Yosys Manual

Claire Xenia Wolf

Abstract

Most of today's 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 necessary to write a custom HDL synthesis tool or add new features to an existing one. In these 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 developed. 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 successfully verified using formal verification and are comparable 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 Decision 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

CONTENTS

CONTENTS

CONTENTS

8

CONTENTS

9

CONTENTS

CONTENTS

CONTENTS

CONTENTS

13

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 author's preference for Verilog over VHDL it was decided early on to go for Verilog instead of VHDL[2]. So the existing FOSS Verilog synthesis tools were evaluated (see App. **??**). 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 as follows:

Chapter 1 is this introduction.

Chapter 2 covers a short introduction to the world of HDL synthesis. Basic principles and the terminology are 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 important pieces of the synthesis pipeline: The Verilog frontend, the optimization passes and the technology mapping to the target architecture, respectively.

Chapter **??** 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. **??**) 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 recent years formal equivalence checking also became an important verification method for validating RTL and lower abstraction representation of the design.
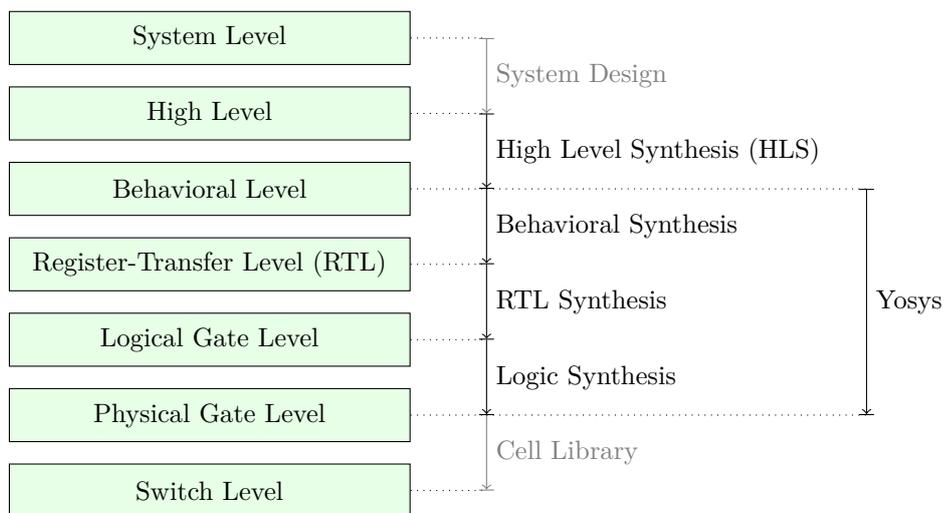
Figure 2.1: Different levels of abstraction and synthesis.

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. At 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 [16] [19].

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. **??**).

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 combinatorial 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

At 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 cannot 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 [27] [26] [28].

Yosys contains basic logic synthesis functionality but can also use ABC [27] 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. **??** 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. **??** and Fig. **??** in App. **??**.

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 (unlike 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-grained level (with complex cells such as adders or multipliers) and others work (best) on a fine-grained 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.
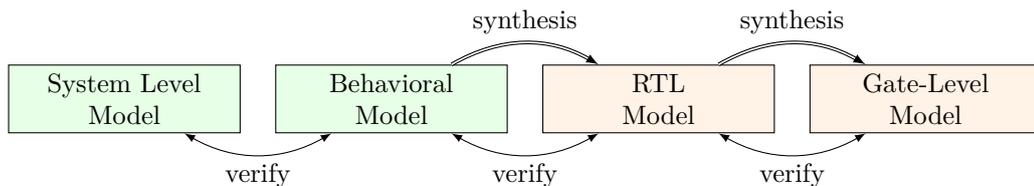
Figure 2.2: Typical design flow. Green boxes represent manually created models. Orange boxes represent models generated by synthesis tools.

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` [17]) 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 whitespace 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` [18]) 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]
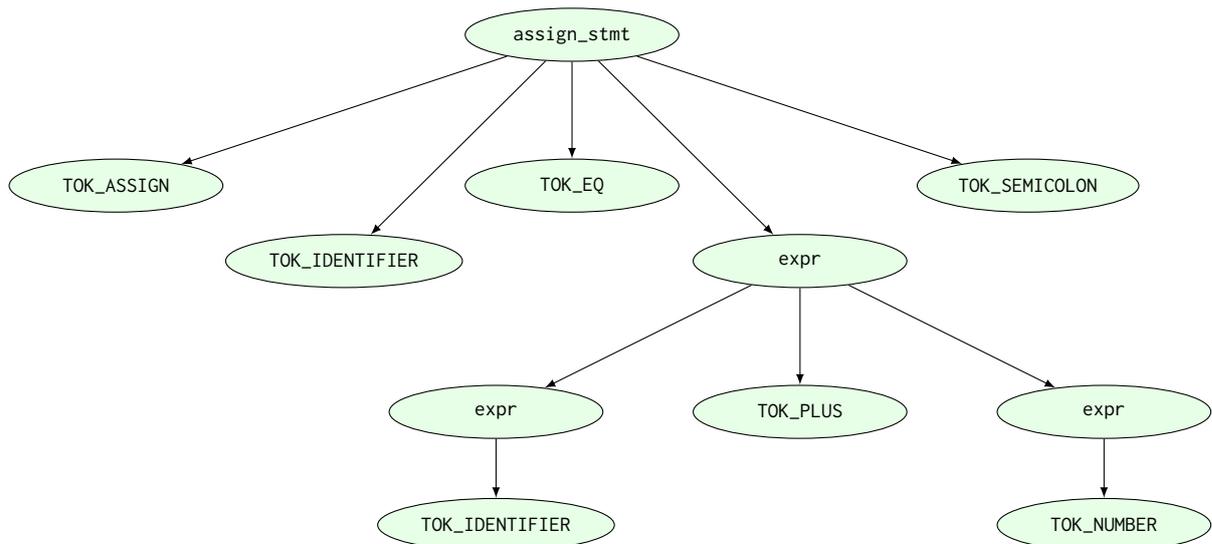
Figure 2.3: Example parse tree for the Verilog expression "`assign foo = bar + 42;`".
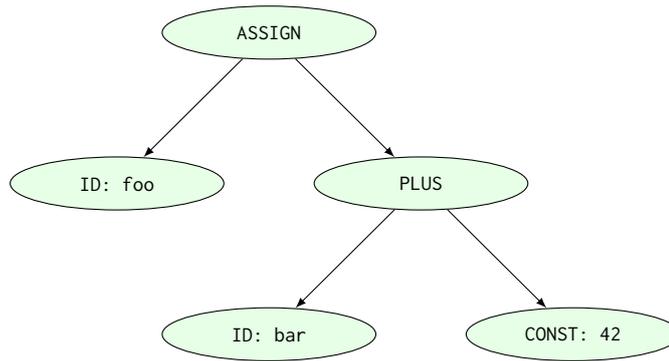
Figure 2.4: Example abstract syntax tree for the Verilog expression "`assign foo = bar + 42;`".

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 pass it to the parser. Instead they run 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 must the complete design 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 produce 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.
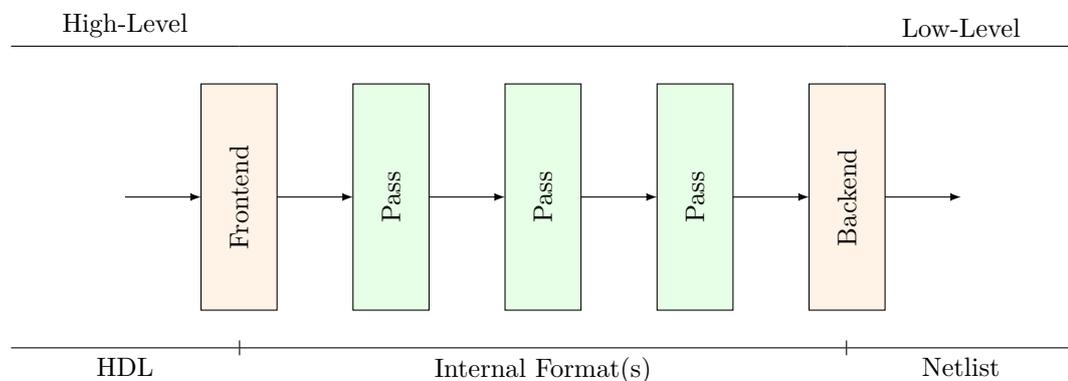
Figure 3.1: General data- and control-flow of a synthesis tool

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 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 rearranged 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 performs 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 unsupported 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 [25] cells.lib:

```
1  # read input file to 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 easily accessible, universal, and vendor-independent synthesis tool, as well as 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 [23], the openMSP430 CPU [22], the OpenCores I^{2}C master [20] and the k68 CPU [21].

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 RTLIL 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 RTLIL Backend for writing the RTLIL data in the same format that is understood by the RTLIL Frontend.

With the exception of the AST Frontend, which 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).
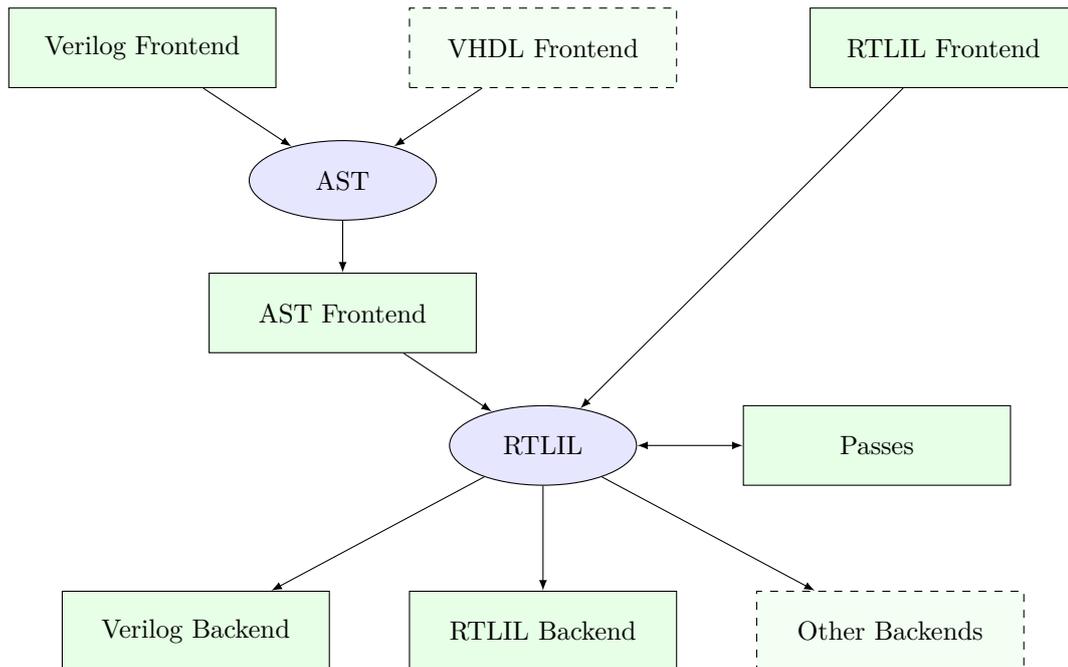
Figure 4.1: Yosys simplified data flow (ellipses: data structures, rectangles: program modules)

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 to 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 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 reinventing 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 which passes operate on, frontends add data to 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.
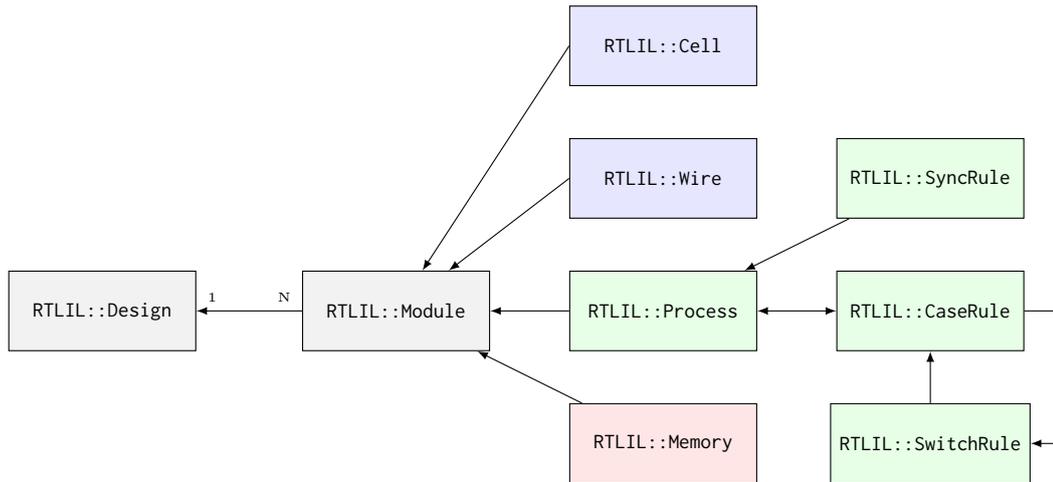
Figure 4.2: Simplified RTLIL Entity-Relationship Diagram

- 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 rationale 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:

- First, it is impossible that an auto-generated identifier collides with an identifier that was provided by the user.

- Second, 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" tries 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.

- Third, 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_*.

Whitespace and control characters (any character with an ASCII code 32 or less) are not allowed in RTLIL identifiers; most frontends and backends cannot support these characters in identifiers.

In order to avoid programming errors, the RTLIL data structures check if all identifiers start with either a backslash or a dollar sign, and contain no whitespace or control characters. Violating these rules results in a runtime error.

All RTLIL identifiers are case sensitive.

Some transformations, such as flattening, may have to change identifiers provided by the user to avoid name collisions. When that happens, attribute "`hdlname`" is attached to the object with the changed identifier. This attribute contains one name (if emitted directly by the frontend, or is a result of disambiguation) or multiple names separated by spaces (if a result of flattening). All names specified in the "`hdlname`" attribute are public and do not include the leading "\".

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 from being 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 (buses are just wires with a width > 1)

- Bus direction (MSB to LSB or vice versa)

- Lowest valid bit index (LSB or MSB depending on bus direction)

- 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 passes.

In Yosys, buses (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.

In Verilog and VHDL, busses may have arbitrary bounds, and LSB can have either the lowest or the highest bit index. In RTLIL, bit 0 always corresponds to LSB; however, information from the HDL frontend is preserved so that the bus will be correctly indexed in error messages, backend output, constraint files, etc.

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 port. 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:

