals.

This can be used to check the synthesis results for simple circuits by
comparing the testbench output for the input files and the synthesis results.

The backend automatically detects clock signals. Additionally a signal can
be forced to be interpreted as clock signal by setting the attribute
'gentb_clock' on the signal.

The attribute 'gentb_constant' can be used to force a signal to a constant
value after initialization. This can e.g. be used to force a reset signal
low in order to explore more inner states in a state machine.
```

C.52 write_ilang – write design to ilang file

```
    write_ilang [filename]

Write the current design to an 'ilang' file. (ilang is a text representation
of a design in yosys's internal format.)
```

C.53 write_intersynth – write design to InterSynth netlist file

```
    write_intersynth [options] [filename]

Write the current design to an 'intersynth' netlist file. InterSynth is
a tool for Coarse-Grain Example-Driven Interconnect Synthesis.

    -notypes
        do not generate celltypes and conntypes commands. i.e. just output
        the netlists. this is used for postsilicon synthesis.

    -lib <verilog_or_ilang_file>
        Use the specified library file for determining whether cell ports are
        inputs or outputs. This option can be used multiple times to specify
        more than one library.

http://www.clifford.at/intersynth/
```

C.54 write_verilog – write design to verilog file

```
    write_verilog [options] [filename]

Write the current design to a verilog file.

    -norename
        without this option all internal object names (the ones with a dollar
        instead of a backslash prefix) are changed to short names in the
        format '_<number>_'.

    -noattr
        with this option no attributes are included in the output

    -attr2comment
        with this option attributes are included as comments in the output

    -noexpr
        without this option all internal cells are converted to verilog
        expressions.

    -placeholders
        usually modules with the 'placeholder' attribute are ignored. with
        this option set only the modules with the 'placeholder' attribute
        are written to the output file.
```

Appendix D

Application Notes

FIXME:
This appendix will cover some typical use-cases of Yosys in the form of application notes.

D.1 Synthesizing using a Cell Library in Liberty Format

D.2 Reverse Engeneering the MOS6502 from an NMOS Transistor Netlist

D.3 Reconfigurable Coarse-Grain Synthesis using Intersynth

Appendix E

Evaluation of other OSS Verilog Synthesis Tools

In this appendix[1] the existing FOSS Verilog synthesis tools[2] are evaluated. Extremely limited or application specific tools (e.g. pure Verilog Netlist parsers) as well as Verilog simulators are not included. These existing solutions are tested using a set of representative Verilog code snippets. It is shown that no existing FOSS tool implements even close to a sufficient subset of Verilog to be usable as synthesis tool for a wide range existing Verilog code.

The packages evaluated are:

- Icarus Verilog [41][3]
- Verilog-to-Routing (VTR) / Odin-II [RLY+12][JR05][39]
- HDL Analyzer and Netlist Architect (HANA) [32]
- Verilog front-end to VIS (vl2mv) [CYB93][40]

In each of the following sections Verilog modules that test a certain Verilog language feature are presented and the support for these features is tested in all the tools mentioned above. It is evaluated whether the tools under test successfully generate netlists for the Verilog input and whether these netlists match the simulation behavior of the designs using testbenches.

All test cases are verified to be synthesizeable using Xilinx XST from the Xilinx WebPACK [44] suite.

Trivial features such as support for simple structural Verilog are not explicitly tested.

Vl2mv and Odin-II generate output in the BLIF (Berkeley Logic Interchange Format) and BLIF-MV (an extended version of BLIF) formats respectively. ABC [35] is used to convert this output to Verilog for verification using testbenches.

Icarus Verilog generates EDIF (Electronic Design Interchange Format) output utilizing LPM (Library of Parameterized Modules) cells. The EDIF files are converted to Verilog using edif2ngd and netgen from Xilinx WebPACK. A hand-written implementation of the LPM cells utilized by the generated netlists is used for verification.

Following these functional tests, a quick analysis of the extensibility of the tools under test is provided in a separate section.

The last section of this chapter finally concludes these series of evaluations with a summary of the results.

[1] This appendix is an updated version of an unpublished student research paper. [Wol12]

[2] To the author's best knowledge, all relevant tools that existed at the time of this writing are included. But as there is no formal channel through which such tools are published it is hard to give any guarantees in that matter.

[3] Icarus Verilog is mainly a simulation tool but also supported synthesis up to version 0.8. Therefore version 0.8.7 is used

```
1  module uut_always01(clock,
2                  reset, count);
3
4  input clock, reset;
5  output [3:0] count;
6  reg [3:0] count;
7
8  always @(posedge clock)
9          count <= reset ?
10                 0 : count + 1;
11
12
13
14 endmodule
```

```
module uut_always02(clock,
                reset, count);

input clock, reset;
output [3:0] count;
reg [3:0] count;

always @(posedge clock) begin
        count <= count + 1;
        if (reset)
                count <= 0;
end

endmodule
```

Figure E.1: 1st and 2nd Verilog always examples

```
1  module uut_always03(clock, in1, in2, in3, in4, in5, in6, in7,
2                  out1, out2, out3);
3
4  input clock, in1, in2, in3, in4, in5, in6, in7;
5  output out1, out2, out3;
6  reg out1, out2, out3;
7
8  always @(posedge clock) begin
9          out1 = in1;
10         if (in2)
11                 out1 = !out1;
12         out2 <= out1;
13         if (in3)
14                 out2 <= out2;
15         if (in4)
16                 if (in5)
17                         out3 <= in6;
18                 else
19                         out3 <= in7;
20         out1 = out1 ^ out2;
21 end
22
23 endmodule
```

Figure E.2: 3rd Verilog always example

E.1 Always blocks and blocking vs. nonblocking assignments

The "always"-block is one of the most fundamental non-trivial Verilog language features. It can be used to model a combinatorial path (with optional registers on the outputs) in a way that mimics a regular programming language.

Within an always block, if- and case-statements can be used to model multiplexers. Blocking assignments (=) and nonblocking assignments (<=) are used to populate the leaf-nodes of these multiplexer trees.

for this evaluation.)

Unassigned leaf-nodes default to feedback paths that cause the output register to hold the previous value. More advanced synthesis tools often convert these feedback paths to register enable signals or even generate circuits with clock gating.

Registers assigned with nonblocking assignments (<=) behave differently from variables in regular programming languages: In a simulation they are not updated immediately after being assigned. Instead the right-hand sides are evaluated and the results stored in temporary memory locations. After all pending updates have been prepared in this way they are executed, thus yielding semi-parallel execution of all nonblocking assignments.

For synthesis this means that every occurrence of that register in an expression addresses the output port of the corresponding register regardless of the question whether the register has been assigned a new value in an earlier command in the same always block. Therefore with nonblocking assignments the order of the assignments has no effect on the resulting circuit as long as the left-hand sides of the assignments are unique.

The three example codes in Fig. E.1 and Fig. E.2 use all these features and can thus be used to test the synthesis tools capabilities to synthesize always blocks correctly.

The first example is only using the most fundamental Verilog features. All tools under test were able to successfully synthesize this design.

The 2nd example is functionally identical to the 1st one but is using an if-statement inside the always block. Odin-II fails to synthesize it and instead produces the following error message:

```
ERROR: (File: always02.v) (Line number: 13)
You've defined the driver "count~0" twice
```

Vl2mv does not produce an error message but outputs an invalid synthesis result that is not using the reset input at all.

Icarus Verilog also doesn't produce an error message but generates an invalid output for this 2nd example. The code generated by Icarus Verilog only implements the reset path for the count register, effectively setting the output to constant 0.

So of all tools under test only HANA was able to create correct synthesis results for the 2nd example.

The 3rd example is using blocking and nonblocking assignments and many if statements. Odin also fails to synthesize this example:

```
1  module uut_arrays01(clock, we, addr, wr_data, rd_data);
2
3  input clock, we;
4  input [3:0] addr, wr_data;
5  output [3:0] rd_data;
6  reg [3:0] rd_data;
7
8  reg [3:0] memory [15:0];
9
10 always @(posedge clock) begin
11         if (we)
12                 memory[addr] <= wr_data;
13         rd_data <= memory[addr];
14 end
15
16 endmodule
```

Figure E.3: Verilog array example

```
1  module uut_forgen01(a, y);
2
3  input [4:0] a;
4  output y;
5
6  integer i, j;
7  reg [31:0] lut;
8
9  initial begin
10         for (i = 0; i < 32; i = i+1) begin
11                 lut[i] = i > 1;
12                 for (j = 2; j*j <= i; j = j+1)
13                         if (i % j == 0)
14                                 lut[i] = 0;
15         end
16 end
17
18 assign y = lut[a];
19
20 endmodule
```

Figure E.4: Verilog for loop example