```
module ff_with_en_and_async_reset(clock, reset, enable, d, q);
input clock, reset, enable, d;
output reg q;
always @(posedge clock, posedge reset)
        if (reset)
                q <= 0;
        else if (enable)
                q <= d;
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 RTLIL syntax:

```
process $proc$ff_with_en_and_async_reset.v:4$1
        assign $0\q[0:0] \q
        switch \reset
                case 1'1
                        assign $0\q[0:0] 1'0
                case
                        switch \enable
                                case 1'1
                                        assign $0\q[0:0] \d
                                case
                        end
        end
        sync posedge \clock
                update \q $0\q[0:0]
        sync posedge \reset
                update \q $0\q[0:0]
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), zero or more assignments (RTLIL::SigSig), and zero or more memory writes (RTLIL::MemWriteAction). The always synchronization condition is used to break combinatorial loops when a latch should be inferred instead.

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[1] 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).

A case can specify zero or more compare values that will determine whether it matches. Each of the compare values must be the exact same width as the control signal. When more than one compare value is specified, the case matches if any of them matches the control signal; when zero compare values are specified, the case always matches (i.e. it is the default case).

A switch prioritizes cases from first to last: multiple cases can match, but only the first matched case becomes active. This normally synthesizes to a priority encoder. The `parallel_case` attribute allows passes to assume that no more than one case will match, and `full_case` attribute allows passes to assume that exactly one case will match; if these invariants are ever dynamically violated, the behavior is undefined. These attributes are useful when an invariant invisible to the synthesizer causes the control signal to never take certain bit patterns.

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:

```
1   process $proc$ff_with_en_and_async_reset.v:4$1
2           assign $0\q[0:0] \q
3           switch \enable
4                   case 1'1
5                           assign $0\q[0:0] \d
6                   case
7           end
8           sync posedge \clock
9                   update \q $0\q[0:0]
10          sync high \reset
11                  update \q 1'0
12  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 into e.g. a d-type flip-flop with asynchronous reset and a multiplexer for the enable signal:

---

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

```
1  cell $adff $procdff$6
2         parameter \ARST_POLARITY 1'1
3         parameter \ARST_VALUE 1'0
4         parameter \CLK_POLARITY 1'1
5         parameter \WIDTH 1
6         connect \ARST \reset
7         connect \CLK \clock
8         connect \D $0\q[0:0]
9         connect \Q \q
10 end
11 cell $mux $procmux$3
12         parameter \WIDTH 1
13         connect \A \q
14         connect \B \d
15         connect \S \enable
16         connect \Y $0\q[0:0]
17 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 tasks.

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. Memory initialization is transformed to `$meminit` cells by the language frontend. 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 about 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 separated 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 specifically 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. 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 into 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_expr, opt_muxtree, opt_reduce, opt_rmdff, opt_rmunused and opt_merge.

- 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.

Good starting points for reading example source code to learn how to write passes are `passes/opt/opt_rmdff.cc` and `passes/opt/opt_merge.cc`.

See the top-level README file for a quick *Getting Started* guide and build instructions. The Yosys build is based solely on Makefiles.

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 in order to programming extensions to Yosys or just browse 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/common/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`.

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.

Table 5.1 lists all cells for unary RTL operators.

For the unary cells that output a logical value ($reduce_and, $reduce_or, $reduce_xor, $reduce_xnor, $reduce_bool, $logic_not), when the \Y_WIDTH parameter is greater than 1, the output is zero-extended, and only the least significant bit varies.

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.

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.

The $shl and $shr cells implement logical shifts, whereas the $sshl and $sshr cells implement arithmetic shifts. The $shl and $sshl cells implement the same operation. All four of these cells interpret the second operand as unsigned, and require \B_SIGNED to be zero.

Two additional shift operator cells are available that do not directly correspond to any operator in Verilog, $shift and $shiftx. The $shift cell performs a right logical shift if the second operand is positive (or unsigned), and a left logical shift if it is negative. The $shiftx cell performs the same operation as the $shift cell, but the vacated bit positions are filled with undef (x) bits, and corresponds to the Verilog indexed part-select expression.

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

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
[N/A]	$divfloor
[N/A]	$modfoor
Y = A ** B	$pow

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

For the binary cells that output a logical value ($logic_and, $logic_or, $eqx, $nex, $lt, $le, $eq, $ne, $ge, $gt), when the \Y_WIDTH parameter is greater than 1, the output is zero-extended, and only the least significant bit varies.

Division and modulo cells are available in two rounding modes. The original $div and $mod cells are based on truncating division, and correspond to the semantics of the verilog / and % operators. The $divfloor and $modfloor cells represent flooring division and flooring modulo, the latter of which is also known as "remainder" in several languages. See table 5.3 for a side-by-side comparison between the different semantics.

Division	Result	Truncating		Flooring	
		$div	$mod	$divfloor	$modfloor
-10 / 3	-3.3	-3	-1	-4	2
10 / -3	-3.3	-3	1	-4	-2
-10 / -3	3.3	3	-1	3	-1
10 / 3	3.3	3	1	3	1

Table 5.3: Comparison between different rounding modes for division and modulo cells.

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 output \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 `$tribuf` cell is used to implement tristate logic. Cells of this type have a `\WIDTH` parameter and inputs `\A` and `\EN` and an output `\Y`. The `\A` input and `\Y` output are `\WIDTH` bits wide, and the `\EN` input is one bit wide. When `\EN` is 0, the output `\Y` is not driven. When `\EN` is 1, the value from `\A` input is sent to the `\Y` output. Therefore, the `$tribuf` cell implements the function `Y = EN ? A : 'bz`.

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

SR-type latches are represented by `$sr` cells. These cells have input ports `\SET` and `\CLR` and an output port `\Q`. They have the following parameters:

- `\WIDTH`
  The width of inputs `\SET` and `\CLR` and output `\Q`.

- `\SET_POLARITY`
  The set input bits are active-high if this parameter has the value `1'b1` and active-low if this parameter is `1'b0`.

- `\CLR_POLARITY`
  The reset input bits are active-high if this parameter has the value `1'b1` and active-low if this parameter is `1'b0`.

Both set and reset inputs have separate bits for every output bit. When both the set and reset inputs of an `$sr` cell are active for a given bit index, the reset input takes precedence.

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 reset 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 active-high if this parameter has the value `1'b1` and active-low if this parameter is `1'b0`.

- `\ARST_VALUE`
  The state of `\Q` will be set to this value when the reset is active.

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

D-type flip-flops with synchronous reset are represented by `$sdff` cells. As the `$dff` cells they have `\CLK`, `\D` and `\Q` ports. In addition they also have a single-bit `\SRST` input port for the reset pin and the following additional two parameters:

- \SRST_POLARITY
  The synchronous reset is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

- \SRST_VALUE
  The state of \Q will be set to this value when the reset is active.

Note that the $adff and $sdff cells can only be used when the reset value is constant.

D-type flip-flops with asynchronous load are represented by $aldff cells. As the $dff cells they have \CLK, \D and \Q ports. In addition they also have a single-bit \ALOAD input port for the async load enable pin, a \AD input port with the same width as data for the async load data, and the following additional parameter:

- \ALOAD_POLARITY
  The asynchronous load is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

D-type flip-flops with asynchronous set and reset are represented by $dffsr cells. As the $dff cells they have \CLK, \D and \Q ports. In addition they also have multi-bit \SET and \CLR input ports and the corresponding polarity parameters, like $sr cells.

D-type flip-flops with enable are represented by $dffe, $adffe, $aldffe, $dffsre, $sdffe, and $sdffce cells, which are enhanced variants of $dff, $adff, $aldff, $dffsr, $sdff (with reset over enable) and $sdff (with enable over reset) cells, respectively. They have the same ports and parameters as their base cell. In addition they also have a single-bit \EN input port for the enable pin and the following parameter:

- \EN_POLARITY
  The enable input is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

D-type latches are represented by $dlatch cells. These cells have an enable port \EN, an input port \D, and an output port \Q. The following parameters are available for $dlatch cells:

- \WIDTH
  The width of input \D and output \Q.

- \EN_POLARITY
  The enable input is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

The latch is transparent when the \EN input is active.

D-type latches with reset are represented by $adlatch cells. In addition to $dlatch ports and parameters, they also have a single-bit \ARST input port for the reset pin and the following additional parameters:

- \ARST_POLARITY
  The asynchronous reset is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

- \ARST_VALUE
  The state of \Q will be set to this value when the reset is active.

D-type latches with set and reset are represented by $dlatchsr cells. In addition to $dlatch ports and parameters, they also have multi-bit \SET and \CLR input ports and the corresponding polarity parameters, like $sr cells.

5.1.5 Memories

Memories are either represented using RTLIL::Memory objects, `$memrd_v2`, `$memwr_v2`, and `$meminit_v2` cells, or by `$mem_v2` cells alone.

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 `$memrd_v2` (read port) or `$memwr_v2` (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. In this alternative, memory initialization data is represented by `$meminit_v2` cells, which allow delaying constant folding for initialization addresses and data until after the frontend finishes.

The `$memrd_v2` cells have a clock input \CLK, an enable input \EN, an address input \ADDR, a data output \DATA, an asynchronous reset input \ARST, and a synchronous reset input \SRST. 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). Note that this may be a power-of-two multiple of the underlying memory's width – such ports are called wide ports and access an aligned group of cells at once. In this case, the corresponding low bits of \ADDR must be tied to 0.

- \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.

- \TRANSPARENCY_MASK
  This parameter is a bitmask of write ports that this read port is transparent with. The bits of this parameter are indexed by the write port's \PORTID parameter. Transparency can only be enabled between synchronous ports sharing a clock domain. When transparency is enabled for a given port pair, a read and write to the same address in the same cycle will return the new value. Otherwise the old value is returned.

- \COLLISION_X_MASK
  This parameter is a bitmask of write ports that have undefined collision behavior with this port. The bits of this parameter are indexed by the write port's \PORTID parameter. This behavior can only be enabled between synchronous ports sharing a clock domain. When undefined collision is enabled for a given port pair, a read and write to the same address in the same cycle will return the undefined (all-X) value. This option is exclusive (for a given port pair) with the transparency option.

- \ARST_VALUE
  Whenever the \ARST input is asserted, the data output will be reset to this value. Only used for synchronous ports.

- \SRST_VALUE
  Whenever the \SRST input is synchronously asserted, the data output will be reset to this value. Only used for synchronous ports.

- \INIT_VALUE
  The initial value of the data output, for synchronous ports.

- \CE_OVER_SRST

  If this parameter is non-zero, the \SRST input is only recognized when \EN is true. Otherwise, \SRST is recognized regardless of \EN.

The $memwr_v2 cells have a clock input \CLK, an enable input \EN (one enable bit for each data bit), 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 write 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). Like with $memrd_v2 cells, the width is allowed to be any power-of-two multiple of memory width, with the corresponding restriction on address.

- \CLK_ENABLE

  When this parameter is non-zero, the clock is used. Otherwise this write 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.

- \PORTID

  An identifier for this write port, used to index write port bit mask parameters.

- \PRIORITY_MASK

  This parameter is a bitmask of write ports that this write port has priority over in case of writing to the same address. The bits of this parameter are indexed by the other write port's \PORTID parameter. Write ports can only have priority over write ports with lower port ID. When two ports write to the same address and neither has priority over the other, the result is undefined. Priority can only be set between two synchronous ports sharing the same clock domain.

The $meminit_v2 cells have an address input \ADDR, a data input \DATA, with the width of the \DATA port equal to \WIDTH parameter times \WORDS parameter, and a bit enable mask input \EN with width equal to \WIDTH parameter. All three of the inputs must resolve to a constant for synthesis to succeed.

- \MEMID

  The name of the RTLIL::Memory object that is associated with this initialization cell.

- \ABITS

  The number of address bits (width of the \ADDR input port).

- \WIDTH

  The number of data bits per memory location.

- \WORDS

  The number of consecutive memory locations initialized by this cell.

- \PRIORITY

  The cell with the higher integer value in this parameter wins an initialization conflict.

The HDL frontend models a memory using RTLIL::Memory objects and asynchronous `$memrd_v2` and `$memwr_v2` cells. The `memory` pass (i.e. its various sub-passes) migrates `$dff` cells into the `$memrd_v2` and `$memwr_v2` cells making them synchronous, then converts them to a single `$mem_v2` 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_v2` cell is left in the design.

The `$mem_v2` cell provides the following parameters:

- `\MEMID`
  The name of the original RTLIL::Memory object that became this `$mem_v2` cell.

- `\SIZE`
  The number of words in the memory.

- `\ABITS`
  The number of address bits.

- `\WIDTH`
  The number of data bits per word.

- `\INIT`
  The initial memory contents.

- `\RD_PORTS`
  The number of read ports on this memory cell.

- `\RD_WIDE_CONTINUATION`
  This parameter is `\RD_PORTS` bits wide, containing a bitmask of "wide continuation" read ports. Such ports are used to represent the extra data bits of wide ports in the combined cell, and must have all control signals identical with the preceding port, except for address, which must have the proper sub-cell address encoded in the low bits.

- `\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.

- `\RD_TRANSPARENCY_MASK`
  This parameter is `\RD_PORTS*WR_PORTS` bits wide, containing a concatenation of all `\TRANSPARENCY_MASK` values of the original `$memrd_v2` cells.

- `\RD_COLLISION_X_MASK`
  This parameter is `\RD_PORTS*WR_PORTS` bits wide, containing a concatenation of all `\COLLISION_X_MASK` values of the original `$memrd_v2` cells.

- `\RD_CE_OVER_SRST`
  This parameter is `\RD_PORTS` bits wide, determining relative synchronous reset and enable priority for each read port.

- `\RD_INIT_VALUE`
  This parameter is `\RD_PORTS*WIDTH` bits wide, containing the initial value for each synchronous read port.

- `\RD_ARST_VALUE`
  This parameter is `\RD_PORTS*WIDTH` bits wide, containing the asynchronous reset value for each synchronous read port.

- `\RD_SRST_VALUE`

  This parameter is `\RD_PORTS*WIDTH` bits wide, containing the synchronous reset value for each synchronous read port.

- `\WR_PORTS`

  The number of write ports on this memory cell.

- `\WR_WIDE_CONTINUATION`

  This parameter is `\WR_PORTS` bits wide, containing a bitmask of "wide continuation" write ports.

- `\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.

- `\WR_PRIORITY_MASK`

  This parameter is `\WR_PORTS*WR_PORTS` bits wide, containing a concatenation of all `\PRIORITY_MASK` values of the original `$memwr_v2` cells.

The `$mem_v2` cell has the following ports:

- `\RD_CLK`

  This input is `\RD_PORTS` bits wide, containing all clock signals for the read ports.

- `\RD_EN`

  This input is `\RD_PORTS` bits wide, containing all enable 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.

- `\RD_ARST`

  This input is `\RD_PORTS` bits wide, containing all asynchronous reset signals for the read ports.

- `\RD_SRST`

  This input is `\RD_PORTS` bits wide, containing all synchronous reset 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*\WIDTH` 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 `memory_collect` pass can be used to convert discrete `$memrd_v2`, `$memwr_v2`, and `$meminit_v2` cells belonging to the same memory to a single `$mem_v2` cell, whereas the `memory_unpack` pass performs the inverse operation. The `memory_dff` pass can combine asynchronous memory ports that are fed by or feeding registers into synchronous memory ports. The `memory_bram` pass can be used to recognize `$mem_v2` cells that can be implemented with a block RAM resource on an FPGA. The `memory_map` pass can be used to implement `$mem_v2` cells as basic logic: word-wide DFFs and address decoders.

5.1.6 Finite State Machines

FIXME:

Add a brief description of the `$fsm` cell type.

5.1.7 Specify rules

FIXME:

Add information about `$specify2`, `$specify3`, and `$specrule` cells.

5.1.8 Formal verification cells

FIXME:

Add information about `$assert`, `$assume`, `$live`, `$fair`, `$cover`, `$equiv`, `$initstate`, `$anyconst`, `$anyseq`, `$anyinit`, `$allconst`, `$allseq` cells.

FIXME:

Add information about `$ff` and `$_FF_` cells.

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/common/simcells.v` in the Yosys source tree.

Tables 5.4, 5.6, 5.5, 5.7, 5.8, 5.9, 5.10, 5.11 and 5.12 list all cell types used for gate level logic. The cell types `$_BUF_`, `$_NOT_`, `$_AND_`, `$_NAND_`, `$_ANDNOT_`, `$_OR_`, `$_NOR_`, `$_ORNOT_`, `$_XOR_`, `$_XNOR_`, `$_AOI3_`, `$_OAI3_`, `$_AOI4_`, `$_OAI4_`, `$_MUX_`, `$_MUX4_`, `$_MUX8_`, `$_MUX16_` and `$_NMUX_` are used to model combinatorial logic. The cell type `$_TBUF_` is used to model tristate logic.

The `$_MUX4_`, `$_MUX8_` and `$_MUX16_` cells are used to model wide muxes, and correspond to the following Verilog code:

```
// $_MUX4_
assign Y = T ? (S ? D : C) :
               (S ? B : A);
// $_MUX8_
assign Y = U ? T ? (S ? H : G) :
                   (S ? F : E) :
               T ? (S ? D : C) :
                   (S ? B : A);
// $_MUX16_
assign Y = V ? U ? T ? (S ? P : O) :
                       (S ? N : M) :
                   T ? (S ? L : K) :
                       (S ? J : I) :
               U ? T ? (S ? H : G) :
                       (S ? F : E) :
                   T ? (S ? D : C) :
                       (S ? B : A);
```

The cell types `$_DFF_N_` and `$_DFF_P_` represent d-type flip-flops.

The cell types `$_DFFE_[NP][NP]_` implement d-type flip-flops with enable. The values in the table for these cell types relate to the following Verilog code template.

Verilog	Cell Type
Y = A	$_BUF_
Y = ~A	$_NOT_
Y = A & B	$_AND_
Y = ~(A & B)	$_NAND_
Y = A & ~B	$_ANDNOT_
Y = A | B	$_OR_
Y = ~(A | B)	$_NOR_
Y = A | ~B	$_ORNOT_
Y = A ^ B	$_XOR_
Y = ~(A ^ B)	$_XNOR_
Y = ~((A & B) | C)	$_AOI3_
Y = ~((A | B) & C)	$_OAI3_
Y = ~((A & B) | (C & D))	$_AOI4_
Y = ~((A | B) & (C | D))	$_OAI4_
Y = S ? B : A	$_MUX_
Y = ~(S ? B : A)	$_NMUX_
(see below)	$_MUX4_
(see below)	$_MUX8_
(see below)	$_MUX16_
Y = EN ? A : 1'bz	$_TBUF_
always @(**negedge** C) Q <= D	$_DFF_N_
always @(**posedge** C) Q <= D	$_DFF_P_
always @* **if** (!E) Q <= D	$_DLATCH_N_
always @* **if** (E) Q <= D	$_DLATCH_P_

Table 5.4: Cell types for gate level logic networks (main list)

```
always @(ClkEdge C)
        if (EN == EnLvl)
                Q <= D;
```

The cell types $_DFF_[NP][NP][01]_ implement d-type flip-flops with asynchronous reset. 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;
```

The cell types $_SDFF_[NP][NP][01]_ implement d-type flip-flops with synchronous reset. The values in the table for these cell types relate to the following Verilog code template:

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

The cell types $_DFFE_[NP][NP][01][NP]_ implement d-type flip-flops with asynchronous reset and enable. 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.

$ClkEdge$	$RstLvl$	$RstVal$	Cell Type
negedge	0	0	$_DFF_NN0_, $_SDFF_NN0_
negedge	0	1	$_DFF_NN1_, $_SDFF_NN1_
negedge	1	0	$_DFF_NP0_, $_SDFF_NP0_
negedge	1	1	$_DFF_NP1_, $_SDFF_NP1_
posedge	0	0	$_DFF_PN0_, $_SDFF_PN0_
posedge	0	1	$_DFF_PN1_, $_SDFF_PN1_
posedge	1	0	$_DFF_PP0_, $_SDFF_PP0_
posedge	1	1	$_DFF_PP1_, $_SDFF_PP1_

Table 5.5: Cell types for gate level logic networks (FFs with reset)

$ClkEdge$	$EnLvl$	Cell Type
negedge	0	$_DFFE_NN_
negedge	1	$_DFFE_NP_
posedge	0	$_DFFE_PN_
posedge	1	$_DFFE_PP_

Table 5.6: Cell types for gate level logic networks (FFs with enable)

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

The cell types $_SDFFE_[NP][NP][01][NP]_ implement d-type flip-flops with synchronous reset and enable, with reset having priority over enable. The values in the table for these cell types relate to the following Verilog code template:

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

The cell types $_SDFFCE_[NP][NP][01][NP]_ implement d-type flip-flops with synchronous reset and enable, with enable having priority over reset. The values in the table for these cell types relate to the following Verilog code template:

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

The cell types $_DFFSR_[NP][NP][NP]_ implement d-type flip-flops with asynchronous set and reset. The values in the table for these cell types relate to the following Verilog code template, where $RstEdge$ is **posedge** if $RstLvl$ if 1, **negedge** otherwise, and $SetEdge$ is **posedge** if $SetLvl$ if 1, **negedge** otherwise.

```
always @(ClkEdge C, RstEdge R, SetEdge S)
        if (R == RstLvl)
                Q <= 0;
        else if (S == SetLvl)
                Q <= 1;
```

ClkEdge	RstLvl	RstVal	EnLvl	Cell Type
negedge	0	0	0	$_DFFE_NN0N_, $_SDFFE_NN0N_, $_SDFFCE_NN0N_
negedge	0	0	1	$_DFFE_NN0P_, $_SDFFE_NN0P_, $_SDFFCE_NN0P_
negedge	0	1	0	$_DFFE_NN1N_, $_SDFFE_NN1N_, $_SDFFCE_NN1N_
negedge	0	1	1	$_DFFE_NN1P_, $_SDFFE_NN1P_, $_SDFFCE_NN1P_
negedge	1	0	0	$_DFFE_NP0N_, $_SDFFE_NP0N_, $_SDFFCE_NP0N_
negedge	1	0	1	$_DFFE_NP0P_, $_SDFFE_NP0P_, $_SDFFCE_NP0P_
negedge	1	1	0	$_DFFE_NP1N_, $_SDFFE_NP1N_, $_SDFFCE_NP1N_
negedge	1	1	1	$_DFFE_NP1P_, $_SDFFE_NP1P_, $_SDFFCE_NP1P_
posedge	0	0	0	$_DFFE_PN0N_, $_SDFFE_PN0N_, $_SDFFCE_PN0N_
posedge	0	0	1	$_DFFE_PN0P_, $_SDFFE_PN0P_, $_SDFFCE_PN0P_
posedge	0	1	0	$_DFFE_PN1N_, $_SDFFE_PN1N_, $_SDFFCE_PN1N_
posedge	0	1	1	$_DFFE_PN1P_, $_SDFFE_PN1P_, $_SDFFCE_PN1P_
posedge	1	0	0	$_DFFE_PP0N_, $_SDFFE_PP0N_, $_SDFFCE_PP0N_
posedge	1	0	1	$_DFFE_PP0P_, $_SDFFE_PP0P_, $_SDFFCE_PP0P_
posedge	1	1	0	$_DFFE_PP1N_, $_SDFFE_PP1N_, $_SDFFCE_PP1N_
posedge	1	1	1	$_DFFE_PP1P_, $_SDFFE_PP1P_, $_SDFFCE_PP1P_

Table 5.7: Cell types for gate level logic networks (FFs with reset and enable)

ClkEdge	SetLvl	RstLvl	Cell Type
negedge	0	0	$_DFFSR_NNN_
negedge	0	1	$_DFFSR_NNP_
negedge	1	0	$_DFFSR_NPN_
negedge	1	1	$_DFFSR_NPP_
posedge	0	0	$_DFFSR_PNN_
posedge	0	1	$_DFFSR_PNP_
posedge	1	0	$_DFFSR_PPN_
posedge	1	1	$_DFFSR_PPP_

Table 5.8: Cell types for gate level logic networks (FFs with set and reset)

```
        else
                Q <= D;
```

The cell types $_DFFSRE_[NP][NP][NP][NP]_ implement d-type flip-flops with asynchronous set and reset and enable. The values in the table for these cell types relate to the following Verilog code template, where $RstEdge$ is **posedge** if $RstLvl$ if 1, **negedge** otherwise, and $SetEdge$ is **posedge** if $SetLvl$ if 1, **negedge** otherwise.

```
        always @(ClkEdge c, RstEdge R, SetEdge S)
                if (R == RstLvl)
                        Q <= 0;
                else if (S == SetLvl)
                        Q <= 1;
                else if (E == EnLvl)
                        Q <= D;
```

The cell types $_DLATCH_N_ and $_DLATCH_P_ represent d-type latches.