```
ERROR: (File: always03.v) (Line number: 8)
ODIN doesn't handle blocking statements in Sequential blocks
```

HANA, Icarus Verilog and vl2mv create invalid synthesis results for the 3rd example.

So unfortunately none of the tools under test provide a complete and correct implementation of blocking and nonblocking assignments.

E.2 Arrays for memory modelling

Verilog arrays are part of the synthesizeable subset of Verilog and are commonly used to model addressable memory. The Verilog code in Fig. E.3 demonstrates this by implementing a single port memory.

For this design HANA, vl2m and ODIN-II generate error messages indicating that arrays are not supported.

Icarus Verilog produces an invalid output that is using the address only for reads. Instead of using the address input for writes, the generated design simply loads the data to all memory locations whenever the write-enable input is active, effectively turning the design into a single 4-bit D-Flip-Flop with enable input.

As all tools under test already fail this simple test, there is nothing to gain by continuing tests on this aspect of Verilog synthesis such as synthesis of dual port memories, correct handling of write collisions, and so forth.

E.3 For-loops and generate blocks

For-loops and generate blocks are more advanced Verilog features. These features allow the circuit designer to add program code to her design that is evaluated during synthesis to generate (parts of) the circuits description; something that could only be done using a code generator otherwise.

```
1  module uut_forgen02(a, b, cin, y, cout);
2
3  parameter WIDTH = 8;
4
5  input [WIDTH-1:0] a, b;
6  input cin;
7
8  output [WIDTH-1:0] y;
9  output cout;
10
11 genvar i;
12 wire [WIDTH-1:0] carry;
13
14 generate
15         for (i = 0; i < WIDTH; i=i+1) begin:adder
16                 wire [2:0] D;
17                 assign D[1:0] = { a[i], b[i] };
18                 if (i == 0) begin:chain
19                         assign D[2] = cin;
20                 end else begin:chain
21                         assign D[2] = carry[i-1];
22                 end
23                 assign y[i] = ^D;
24                 assign carry[i] = &D[1:0] | (^D[1:0] & D[2]);
25         end
26 endgenerate
27
28 assign cout = carry[WIDTH-1];
29
30 endmodule
```

Figure E.5: Verilog generate example

For-loops are only allowed in synthesizeable Verilog if they can be completely unrolled. Then they can be a powerful tool to generate array logic or static lookup tables. The code in Fig. E.4 generates a circuit that tests a 5 bit value for being a prime number using a static lookup table.

Generate blocks can be used to model array logic in complex parametric designs. The code in Fig. E.5 implements a ripple-carry adder with parametric width from simple assign-statements and logic operations using a Verilog generate block.

All tools under test failed to synthesize both test cases. HANA creates invalid output in both cases. Icarus Verilog creates invalid output for the first test and fails with an error for the second case. The other two tools fail with error messages for both tests.

E.4 Extensibility

This section briefly discusses the extensibility of the tools under test and their internal data- and control-flow. As all tools under test already failed to synthesize simple Verilog always-blocks correctly, not much resources have been spent on evaluating the extensibility of these tools and therefore only a very brief discussion of the topic is provided here.

HANA synthesizes for a built-in library of standard cells using two passes over an AST representation of the Verilog input. This approach executes fast but limits the extensibility as everything happens in only two comparable complex AST walks and there is no universal intermediate representation that is flexible enough to be used in arbitrary optimizations.

Odin-II and vl2m are both front ends to existing synthesis flows. As such they only try to quickly convert the Verilog input into the internal representation of their respective flows (BLIF). So extensibility is less of an issue here as potential extensions would likely be implemented in other components of the flow.