The cell types $_DLATCH_[NP][NP][01]_ implement d-type latches with reset. The values in the table for these cell types relate to the following Verilog code template:

```
        always @*
                if (R == RstLvl)
                        Q <= RstVal;
                else if (E == EnLvl)
```

ClkEdge	SetLvl	RstLvl	EnLvl	Cell Type
negedge	0	0	0	$_DFFSRE_NNNN_
negedge	0	0	1	$_DFFSRE_NNNP_
negedge	0	1	0	$_DFFSRE_NNPN_
negedge	0	1	1	$_DFFSRE_NNPP_
negedge	1	0	0	$_DFFSRE_NPNN_
negedge	1	0	1	$_DFFSRE_NPNP_
negedge	1	1	0	$_DFFSRE_NPPN_
negedge	1	1	1	$_DFFSRE_NPPP_
posedge	0	0	0	$_DFFSRE_PNNN_
posedge	0	0	1	$_DFFSRE_PNNP_
posedge	0	1	0	$_DFFSRE_PNPN_
posedge	0	1	1	$_DFFSRE_PNPP_
posedge	1	0	0	$_DFFSRE_PPNN_
posedge	1	0	1	$_DFFSRE_PPNP_
posedge	1	1	0	$_DFFSRE_PPPN_
posedge	1	1	1	$_DFFSRE_PPPP_

Table 5.9: Cell types for gate level logic networks (FFs with set and reset and enable)

EnLvl	RstLvl	RstVal	Cell Type
0	0	0	$_DLATCH_NN0_
0	0	1	$_DLATCH_NN1_
0	1	0	$_DLATCH_NP0_
0	1	1	$_DLATCH_NP1_
1	0	0	$_DLATCH_PN0_
1	0	1	$_DLATCH_PN1_
1	1	0	$_DLATCH_PP0_
1	1	1	$_DLATCH_PP1_

Table 5.10: Cell types for gate level logic networks (latches with reset)

```
Q <= D;
```

The cell types $_DLATCHSR_[NP][NP][NP]_ implement d-type latches with set and reset. The values in the table for these cell types relate to the following Verilog code template:

```
always @*
        if (R == RstLvl)
                Q <= 0;
        else if (S == SetLvl)
                Q <= 1;
        else if (E == EnLvl)
                Q <= D;
```

The cell types $_SR_[NP][NP]_ implement sr-type latches. The values in the table for these cell types relate to the following Verilog code template:

```
always @*
        if (R == RstLvl)
                Q <= 0;
        else if (S == SetLvl)
                Q <= 1;
```

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

EnLvl	SetLvl	RstLvl	Cell Type
0	0	0	$_DLATCHSR_NNN_
0	0	1	$_DLATCHSR_NNP_
0	1	0	$_DLATCHSR_NPN_
0	1	1	$_DLATCHSR_NPP_
1	0	0	$_DLATCHSR_PNN_
1	0	1	$_DLATCHSR_PNP_
1	1	0	$_DLATCHSR_PPN_
1	1	1	$_DLATCHSR_PPP_

Table 5.11: Cell types for gate level logic networks (latches with set and reset)

SetLvl	RstLvl	Cell Type
0	0	$_SR_NN_
0	1	$_SR_NP_
1	0	$_SR_PN_
1	1	$_SR_PP_

Table 5.12: Cell types for gate level logic networks (SR latches)

the dfflibmap pass. The combinatorial logic cells can be mapped to physical cells from a Liberty file via ABC [27] using the abc pass.

FIXME:

Add information about $slice and $concat cells.

FIXME:

Add information about $lut and $sop cells.

FIXME:

Add information about $alu, $macc, $fa, and $lcu cells.

Chapter 6

Programming Yosys Extensions

This chapter contains some bits and pieces of information about programming yosys extensions. Also consult the section on programming in the "Yosys Presentation" (can be downloaded from the Yosys website as PDF) and don't be afraid to ask questions on the YosysHQ Slack.

6.1 Guidelines

The `guidelines` directory contains notes on various aspects of Yosys development. The files `GettingStarted` and `CodingStyle` may be of particular interest, and are reproduced here.

GettingStarted

```
 1  Getting Started
 2  ===============
 3
 4
 5  Outline of a Yosys command
 6  --------------------------
 7
 8  Here is a the C++ code for a "hello_world" Yosys command (hello.cc):
 9
10          #include "kernel/yosys.h"
11
12          USING_YOSYS_NAMESPACE
13          PRIVATE_NAMESPACE_BEGIN
14
15          struct HelloWorldPass : public Pass {
16                  HelloWorldPass() : Pass("hello_world") { }
17                  void execute(vector<string>, Design*) override {
18                          log("Hello World!\n");
19                  }
20          } HelloWorldPass;
21
22          PRIVATE_NAMESPACE_END
23
24  This can be built into a Yosys module using the following command:
25
26          yosys-config --exec --cxx --cxxflags --ldflags -o hello.so -shared hello.cc --ldlibs
```

```
27
28   Or short:
29
30          yosys-config --build hello.so hello.cc
31
32   And then executed using the following command:
33
34          yosys -m hello.so -p hello_world
35
36
37   Yosys Data Structures
38   ---------------------
39
40   Here is a short list of data structures that you should make yourself familiar
41   with before you write C++ code for Yosys. The following data structures are all
42   defined when "kernel/yosys.h" is included and USING_YOSYS_NAMESPACE is used.
43
44     1. Yosys Container Classes
45
46   Yosys uses dict<K, T> and pool<T> as main container classes. dict<K, T> is
47   essentially a replacement for std::unordered_map<K, T> and pool<T> is a
48   replacement for std::unordered_set<T>. The main characteristics are:
49
50          - dict<K, T> and pool<T> are about 2x faster than the std containers
51
52          - references to elements in a dict<K, T> or pool<T> are invalidated by
53            insert and remove operations (similar to std::vector<T> on push_back()).
54
55          - some iterators are invalidated by erase(). specifically, iterators
56            that have not passed the erased element yet are invalidated. (erase()
57            itself returns valid iterator to the next element.)
58
59          - no iterators are invalidated by insert(). elements are inserted at
60            begin(). i.e. only a new iterator that starts at begin() will see the
61            inserted elements.
62
63          - the method .count(key, iterator) is like .count(key) but only
64            considers elements that can be reached via the iterator.
65
66          - iterators can be compared. it1 < it2 means that the position of t2
67            can be reached via t1 but not vice versa.
68
69          - the method .sort() can be used to sort the elements in the container
70            the container stays sorted until elements are added or removed.
71
72          - dict<K, T> and pool<T> will have the same order of iteration across
73            all compilers, standard libraries and architectures.
74
75   In addition to dict<K, T> and pool<T> there is also an idict<K> that
76   creates a bijective map from K to the integers. For example:
77
78          idict<string, 42> si;
79          log("%d\n", si("hello"));     // will print 42
80          log("%d\n", si("world"));     // will print 43
```

```
 81            log("%d\n", si.at("world"));   // will print 43
 82            log("%d\n", si.at("dummy"));   // will throw exception
 83            log("%s\n", si[42].c_str()));  // will print hello
 84            log("%s\n", si[43].c_str()));  // will print world
 85            log("%s\n", si[44].c_str()));  // will throw exception
 86
 87  It is not possible to remove elements from an idict.
 88
 89  Finally mfp<K> implements a merge-find set data structure (aka. disjoint-set or
 90  union-find) over the type K ("mfp" = merge-find-promote).
 91
 92    2. Standard STL data types
 93
 94  In Yosys we use std::vector<T> and std::string whenever applicable. When
 95  dict<K, T> and pool<T> are not suitable then std::map<K, T> and std::set<T>
 96  are used instead.
 97
 98  The types std::vector<T> and std::string are also available as vector<T>
 99  and string in the Yosys namespace.
100
101    3. RTLIL objects
102
103  The current design (essentially a collection of modules, each defined by a
104  netlist) is stored in memory using RTLIL object (declared in kernel/rtlil.h,
105  automatically included by kernel/yosys.h). You should glance over at least
106  the declarations for the following types in kernel/rtlil.h:
107
108          RTLIL::IdString
109                  This is a handle for an identifier (e.g. cell or wire name).
110                  It feels a lot like a std::string, but is only a single int
111                  in size. (The actual string is stored in a global lookup
112                  table.)
113
114          RTLIL::SigBit
115                  A single signal bit. I.e. either a constant state (0, 1,
116                  x, z) or a single bit from a wire.
117
118          RTLIL::SigSpec
119                  Essentially a vector of SigBits.
120
121          RTLIL::Wire
122          RTLIL::Cell
123                  The building blocks of the netlist in a module.
124
125          RTLIL::Module
126          RTLIL::Design
127                  The module is a container with connected cells and wires
128                  in it. The design is a container with modules in it.
129
130  All this types are also available without the RTLIL:: prefix in the Yosys
131  namespace.
132
133    4. SigMap and other Helper Classes
134
```

```
There are a couple of additional helper classes that are in wide use
in Yosys. Most importantly there is SigMap (declared in kernel/sigtools.h).

When a design has many wires in it that are connected to each other, then a
single signal bit can have multiple valid names. The SigMap object can be used
to map SigSpecs or SigBits to unique SigSpecs and SigBits that consistently
only use one wire from such a group of connected wires. For example:

        SigBit a = module->addWire(NEW_ID);
        SigBit b = module->addWire(NEW_ID);
        module->connect(a, b);

        log("%d\n", a == b); // will print 0

        SigMap sigmap(module);
        log("%d\n", sigmap(a) == sigmap(b)); // will print 1


Using the RTLIL Netlist Format
------------------------------

In the RTLIL netlist format the cell ports contain SigSpecs that point to the
Wires. There are no references in the other direction. This has two direct
consequences:

(1) It is very easy to go from cells to wires but hard to go in the other way.

(2) There is no danger in removing cells from the netlists, but removing wires
can break the netlist format when there are still references to the wire
somewhere in the netlist.

The solution to (1) is easy: Create custom indexes that allow you to make fast
lookups for the wire-to-cell direction. You can either use existing generic
index structures to do that (such as the ModIndex class) or write your own
index. For many application it is simplest to construct a custom index. For
example:

        SigMap sigmap(module);
        dict<SigBit, Cell*> sigbit_to_driver_index;

        for (auto cell : module->cells())
                for (auto &conn : cell->connections())
                        if (cell->output(conn.first))
                                for (auto bit : sigmap(conn.second))
                                        sigbit_to_driver_index[bit] = cell;

Regarding (2): There is a general theme in Yosys that you don't remove wires
from the design. You can rename them, unconnect them, but you do not actually remove
the Wire object from the module. Instead you let the "clean" command take care
of the dangling wires. On the other hand it is safe to remove cells (as long as
you make sure this does not invalidate a custom index you are using in your code).


Example Code
```

```
189  ------------
190
191  The following yosys commands are a good starting point if you are looking for examples
192  of how to use the Yosys API:
193
194          manual/CHAPTER_Prog/stubnets.cc
195          manual/PRESENTATION_Prog/my_cmd.cc
196
197
198  Script Passes
199  -------------
200
201  The ScriptPass base class can be used to implement passes that just call other passes,
202  like a script. Examples for such passes are:
203
204          techlibs/common/prep.cc
205          techlibs/common/synth.cc
206
207  In some cases it is easier to implement such a pass as regular pass, for example when
208  ScriptPass doesn't provide the type of flow control desired. (But many of the
209  script passes in Yosys that don't use ScriptPass simply predate the ScriptPass base
210  class.) Examples for such passes are:
211
212          passes/opt/opt.cc
213          passes/proc/proc.cc
214
215  Whether they use the ScriptPass base-class or not, a pass should always either
216  call other passes without doing any non-trivial work itself, or should implement
217  a non-trivial algorithm but not call any other passes. The reason for this is that
218  this helps containing complexity in individual passes and simplifies debugging the
219  entire system.
220
221  Exceptions to this rule should be rare and limited to cases where calling other
222  passes is optional and only happens when requested by the user (such as for
223  example `techmap -autoproc`), or where it is about commands that are "top-level
224  commands" in their own right, not components to be used in regular synthesis
225  flows (such as the `bugpoint` command).
226
227  A pass that would "naturally" call other passes and also do some work itself
228  should be re-written in one of two ways:
229
230  1) It could be re-written as script pass with the parts that are not calls
231  to other passes factored out into individual new passes. Usually in those
232  cases the new sub passes share the same prefix as the top-level script pass.
233
234  2) It could be re-written so that it already expects the design in a certain
235  state, expecting the calling script to set up this state before calling the
236  pass in questions.
237
238  Many back-ends are examples for the 2nd approach. For example, `write_aiger`
239  does not convert the design into AIG representation, but expects the design
240  to be already in this form, and prints an `Unsupported cell type` error
241  message otherwise.
242
```

```
243
244  Notes on the existing codebase
245  ------------------------------
246
247  For historical reasons not all parts of Yosys adhere to the current coding
248  style. When adding code to existing parts of the system, adhere to this guide
249  for the new code instead of trying to mimic the style of the surrounding code.
```

CodingStyle

```
 1  Coding Style
 2  ============
 3
 4
 5  Formatting of code
 6  ------------------
 7
 8  - Yosys code is using tabs for indentation. Tabs are 8 characters.
 9
10  - A continuation of a statement in the following line is indented by
11    two additional tabs.
12
13  - Lines are as long as you want them to be. A good rule of thumb is
14    to break lines at about column 150.
15
16  - Opening braces can be put on the same or next line as the statement
17    opening the block (if, switch, for, while, do). Put the opening brace
18    on its own line for larger blocks, especially blocks that contains
19    blank lines.
20
21  - Otherwise stick to the Linux Kernel Coding Style:
22      https://www.kernel.org/doc/Documentation/CodingStyle
23
24
25  C++ Language
26  ------------
27
28  Yosys is written in C++11. At the moment only constructs supported by
29  gcc 4.8 are allowed in Yosys code. This will change in future releases.
30
31  In general Yosys uses "int" instead of "size_t". To avoid compiler
32  warnings for implicit type casts, always use "GetSize(foobar)" instead
33  of "foobar.size()". (GetSize() is defined in kernel/yosys.h)
34
35  Use range-based for loops whenever applicable.
```

6.2 The "stubsnets" Example Module

The following is the complete code of the "stubsnets" example module. It is included in the Yosys source distribution as manual/CHAPTER_Prog/stubnets.cc.

stubnets.cc

```
 1  // This is free and unencumbered software released into the public domain.
 2  //
 3  // Anyone is free to copy, modify, publish, use, compile, sell, or
 4  // distribute this software, either in source code form or as a compiled
 5  // binary, for any purpose, commercial or non-commercial, and by any
 6  // means.
 7
 8  #include "kernel/yosys.h"
 9  #include "kernel/sigtools.h"
10
11  #include <string>
12  #include <map>
13  #include <set>
14
15  USING_YOSYS_NAMESPACE
16  PRIVATE_NAMESPACE_BEGIN
17
18  // this function is called for each module in the design
19  static void find_stub_nets(RTLIL::Design *design, RTLIL::Module *module, bool report_bits)
20  {
21          // use a SigMap to convert nets to a unique representation
22          SigMap sigmap(module);
23
24          // count how many times a single-bit signal is used
25          std::map<RTLIL::SigBit, int> bit_usage_count;
26
27          // count output lines for this module (needed only for summary output at the end)
28          int line_count = 0;
29
30          log("Looking for stub wires in module %s:\n", RTLIL::id2cstr(module->name));
31
32          // For all ports on all cells
33          for (auto &cell_iter : module->cells_)
34          for (auto &conn : cell_iter.second->connections())
35          {
36                  // Get the signals on the port
37                  // (use sigmap to get a uniqe signal name)
38                  RTLIL::SigSpec sig = sigmap(conn.second);
39
40                  // add each bit to bit_usage_count, unless it is a constant
41                  for (auto &bit : sig)
42                          if (bit.wire != NULL)
43                                  bit_usage_count[bit]++;
44          }
45
46          // for each wire in the module
47          for (auto &wire_iter : module->wires_)
48          {
49                  RTLIL::Wire *wire = wire_iter.second;
50
51                  // .. but only selected wires
52                  if (!design->selected(module, wire))
53                          continue;
54
```

```
55                          // add +1 usage if this wire actually is a port
56                          int usage_offset = wire->port_id > 0 ? 1 : 0;
57
58                          // we will record which bits of the (possibly multi-bit) wire are stub signals
59                          std::set<int> stub_bits;
60
61                          // get a signal description for this wire and split it into separate bits
62                          RTLIL::SigSpec sig = sigmap(wire);
63
64                          // for each bit (unless it is a constant):
65                          // check if it is used at least two times and add to stub_bits otherwise
66                          for (int i = 0; i < GetSize(sig); i++)
67                                  if (sig[i].wire != NULL && (bit_usage_count[sig[i]] + usage_offset) < 2)
68                                          stub_bits.insert(i);
69
70                          // continue if no stub bits found
71                          if (stub_bits.size() == 0)
72                                  continue;
73
74                          // report stub bits and/or stub wires, don't report single bits
75                          // if called with report_bits set to false.
76                          if (GetSize(stub_bits) == GetSize(sig)) {
77                                  log("  found stub wire: %s\n", RTLIL::id2cstr(wire->name));
78                          } else {
79                                  if (!report_bits)
80                                          continue;
81                                  log("  found wire with stub bits: %s [", RTLIL::id2cstr(wire->name));
82                                  for (int bit : stub_bits)
83                                          log("%s%d", bit == *stub_bits.begin() ? "" : ", ", bit);
84                                  log("]\n");
85                          }
86
87                          // we have outputted a line, increment summary counter
88                          line_count++;
89                  }
90
91          // report summary
92          if (report_bits)
93                  log("  found %d stub wires or wires with stub bits.\n", line_count);
94          else
95                  log("  found %d stub wires.\n", line_count);
96  }
97
98  // each pass contains a singleton object that is derived from Pass
99  struct StubnetsPass : public Pass {
100         StubnetsPass() : Pass("stubnets") { }
101         void execute(std::vector<std::string> args, RTLIL::Design *design) override
102         {
103                 // variables to mirror information from passed options
104                 bool report_bits = 0;
105
106                 log_header(design, "Executing STUBNETS pass (find stub nets).\n");
107
108                 // parse options
```

```
109                      size_t argidx;
110                      for (argidx = 1; argidx < args.size(); argidx++) {
111                              std::string arg = args[argidx];
112                              if (arg == "-report_bits") {
113                                      report_bits = true;
114                                      continue;
115                              }
116                              break;
117                      }
118
119                      // handle extra options (e.g. selection)
120                      extra_args(args, argidx, design);
121
122                      // call find_stub_nets() for each module that is either
123                      // selected as a whole or contains selected objects.
124                      for (auto &it : design->modules_)
125                              if (design->selected_module(it.first))
126                                      find_stub_nets(design, it.second, report_bits);
127              }
128 } StubnetsPass;
129
130 PRIVATE_NAMESPACE_END
```

Makefile

```
1  test: stubnets.so
2          yosys -ql test1.log -m ./stubnets.so test.v -p "stubnets"
3          yosys -ql test2.log -m ./stubnets.so test.v -p "opt;␣stubnets"
4          yosys -ql test3.log -m ./stubnets.so test.v -p "techmap;␣opt;␣stubnets␣-report_bits"
5          tail test1.log test2.log test3.log
6
7  stubnets.so: stubnets.cc
8          yosys-config --exec --cxx --cxxflags --ldflags -o $@ -shared $^ --ldlibs
9
10 clean:
11          rm -f test1.log test2.log test3.log
12          rm -f stubnets.so stubnets.d
```

test.v

```
1  module uut(in1, in2, in3, out1, out2);
2
3  input [8:0] in1, in2, in3;
4  output [8:0] out1, out2;
5
6  assign out1 = in1 + in2 + (in3 >> 4);
7
8  endmodule
```

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.
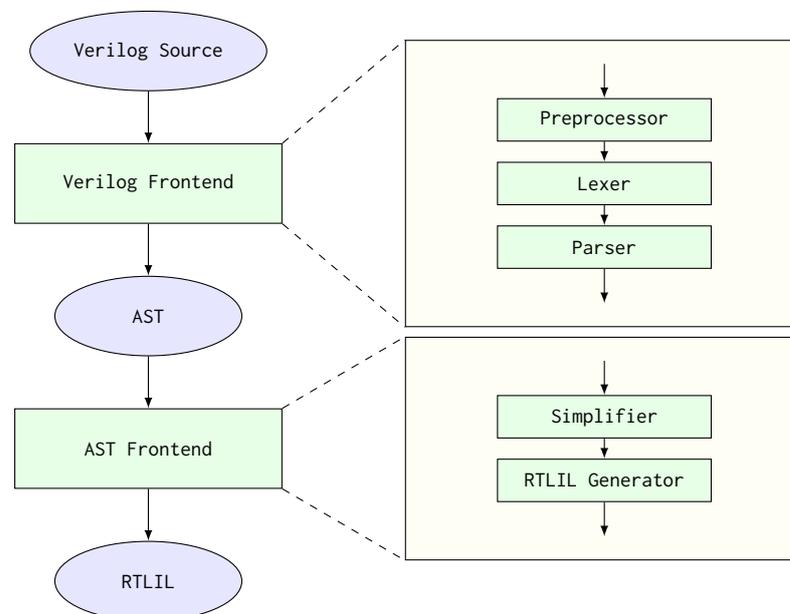
Figure 7.1: Simplified Verilog to RTLIL data flow

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* [17]. Its source code can be found in `frontends/verilog/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* [18]. Its source code can be found in `frontends/verilog/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 instantiation
AST_ARGUMENT	Port connection in cell instantiation
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 instantiation
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)

67

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 boolean value (equivalent 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 instantiation
`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 with 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 necessary (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 equivalent 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.

- A process may also contain memory writes. A RTLIL::MemWriteAction is created for each of them.