Icarus Verilog is clearly designed to be a simulation tool rather than a synthesis tool. The synthesis part of Icarus Verilog is an ad-hoc add-on to Icarus Verilog that aims at converting an internal representation that is meant for generation of a virtual machine based simulation code to netlists.

E.5 Summary and Outlook

Table E.1 summarizes the tests performed. Clearly none of the tools under test make a serious attempt at providing a feature-complete implementation of Verilog. It can be argued that Odin-II performed best in the test as it never generated incorrect code but instead produced error messages indicating that unsupported Verilog features where used in the Verilog input.

In conclusion, to the best knowledge of the author, there is no FOSS Verilog synthesis tool other than Yosys that is anywhere near feature completeness and therefore there is no other candidate for a generic Verilog front end and/or synthesis framework to be used as a basis for custom synthesis tools.

Yosys could also replace vl2m and/or Odin-II in their respective flows or function as a pre-compiler that can translate full-featured Verilog code to the simple subset of Verilog that is understood by vl2m and Odin-II.

Yosys is designed for extensibility. It can be used as-is to synthesize Verilog code to netlists, but its main purpose is to be used as basis for custom tools. Yosys is structured in a language dependent Verilog front end and language independent synthesis code (which is in itself structured in independent passes). This architecture will simplify implementing additional HDL front ends and/or additional synthesis passes.

Chapter 10 contains a more detailed evaluation of Yosys using real-world designes that are far out of reach for any of the other tools discussed in this appendix.

	HANA	VIS / vl2m	Icarus Verilog	Odin-II	Yosys
always01	✔	✔	✔	✔	✔
always02	✔	☠	☠	✘	✔
always03	☠	☠	☠	✘	✔
arrays01	✘	✘	☠	✘	✔
forgen01	☠	✘	☠	✘	✔
forgen02	☠	✘	✘	✘	✔

✔ . . . passed ✘ . . . produced error ☠ . . . incorrect output

Table E.1: Summary of all test results

Bibliography

[ASU86] Aho, Alfred V. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: principles, techniques, and tools*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1986. – ISBN 0–201–10088–6

[BHSV90] Brayton, R.K. ; Hachtel, G.D. ; Sangiovanni-Vincentelli, A.L.: Multilevel logic synthesis. In: *Proceedings of the IEEE* 78 (1990), Nr. 2, S. 264–300. http://dx.doi.org/10.1109/5.52213. – DOI 10.1109/5.52213. – ISSN 0018–9219

[CI00] Cummings, Clifford E. ; Inc, Sunburst D.: Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill. In: *SNUG (Synopsys Users Group) 2000 User Papers, section-MC1 (1 st paper*, 2000

[CYB93] Cheng, S-T ; York, G ; Brayton, R K.: *VL2MV: A Compiler from Verilog to BLIF-MV*. 1993

[GW13] Glaser, Johann ; Wolf, Clifford: Methodology and Example-Driven Interconnect Synthesis for Designing Heterogeneous Coarse-Grain Reconfigurable Architectures. In: Haase, Jan (Hrsg.): *Advances in Models, Methods, and Tools for Complex Chip Design — Selected contributions from FDL'12*. Springer, 2013. – to appear

[HS96] Hachtel, G D. ; Somenzi, F: *Logic Synthesis and Verification Algorithms*. 1996

[IP-10] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows. In: *IEEE Std 1685-2009* (2010), S. C1–360. http://dx.doi.org/10.1109/IEEESTD.2010.5417309. – DOI 10.1109/IEEESTD.2010.5417309

[JR05] Jamieson, Peter ; Rose, Jonathan: *A VERILOG RTL SYNTHESIS TOOL FOR HETEROGENEOUS FPGAS*. 2005

[Kli67] Klipstein, D. L.: The Contributions of Edsel Murphy to the Understanding of the Behavior of Inanimate Objects. In: *Cahners Publishing Co., EEE Magazine, Vol. 15, No. 8* (August 1967)

[LHBB85] Lee, Kyu Y. ; Holley, Michael ; Bailey, Mary ; Bright, Walter: A High-Level Design Language for Programmable Logic Devices. In: *VLSI Design (Manhasset NY: CPM Publications)* (June 1985), S. 50–62

[RLY+12] Rose, Jonathan ; Luu, Jason ; Yu, Chi W. ; Densmore, Opal ; Goeders, Jeff ; Somerville, Andrew ; Kent, Kenneth B. ; Jamieson, Peter ; Anderson, Jason: The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In: *Proceedings of the 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* ACM, 2012, S. 77–86