- 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:

```verilog
1  always @(posedge clock) begin
2        out1 = in1;
3        if (in2)
4              out1 = !out1;
5        out2 <= out1;
6        if (in3)
7              out2 <= out2;
8        if (in4)
9              if (in5)
10                   out3 <= in6;
11             else
12                   out3 <= in7;
13       out1 = out1 ^ out2;
14 end
```

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

```
1  cell $logic_not $logic_not$<input>:4$2
2    connect \A \in1
3    connect \Y $logic_not$<input>:4$2_Y
4  end
5  cell $xor $xor$<input>:13$3
6    connect \A $1\out1[0:0]
7    connect \B \out2
8    connect \Y $xor$<input>:13$3_Y
9  end
10 process $proc$<input>:1$1
11   assign $0\out3[0:0] \out3
12   assign $0\out2[0:0] $1\out1[0:0]
13   assign $0\out1[0:0] $xor$<input>:13$3_Y
14   switch \in2
15     case 1'1
16       assign $1\out1[0:0] $logic_not$<input>:4$2_Y
17     case
18       assign $1\out1[0:0] \in1
19   end
20   switch \in3
21     case 1'1
22       assign $0\out2[0:0] \out2
23     case
24   end
25   switch \in4
26     case 1'1
27       switch \in5
28         case 1'1
29           assign $0\out3[0:0] \in6
30         case
31           assign $0\out3[0:0] \in7
32       end
33     case
34   end
35   sync posedge \clock
36     update \out1 $0\out1[0:0]
```

70

```
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 continuously 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 object 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::SyncRules` that describe the output registers.

- `proc_dff`
  This pass replaces the `RTLIL::SyncRules` to d-type flip-flops (with asynchronous resets if necessary).

- `proc_dff`
  This pass replaces the `RTLIL::MemWriteActionss` with `$memwr` cells.

- `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 separate 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 processed 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_expr`
    - `opt_merge -nomux`
- Repeat until result is stable:
    - `opt_muxtree`
    - `opt_reduce`
    - `opt_merge`
    - `opt_rmdff`
    - `opt_clean`
    - `opt_expr`

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

8.1.1 The opt_expr 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_expr`.

The next two lines assume the value 0 for undef states. These two rules are only used if no other substitutions 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_expr` pass can replace 1-bit wide `$eq` and `$ne` cells with buffers or not-gates if one input is constant.

The `opt_expr` 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_merge 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 (`$mux` and `$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.

Note however that marking state registers with `\fsm_encoding` that are not suitable for FSM recoding can cause synthesis to fail or produce invalid results.

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 altered 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 [24].

8.3 Logic Optimization

Yosys can perform multi-level combinational logic optimization on gate-level netlists using the external program ABC [27]. 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 equivalent netlist utilizing 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/common/techmap.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.210). This can for example be useful 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 [25].

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 [27] 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.

Appendix A

Auxiliary Libraries

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

A.1 SHA1

The files in `libs/sha1/` provide a public domain SHA1 implementation written by Steve Reid, Bruce Guenter, and Volker Grabsch. 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 [29].

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 C. Wolf and based on the Ullmann Subgraph Isomorphism Algorithm [Ull76]. It is used by the `extract` pass (see `help extract` or Sec. C.63).

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 C. Wolf. It is used by the `sat` pass (see `help sat` or Sec. C.170).

Appendix B

Auxiliary 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-abc

This is a fork of ABC [27] with a small set of custom modifications that have not yet been accepted upstream. Not all versions of Yosys work with all versions of ABC. So Yosys comes with its own yosys-abc to avoid compatibility issues between the two.

Appendix C

Command Reference Manual

C.1 abc – use ABC for technology mapping

```
 1      abc [options] [selection]
 2
 3  This pass uses the ABC tool [1] for technology mapping of yosys's internal gate
 4  library to a target architecture.
 5
 6      -exe <command>
 7          use the specified command instead of "<yosys-bindir>/yosys-abc" to execute ABC.
 8          This can e.g. be used to call a specific version of ABC or a wrapper.
 9
10      -script <file>
11          use the specified ABC script file instead of the default script.
12
13          if <file> starts with a plus sign (+), then the rest of the filename
14          string is interpreted as the command string to be passed to ABC. The
15          leading plus sign is removed and all commas (,) in the string are
16          replaced with blanks before the string is passed to ABC.
17
18          if no -script parameter is given, the following scripts are used:
19
20          for -liberty/-genlib without -constr:
21            strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
22                &get -n; &dch -f; &nf {D}; &put
23
24          for -liberty/-genlib with -constr:
25            strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
26                &get -n; &dch -f; &nf {D}; &put; buffer; upsize {D};
27                dnsize {D}; stime -p
28
29          for -lut/-luts (only one LUT size):
30            strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
31                dch -f; if; mfs2; lutpack {S}
32
33          for -lut/-luts (different LUT sizes):
34            strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
35                dch -f; if; mfs2
```

```
36
37        for -sop:
38          strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
39              dch -f; cover {I} {P}
40
41        otherwise:
42          strash; &get -n; &fraig -x; &put; scorr; dc2; dretime; strash;
43              &get -n; &dch -f; &nf {D}; &put
44
45    -fast
46        use different default scripts that are slightly faster (at the cost
47        of output quality):
48
49        for -liberty/-genlib without -constr:
50          strash; dretime; map {D}
51
52        for -liberty/-genlib with -constr:
53          strash; dretime; map {D}; buffer; upsize {D}; dnsize {D};
54              stime -p
55
56        for -lut/-luts:
57          strash; dretime; if
58
59        for -sop:
60          strash; dretime; cover {I} {P}
61
62        otherwise:
63          strash; dretime; map
64
65    -liberty <file>
66        generate netlists for the specified cell library (using the liberty
67        file format).
68
69    -genlib <file>
70        generate netlists for the specified cell library (using the SIS Genlib
71        file format).
72
73    -constr <file>
74        pass this file with timing constraints to ABC.
75        use with -liberty/-genlib.
76
77        a constr file contains two lines:
78            set_driving_cell <cell_name>
79            set_load <floating_point_number>
80
81        the set_driving_cell statement defines which cell type is assumed to
82        drive the primary inputs and the set_load statement sets the load in
83        femtofarads for each primary output.
84
85    -D <picoseconds>
86        set delay target. the string {D} in the default scripts above is
87        replaced by this option when used, and an empty string otherwise.
88        this also replaces 'dretime' with 'dretime; retime -o {D}' in the
89        default scripts above.
```

```
 90
 91        -I <num>
 92            maximum number of SOP inputs.
 93            (replaces {I} in the default scripts above)
 94
 95        -P <num>
 96            maximum number of SOP products.
 97            (replaces {P} in the default scripts above)
 98
 99        -S <num>
100            maximum number of LUT inputs shared.
101            (replaces {S} in the default scripts above, default: -S 1)
102
103        -lut <width>
104            generate netlist using luts of (max) the specified width.
105
106        -lut <w1>:<w2>
107            generate netlist using luts of (max) the specified width <w2>. All
108            luts with width <= <w1> have constant cost. for luts larger than <w1>
109            the area cost doubles with each additional input bit. the delay cost
110            is still constant for all lut widths.
111
112        -luts <cost1>,<cost2>,<cost3>,<sizeN>:<cost4-N>,..
113            generate netlist using luts. Use the specified costs for luts with 1,
114            2, 3, .. inputs.
115
116        -sop
117            map to sum-of-product cells and inverters
118
119        -g type1,type2,...
120            Map to the specified list of gate types. Supported gates types are:
121                AND, NAND, OR, NOR, XOR, XNOR, ANDNOT, ORNOT, MUX,
122                NMUX, AOI3, OAI3, AOI4, OAI4.
123            (The NOT gate is always added to this list automatically.)
124
125            The following aliases can be used to reference common sets of gate
126            types:
127              simple: AND OR XOR MUX
128              cmos2:  NAND NOR
129              cmos3:  NAND NOR AOI3 OAI3
130              cmos4:  NAND NOR AOI3 OAI3 AOI4 OAI4
131              cmos:   NAND NOR AOI3 OAI3 AOI4 OAI4 NMUX MUX XOR XNOR
132              gates:  AND NAND OR NOR XOR XNOR ANDNOT ORNOT
133              aig:    AND NAND OR NOR ANDNOT ORNOT
134
135            The alias 'all' represent the full set of all gate types.
136
137            Prefix a gate type with a '-' to remove it from the list. For example
138            the arguments 'AND,OR,XOR' and 'simple,-MUX' are equivalent.
139
140            The default is 'all,-NMUX,-AOI3,-OAI3,-AOI4,-OAI4'.
141
142        -dff
143            also pass $_DFF_?_ and $_DFFE_??_ cells through ABC. modules with many
```

```
144         clock domains are automatically partitioned in clock domains and each
145         domain is passed through ABC independently.
146
147     -clk [!]<clock-signal-name>[,[!]<enable-signal-name>]
148         use only the specified clock domain. this is like -dff, but only FF
149         cells that belong to the specified clock domain are used.
150
151     -keepff
152         set the "keep" attribute on flip-flop output wires. (and thus preserve
153         them, for example for equivalence checking.)
154
155     -nocleanup
156         when this option is used, the temporary files created by this pass
157         are not removed. this is useful for debugging.
158
159     -showtmp
160         print the temp dir name in log. usually this is suppressed so that the
161         command output is identical across runs.
162
163     -markgroups
164         set a 'abcgroup' attribute on all objects created by ABC. The value of
165         this attribute is a unique integer for each ABC process started. This
166         is useful for debugging the partitioning of clock domains.
167
168     -dress
169         run the 'dress' command after all other ABC commands. This aims to
170         preserve naming by an equivalence check between the original and
171         post-ABC netlists (experimental).
172
173 When no target cell library is specified the Yosys standard cell library is
174 loaded into ABC before the ABC script is executed.
175
176 Note that this is a logic optimization pass within Yosys that is calling ABC
177 internally. This is not going to "run ABC on your design". It will instead run
178 ABC on logic snippets extracted from your design. You will not get any useful
179 output when passing an ABC script that writes a file. Instead write your full
180 design as BLIF file with write_blif and then load that into ABC externally if
181 you want to use ABC to convert your design into another format.
182
183 [1] http://www.eecs.berkeley.edu/~alanmi/abc/
```

C.2   abc9 – use ABC9 for technology mapping

```
1     abc9 [options] [selection]
2
3 This script pass performs a sequence of commands to facilitate the use of the
4 ABC tool [1] for technology mapping of the current design to a target FPGA
5 architecture. Only fully-selected modules are supported.
6
7     -run <from_label>:<to_label>
8         only run the commands between the labels (see below). an empty
9         from label is synonymous to 'begin', and empty to label is
```

```
10          synonymous to the end of the command list.
11
12      -exe <command>
13          use the specified command instead of "<yosys-bindir>/yosys-abc" to execute ABC.
14          This can e.g. be used to call a specific version of ABC or a wrapper.
15
16      -script <file>
17          use the specified ABC script file instead of the default script.
18
19          if <file> starts with a plus sign (+), then the rest of the filename
20          string is interpreted as the command string to be passed to ABC. The
21          leading plus sign is removed and all commas (,) in the string are
22          replaced with blanks before the string is passed to ABC.
23
24          if no -script parameter is given, the following scripts are used:
25            &scorr; &sweep; &dc2; &dch -f; &ps; &if {C} {W} {D} {R} -v; &mfs
26
27      -fast
28          use different default scripts that are slightly faster (at the cost
29          of output quality):
30            &if {C} {W} {D} {R} -v
31
32      -D <picoseconds>
33          set delay target. the string {D} in the default scripts above is
34          replaced by this option when used, and an empty string otherwise
35          (indicating best possible delay).
36
37      -lut <width>
38          generate netlist using luts of (max) the specified width.
39
40      -lut <w1>:<w2>
41          generate netlist using luts of (max) the specified width <w2>. All
42          luts with width <= <w1> have constant cost. for luts larger than <w1>
43          the area cost doubles with each additional input bit. the delay cost
44          is still constant for all lut widths.
45
46      -lut <file>
47          pass this file with lut library to ABC.
48
49      -luts <cost1>,<cost2>,<cost3>,<sizeN>:<cost4-N>,..
50          generate netlist using luts. Use the specified costs for luts with 1,
51          2, 3, .. inputs.
52
53      -maxlut <width>
54          when auto-generating the lut library, discard all luts equal to or
55          greater than this size (applicable when neither -lut nor -luts is
56          specified).
57
58      -dff
59          also pass $_DFF_[NP]_ cells through to ABC. modules with many clock
60          domains are supported and automatically partitioned by ABC.
61
62      -nocleanup
63          when this option is used, the temporary files created by this pass
```

```
64          are not removed. this is useful for debugging.
65
66      -showtmp
67          print the temp dir name in log. usually this is suppressed so that the
68          command output is identical across runs.
69
70      -box <file>
71          pass this file with box library to ABC.
72
73  Note that this is a logic optimization pass within Yosys that is calling ABC
74  internally. This is not going to "run ABC on your design". It will instead run
75  ABC on logic snippets extracted from your design. You will not get any useful
76  output when passing an ABC script that writes a file. Instead write your full
77  design as an XAIGER file with `write_xaiger' and then load that into ABC
78  externally if you want to use ABC to convert your design into another format.
79
80  [1] http://www.eecs.berkeley.edu/~alanmi/abc/
81
82
83      check:
84          abc9_ops -check [-dff]    (option if -dff)
85
86      map:
87          abc9_ops -prep_hier [-dff]    (option if -dff)
88          scc -specify -set_attr abc9_scc_id {}
89          abc9_ops -prep_bypass [-prep_dff]    (option if -dff)
90          design -stash $abc9
91          design -load $abc9_map
92          proc
93          wbflip
94          techmap -wb -map %$abc9 -map +/techmap.v A:abc9_flop
95          opt -nodffe -nosdff
96          abc9_ops -prep_dff_submod                                    (only if -dff)
97          setattr -set submod "$abc9_flop" t:$_DFF_?_ %ci* %co* t:$_DFF_?_ %d    (only if -dff)
98          submod                                                       (only if -dff)
99          setattr -mod -set whitebox 1 -set abc9_flop 1 -set abc9_box 1 *_$abc9_flop    (only if -dff)
100         foreach module in design
101             rename <module-name>_$abc9_flop _TECHMAP_REPLACE_        (only if -dff)
102         abc9_ops -prep_dff_unmap                                     (only if -dff)
103         design -copy-to $abc9 =*_$abc9_flop                          (only if -dff)
104         delete =*_$abc9_flop                                         (only if -dff)
105         design -stash $abc9_map
106         design -load $abc9
107         design -delete $abc9
108         techmap -wb -max_iter 1 -map %$abc9_map -map +/abc9_map.v [-D DFF]    (option if -dff)
109         design -delete $abc9_map
110
111     pre:
112         read_verilog -icells -lib -specify +/abc9_model.v
113         abc9_ops -break_scc -prep_delays -prep_xaiger [-dff]    (option for -dff)
114         abc9_ops -prep_lut <maxlut>    (skip if -lut or -luts)
115         abc9_ops -prep_box    (skip if -box)
116         design -stash $abc9
117         design -load $abc9_holes
```

90