[STGR10] Shi, Yiqiong ; Ting, Chan W. ; Gwee, Bah-Hwee ; Ren, Ye: A highly efficient method for extracting FSMs from flattened gate-level netlist. In: *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010, S. 2610–2613

[Ull76] Ullmann, J. R.: An Algorithm for Subgraph Isomorphism. In: *J. ACM* 23 (1976), Januar, Nr. 1, S. 31–42. http://dx.doi.org/10.1145/321921.321925. – DOI 10.1145/321921.321925. – ISSN 0004–5411

[Ver02] IEEE Standard for Verilog Register Transfer Level Synthesis. In: *IEEE Std 1364.1-2002* (2002). http://dx.doi.org/10.1109/IEEESTD.2002.94220. – DOI 10.1109/IEEESTD.2002.94220

[Ver06] IEEE Standard for Verilog Hardware Description Language. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006). http://dx.doi.org/10.1109/IEEESTD.2006.99495. – DOI 10.1109/IEEESTD.2006.99495

[VHD04] IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. In: *IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999)* (2004). http://dx.doi.org/10.1109/IEEESTD.2004.94802. – DOI 10.1109/IEEESTD.2004.94802

[VHD09] IEEE Standard VHDL Language Reference Manual. In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), 26. http://dx.doi.org/10.1109/IEEESTD.2009.4772740. – DOI 10.1109/IEEESTD.2009.4772740

[WGS+12] Wolf, Clifford ; Glaser, Johann ; Schupfer, Florian ; Haase, Jan ; Grimm, Christoph: Example-driven interconnect synthesis for heterogeneous coarse-grain reconfigurable logic. In: *FDL Proceeding of the 2012 Forum on Specification and Design Languages*, 2012, S. 194–201

[Wol12] Wolf, Clifford: *Evaluation of Open Source Verilog Synthesis Tools for Feature-Completeness and Extensibility.* 2012. – Unpublished Student Research Paper, Vienna University of Technology

[Wol13] Wolf, Clifford: *Design and Implementation of the Yosys Open SYnthesis Suite.* 2013. – Bachelor Thesis, Vienna University of Technology

Internet References

[21] C-to-Verilog. http://www.c-to-verilog.com/.

[22] Flex. http://flex.sourceforge.net/.

[23] GNU Bison. http://www.gnu.org/software/bison/.

[24] LegUp. http://legup.eecg.utoronto.ca/.

[25] OpenCores I^2C Core. http://opencores.org/project,i2c.

[26] OpenCores k68 Core. http://opencores.org/project,k68.

[27] openMSP430 CPU. http://opencores.org/project,openmsp430.

[28] OpenRISC 1200 CPU. http://opencores.org/or1k/OR1200_OpenRISC_Processor.

[29] Synopsys Formality Equivalence Checking. http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/Formality.aspx.

[30] The Liberty Library Modeling Standard. http://www.opensourceliberty.org/.

[31] World of ASIC. http://www.asic-world.com/.

[32] P. Ahmad. HDL Analyzer and Netlist Architect (HANA). Verison linux64-1.0-alpha (2012-10-14), http://sourceforge.net/projects/sim-sim/.

[33] Altera, Inc. Quartus II Web Edition Software. http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html.

[34] Armin Biere, Johannes Kepler University Linz, Austria. AIGER. http://fmv.jku.at/aiger/.

[35] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. HQ Rev b5750272659f, 2012-10-28, http://www.eecs.berkeley.edu/~alanmi/abc/.

[36] M. group at Berkeley studies logic synthesis and verification for VLSI design. MVSIS: Logic Synthesis and Verification. Version 3.0, http://embedded.eecs.berkeley.edu/mvsis/.

[37] M. Hildenborg. smallsha1. https://code.google.com/p/smallsha1/.

[38] M. McCutchen. C++ Big Integer Library. http://mattmccutchen.net/bigint/.

[39] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson. The Verilog-to-Routing (VTR) Project for FPGAs. Version 1.0, https://code.google.com/p/vtr-verilog-to-routing/.

[40] The VIS group. VIS: A system for Verification and Synthesis. Version 2.4, http://vlsi.colorado.edu/~vis/.

[41] S. Williams. Icarus Verilog. Version 0.8.7, http://iverilog.icarus.com/.

[42] C. Wolf. VlogHammer Verilog Synthesis Regression Tests. http://github.com/cliffordwolf/VlogHammer.

[43] C. Wolf. Yosys Test Bench. http://github.com/cliffordwolf/yosys-tests.

[44] Xilinx, Inc. ISE WebPACK Design Software. http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.htm.