```
118         techmap -wb -map %$abc9 -map +/techmap.v
119         opt -purge
120         aigmap
121         design -stash $abc9_holes
122         design -load $abc9
123         design -delete $abc9
124
125     exe:
126         aigmap
127         foreach module in selection
128             abc9_ops -write_lut <abc-temp-dir>/input.lut    (skip if '-lut' or '-luts')
129             abc9_ops -write_box <abc-temp-dir>/input.box     (skip if '-box')
130             write_xaiger -map <abc-temp-dir>/input.sym [-dff] <abc-temp-dir>/input.xaig
131             abc9_exe [options] -cwd <abc-temp-dir> -lut [<abc-temp-dir>/input.lut] -box [<abc-temp-dir>/inp
132             read_aiger -xaiger -wideports -module_name <module-name>$abc9 -map <abc-temp-dir>/input.sym <ab
133             abc9_ops -reintegrate [-dff]
134
135     unmap:
136         techmap -wb -map %$abc9_unmap -map +/abc9_unmap.v
137         design -delete $abc9_unmap
138         design -delete $abc9_holes
139         delete =*_$abc9_byp
140         setattr -mod -unset abc9_box_id
```

C.3   abc9_exe – use ABC9 for technology mapping

```
1       abc9_exe [options]
2
3
4  This pass uses the ABC tool [1] for technology mapping of the top module
5  (according to the (* top *) attribute or if only one module is currently
6  selected) to a target FPGA architecture.
7
8      -exe <command>
9          use the specified command instead of "<yosys-bindir>/yosys-abc" to execute ABC.
10         This can e.g. be used to call a specific version of ABC or a wrapper.
11
12     -script <file>
13         use the specified ABC script file instead of the default script.
14
15         if <file> starts with a plus sign (+), then the rest of the filename
16         string is interpreted as the command string to be passed to ABC. The
17         leading plus sign is removed and all commas (,) in the string are
18         replaced with blanks before the string is passed to ABC.
19
20         if no -script parameter is given, the following scripts are used:
21           &scorr; &sweep; &dc2; &dch -f; &ps; &if {C} {W} {D} {R} -v; &mfs
22
23     -fast
24         use different default scripts that are slightly faster (at the cost
25         of output quality):
26           &if {C} {W} {D} {R} -v
```

```
27
28      -D <picoseconds>
29          set delay target. the string {D} in the default scripts above is
30          replaced by this option when used, and an empty string otherwise
31          (indicating best possible delay).
32
33      -lut <width>
34          generate netlist using luts of (max) the specified width.
35
36      -lut <w1>:<w2>
37          generate netlist using luts of (max) the specified width <w2>. All
38          luts with width <= <w1> have constant cost. for luts larger than <w1>
39          the area cost doubles with each additional input bit. the delay cost
40          is still constant for all lut widths.
41
42      -lut <file>
43          pass this file with lut library to ABC.
44
45      -luts <cost1>,<cost2>,<cost3>,<sizeN>:<cost4-N>,..
46          generate netlist using luts. Use the specified costs for luts with 1,
47          2, 3, .. inputs.
48
49      -showtmp
50          print the temp dir name in log. usually this is suppressed so that the
51          command output is identical across runs.
52
53      -box <file>
54          pass this file with box library to ABC.
55
56      -cwd <dir>
57          use this as the current working directory, inside which the 'input.xaig'
58          file is expected. temporary files will be created in this directory, and
59          the mapped result will be written to 'output.aig'.
60
61  Note that this is a logic optimization pass within Yosys that is calling ABC
62  internally. This is not going to "run ABC on your design". It will instead run
63  ABC on logic snippets extracted from your design. You will not get any useful
64  output when passing an ABC script that writes a file. Instead write your full
65  design as BLIF file with write_blif and then load that into ABC externally if
66  you want to use ABC to convert your design into another format.
67
68  [1] http://www.eecs.berkeley.edu/~alanmi/abc/
```

C.4   abc9_ops – helper functions for ABC9

```
1       abc9_ops [options] [selection]
2
3   This pass contains a set of supporting operations for use during ABC technology
4   mapping, and is expected to be called in conjunction with other operations from
5   the `abc9' script pass. Only fully-selected modules are supported.
6
7       -check
```

```
 8            check that the design is valid, e.g. (* abc9_box_id *) values are
 9            unique, (* abc9_carry *) is only given for one input/output port, etc.
10
11     -prep_hier
12            derive all used (* abc9_box *) or (* abc9_flop *) (if -dff option)
13            whitebox modules. with (* abc9_flop *) modules, only those containing
14            $dff/$_DFF_[NP]_ cells with zero initial state -- due to an ABC
15            limitation -- will be derived.
16
17     -prep_bypass
18            create techmap rules in the '$abc9_map' and '$abc9_unmap' designs for
19            bypassing sequential (* abc9_box *) modules using a combinatorial box
20            (named *_$abc9_byp). bypassing is necessary if sequential elements (e.g.
21            $dff, $mem, etc.) are discovered inside so that any combinatorial paths
22            will be correctly captured. this bypass box will only contain ports that
23            are referenced by a simple path declaration ($specify2 cell) inside a
24            specify block.
25
26     -prep_dff
27            select all (* abc9_flop *) modules instantiated in the design and store
28            in the named selection '$abc9_flops'.
29
30     -prep_dff_submod
31            within (* abc9_flop *) modules, rewrite all edge-sensitive path
32            declarations and $setup() timing checks ($specify3 and $specrule cells)
33            that share a 'DST' port with the $_DFF_[NP]_.Q port from this 'Q' port
34            to the DFF's 'D' port. this is to prepare such specify cells to be moved
35            into the flop box.
36
37     -prep_dff_unmap
38            populate the '$abc9_unmap' design with techmap rules for mapping
39            *_$abc9_flop cells back into their derived cell types (where the rules
40            created by -prep_hier will then map back to the original cell with
41            parameters).
42
43     -prep_delays
44            insert `$__ABC9_DELAY' blackbox cells into the design to account for
45            certain required times.
46
47     -break_scc
48            for an arbitrarily chosen cell in each unique SCC of each selected
49            module (tagged with an (* abc9_scc_id = <int> *) attribute) interrupt
50            all wires driven by this cell's outputs with a temporary
51            $__ABC9_SCC_BREAKER cell to break the SCC.
52
53     -prep_xaiger
54            prepare the design for XAIGER output. this includes computing the
55            topological ordering of ABC9 boxes, as well as preparing the
56            '$abc9_holes' design that contains the logic behaviour of ABC9
57            whiteboxes.
58
59     -dff
60            consider flop cells (those instantiating modules marked with
61            (* abc9_flop *)) during -prep_{delays,xaiger,box}.
```

```
62
63      -prep_lut <maxlut>
64          pre-compute the lut library by analysing all modules marked with
65          (* abc9_lut=<area> *).
66
67      -write_lut <dst>
68          write the pre-computed lut library to <dst>.
69
70      -prep_box
71          pre-compute the box library by analysing all modules marked with
72          (* abc9_box *).
73
74      -write_box <dst>
75          write the pre-computed box library to <dst>.
76
77      -reintegrate
78          for each selected module, re-intergrate the module '<module-name>$abc9'
79          by first recovering ABC9 boxes, and then stitching in the remaining
80          primary inputs and outputs.
```

C.5 add − add objects to the design

```
1       add <command> [selection]
2
3   This command adds objects to the design. It operates on all fully selected
4   modules. So e.g. 'add -wire foo' will add a wire foo to all selected modules.
5
6
7       add {-wire|-input|-inout|-output} <name> <width> [selection]
8
9   Add a wire (input, inout, output port) with the given name and width. The
10  command will fail if the object exists already and has different properties
11  than the object to be created.
12
13
14      add -global_input <name> <width> [selection]
15
16  Like 'add -input', but also connect the signal between instances of the
17  selected modules.
18
19
20      add {-assert|-assume|-live|-fair|-cover} <name1> [-if <name2>]
21
22  Add an $assert, $assume, etc. cell connected to a wire named name1, with its
23  enable signal optionally connected to a wire named name2 (default: 1'b1).
24
25
26      add -mod <name[s]>
27
28  Add module[s] with the specified name[s].
```

C.6 aigmap – map logic to and-inverter-graph circuit

```
1       aigmap [options] [selection]
2
3   Replace all logic cells with circuits made of only $_AND_ and
4   $_NOT_ cells.
5
6       -nand
7           Enable creation of $_NAND_ cells
8
9       -select
10          Overwrite replaced cells in the current selection with new $_AND_,
11          $_NOT_, and $_NAND_, cells
```

C.7 alumacc – extract ALU and MACC cells

```
1       alumacc [selection]
2
3   This pass translates arithmetic operations like $add, $mul, $lt, etc. to $alu
4   and $macc cells.
```

C.8 anlogic_eqn – Anlogic: Calculate equations for luts

```
1       anlogic_eqn [selection]
2
3   Calculate equations for luts since bitstream generator depends on it.
```

C.9 anlogic_fixcarry – Anlogic: fix carry chain

```
1       anlogic_fixcarry [options] [selection]
2
3   Add Anlogic adders to fix carry chain if needed.
```

C.10 assertpmux – adds asserts for parallel muxes

```
1       assertpmux [options] [selection]
2
3   This command adds asserts to the design that assert that all parallel muxes
4   ($pmux cells) have a maximum of one of their inputs enable at any time.
5
6       -noinit
7           do not enforce the pmux condition during the init state
8
9       -always
```

```
10          usually the $pmux condition is only checked when the $pmux output
11          is used by the mux tree it drives. this option will deactivate this
12          additional constraint and check the $pmux condition always.
```

C.11   async2sync – convert async FF inputs to sync circuits

```
1      async2sync [options] [selection]
2
3  This command replaces async FF inputs with sync circuits emulating the same
4  behavior for when the async signals are actually synchronized to the clock.
5
6  This pass assumes negative hold time for the async FF inputs. For example when
7  a reset deasserts with the clock edge, then the FF output will still drive the
8  reset value in the next cycle regardless of the data-in value at the time of
9  the clock edge.
```

C.12   attrmap – renaming attributes

```
1      attrmap [options] [selection]
2
3  This command renames attributes and/or maps key/value pairs to
4  other key/value pairs.
5
6      -tocase <name>
7          Match attribute names case-insensitively and set it to the specified
8          name.
9
10     -rename <old_name> <new_name>
11         Rename attributes as specified
12
13     -map <old_name>=<old_value> <new_name>=<new_value>
14         Map key/value pairs as indicated.
15
16     -imap <old_name>=<old_value> <new_name>=<new_value>
17         Like -map, but use case-insensitive match for <old_value> when
18         it is a string value.
19
20     -remove <name>=<value>
21         Remove attributes matching this pattern.
22
23     -modattr
24         Operate on module attributes instead of attributes on wires and cells.
25
26  For example, mapping Xilinx-style "keep" attributes to Yosys-style:
27
28     attrmap -tocase keep -imap keep="true" keep=1 \
29             -imap keep="false" keep=0 -remove keep=0
```

C.13   attrmvcp – move or copy attributes from wires to driving cells

```
1      attrmvcp [options] [selection]
2
3  Move or copy attributes on wires to the cells driving them.
4
5      -copy
6          By default, attributes are moved. This will only add
7          the attribute to the cell, without removing it from
8          the wire.
9
10     -purge
11         If no selected cell consumes the attribute, then it is
12         left on the wire by default. This option will cause the
13         attribute to be removed from the wire, even if no selected
14         cell takes it.
15
16     -driven
17         By default, attriburtes are moved to the cell driving the
18         wire. With this option set it will be moved to the cell
19         driven by the wire instead.
20
21     -attr <attrname>
22         Move or copy this attribute. This option can be used
23         multiple times.
```

C.14   autoname – automatically assign names to objects

```
1      autoname [selection]
2
3  Assign auto-generated public names to objects with private names (the ones
4  with $-prefix).
```

C.15   blackbox – convert modules into blackbox modules

```
1      blackbox [options] [selection]
2
3  Convert modules into blackbox modules (remove contents and set the blackbox
4  module attribute).
```

C.16   bmuxmap – transform $bmux cells to trees of $mux cells

```
1      bmuxmap [selection]
2
3  This pass transforms $bmux cells to trees of $mux cells.
```

C.17   bugpoint – minimize testcases

```
 1      bugpoint [options] [-script <filename> | -command "<command>"]
 2
 3  This command minimizes the current design that is known to crash Yosys with the
 4  given script into a smaller testcase. It does this by removing an arbitrary part
 5  of the design and recursively invokes a new Yosys process with this modified
 6  design and the same script, repeating these steps while it can find a smaller
 7  design that still causes a crash. Once this command finishes, it replaces the
 8  current design with the smallest testcase it was able to produce.
 9  In order to save the reduced testcase you must write this out to a file with
10  another command after `bugpoint` like `write_rtlil` or `write_verilog`.
11
12      -script <filename> | -command "<command>"
13          use this script file or command to crash Yosys. required.
14
15      -yosys <filename>
16          use this Yosys binary. if not specified, `yosys` is used.
17
18      -grep "<string>"
19          only consider crashes that place this string in the log file.
20
21      -fast
22          run `proc_clean; clean -purge` after each minimization step. converges
23          faster, but produces larger testcases, and may fail to produce any
24          testcase at all if the crash is related to dangling wires.
25
26      -clean
27          run `proc_clean; clean -purge` before checking testcase and after
28          finishing. produces smaller and more useful testcases, but may fail to
29          produce any testcase at all if the crash is related to dangling wires.
30
31  It is possible to constrain which parts of the design will be considered for
32  removal. Unless one or more of the following options are specified, all parts
33  will be considered.
34
35      -modules
36          try to remove modules. modules with a (* bugpoint_keep *) attribute
37          will be skipped.
38
39      -ports
40          try to remove module ports. ports with a (* bugpoint_keep *) attribute
41          will be skipped (useful for clocks, resets, etc.)
42
43      -cells
44          try to remove cells. cells with a (* bugpoint_keep *) attribute will
45          be skipped.
46
47      -connections
48          try to reconnect ports to 'x.
49
50      -processes
51          try to remove processes. processes with a (* bugpoint_keep *) attribute
52          will be skipped.
```

```
53
54      -assigns
55          try to remove process assigns from cases.
56
57      -updates
58          try to remove process updates from syncs.
59
60      -runner "<prefix>"
61          child process wrapping command, e.g., "timeout 30", or valgrind.
```

C.18   cd – a shortcut for 'select -module <name>'

```
1       cd <modname>
2
3  This is just a shortcut for 'select -module <modname>'.
4
5
6       cd <cellname>
7
8  When no module with the specified name is found, but there is a cell
9  with the specified name in the current module, then this is equivalent
10 to 'cd <celltype>'.
11
12      cd ..
13
14 Remove trailing substrings that start with '.' in current module name until
15 the name of a module in the current design is generated, then switch to that
16 module. Otherwise clear the current selection.
17
18      cd
19
20 This is just a shortcut for 'select -clear'.
```

C.19   check – check for obvious problems in the design

```
1       check [options] [selection]
2
3  This pass identifies the following problems in the current design:
4
5    - combinatorial loops
6    - two or more conflicting drivers for one wire
7    - used wires that do not have a driver
8
9  Options:
10
11      -noinit
12          also check for wires which have the 'init' attribute set
13
14      -initdrv
15          also check for wires that have the 'init' attribute set and are not
```

```
16          driven by an FF cell type
17
18      -mapped
19          also check for internal cells that have not been mapped to cells of the
20          target architecture
21
22      -allow-tbuf
23          modify the -mapped behavior to still allow $_TBUF_ cells
24
25      -assert
26          produce a runtime error if any problems are found in the current design
```

C.20   chformal – change formal constraints of the design

```
1       chformal [types] [mode] [options] [selection]
2
3   Make changes to the formal constraints of the design. The [types] options
4   the type of constraint to operate on. If none of the following options are
5   given, the command will operate on all constraint types:
6
7       -assert        $assert cells, representing assert(...) constraints
8       -assume        $assume cells, representing assume(...) constraints
9       -live          $live cells, representing assert(s_eventually ...)
10      -fair          $fair cells, representing assume(s_eventually ...)
11      -cover         $cover cells, representing cover() statements
12
13  Exactly one of the following modes must be specified:
14
15      -remove
16          remove the cells and thus constraints from the design
17
18      -early
19          bypass FFs that only delay the activation of a constraint
20
21      -delay <N>
22          delay activation of the constraint by <N> clock cycles
23
24      -skip <N>
25          ignore activation of the constraint in the first <N> clock cycles
26
27      -assert2assume
28      -assume2assert
29      -live2fair
30      -fair2live
31          change the roles of cells as indicated. these options can be combined
```

C.21   chparam – re-evaluate modules with new parameters

```
1       chparam [ -set name value ]... [selection]
2
```

```
3   Re-evaluate the selected modules with new parameters. String values must be
4   passed in double quotes (").
5
6
7       chparam -list [selection]
8
9   List the available parameters of the selected modules.
```

C.22   chtype – change type of cells in the design

```
1       chtype [options] [selection]
2
3   Change the types of cells in the design.
4
5       -set <type>
6           set the cell type to the given type
7
8       -map <old_type> <new_type>
9           change cells types that match <old_type> to <new_type>
```

C.23   clean – remove unused cells and wires

```
1       clean [options] [selection]
2
3   This is identical to 'opt_clean', but less verbose.
4
5   When commands are separated using the ';;' token, this command will be executed
6   between the commands.
7
8   When commands are separated using the ';;;' token, this command will be executed
9   in -purge mode between the commands.
```

C.24   clean_zerowidth – clean zero-width connections from the design

```
1       clean_zerowidth [selection]
2
3   Fixes the selected cells and processes to contain no zero-width connections.
4   Depending on the cell type, this may be implemented by removing the connection,
5   widening it to 1-bit, or removing the cell altogether.
```

C.25   clk2fflogic – convert clocked FFs to generic $ff cells

```
1      clk2fflogic [options] [selection]
2
3   This command replaces clocked flip-flops with generic $ff cells that use the
4   implicit global clock. This is useful for formal verification of designs with
5   multiple clocks.
6
7   This pass assumes negative hold time for the async FF inputs. For example when
8   a reset deasserts with the clock edge, then the FF output will still drive the
9   reset value in the next cycle regardless of the data-in value at the time of
10  the clock edge.
```

C.26   clkbufmap – insert clock buffers on clock networks

```
1       clkbufmap [options] [selection]
2
3   Inserts clock buffers between nets connected to clock inputs and their drivers.
4
5   In the absence of any selection, all wires without the 'clkbuf_inhibit'
6   attribute will be considered for clock buffer insertion.
7   Alternatively, to consider all wires without the 'buffer_type' attribute set to
8   'none' or 'bufr' one would specify:
9     'w:* a:buffer_type=none a:buffer_type=bufr %u %d'
10  as the selection.
11
12      -buf <celltype> <portname_out>:<portname_in>
13          Specifies the cell type to use for the clock buffers
14          and its port names.  The first port will be connected to
15          the clock network sinks, and the second will be connected
16          to the actual clock source.
17
18      -inpad <celltype> <portname_out>:<portname_in>
19          If specified, a PAD cell of the given type is inserted on
20          clock nets that are also top module's inputs (in addition
21          to the clock buffer, if any).
22
23  At least one of -buf or -inpad should be specified.
```

C.27   connect – create or remove connections

```
1       connect [-nomap] [-nounset] -set <lhs-expr> <rhs-expr>
2
3   Create a connection. This is equivalent to adding the statement 'assign
4   <lhs-expr> = <rhs-expr>;' to the Verilog input. Per default, all existing
5   drivers for <lhs-expr> are unconnected. This can be overwritten by using
6   the -nounset option.
7
8
9       connect [-nomap] -unset <expr>
10
```

```
11   Unconnect all existing drivers for the specified expression.
12
13
14       connect [-nomap] [-assert] -port <cell> <port> <expr>
15
16   Connect the specified cell port to the specified cell port.
17
18
19   Per default signal alias names are resolved and all signal names are mapped
20   the the signal name of the primary driver. Using the -nomap option deactivates
21   this behavior.
22
23   The connect command operates in one module only. Either only one module must
24   be selected or an active module must be set using the 'cd' command.
25
26   The -assert option verifies that the connection already exists, instead of
27   making it.
28
29   This command does not operate on module with processes.
```

C.28   connect_rpc – connect to RPC frontend

```
 1       connect_rpc -exec <command> [args...]
 2       connect_rpc -path <path>
 3
 4   Load modules using an out-of-process frontend.
 5
 6       -exec <command> [args...]
 7           run <command> with arguments [args...]. send requests on stdin, read
 8           responses from stdout.
 9
10       -path <path>
11           connect to Unix domain socket at <path>. (Unix)
12           connect to bidirectional byte-type named pipe at <path>. (Windows)
13
14   A simple JSON-based, newline-delimited protocol is used for communicating with
15   the frontend. Yosys requests data from the frontend by sending exactly 1 line
16   of JSON. Frontend responds with data or error message by replying with exactly
17   1 line of JSON as well.
18
19       -> {"method": "modules"}
20       <- {"modules": ["<module-name>", ...]}
21       <- {"error": "<error-message>"}
22           request for the list of modules that can be derived by this frontend.
23           the 'hierarchy' command will call back into this frontend if a cell
24           with type <module-name> is instantiated in the design.
25
26       -> {"method": "derive", "module": "<module-name>", "parameters": {
27           "<param-name>": {"type": "[unsigned|signed|string|real]",
28                            "value": "<param-value>"}, ...}}
29       <- {"frontend": "[rtlil|verilog|...]","source": "<source>"}}
30       <- {"error": "<error-message>"}
```

```
31      request for the module <module-name> to be derived for a specific set of
32      parameters. <param-name> starts with \ for named parameters, and with $
33      for unnamed parameters, which are numbered starting at 1.<param-value>
34      for integer parameters is always specified as a binary string of
35      unlimited precision. the <source> returned by the frontend is
36      hygienically parsedby a built-in Yosys <frontend>, allowing the RPC
37      frontend to return anyconvenient representation of the module. the
38      derived module is cached,so the response should be the same whenever the
39      same set of parameters is provided.
```

C.29   connwrappers – match width of input-output port pairs

```
1      connwrappers [options] [selection]
2
3  Wrappers are used in coarse-grain synthesis to wrap cells with smaller ports
4  in wrapper cells with a (larger) constant port size. I.e. the upper bits
5  of the wrapper output are signed/unsigned bit extended. This command uses this
6  knowledge to rewire the inputs of the driven cells to match the output of
7  the driving cell.
8
9      -signed <cell_type> <port_name> <width_param>
10     -unsigned <cell_type> <port_name> <width_param>
11         consider the specified signed/unsigned wrapper output
12
13     -port <cell_type> <port_name> <width_param> <sign_param>
14         use the specified parameter to decide if signed or unsigned
15
16 The options -signed, -unsigned, and -port can be specified multiple times.
```

C.30   coolrunner2_fixup – insert necessary buffer cells for CoolRunner-II architecture

```
1      coolrunner2_fixup [options] [selection]
2
3  Insert necessary buffer cells for CoolRunner-II architecture.
```

C.31   coolrunner2_sop – break $sop cells into ANDTERM/ORTERM cells

```
1      coolrunner2_sop [options] [selection]
2
3  Break $sop cells into ANDTERM/ORTERM cells.
```

C.32   copy – copy modules in the design

```
1      copy old_name new_name
2
3   Copy the specified module. Note that selection patterns are not supported
4   by this command.
```

C.33   cover – print code coverage counters

```
1      cover [options] [pattern]
2
3   Print the code coverage counters collected using the cover() macro in the Yosys
4   C++ code. This is useful to figure out what parts of Yosys are utilized by a
5   test bench.
6
7      -q
8          Do not print output to the normal destination (console and/or log file)
9
10     -o file
11         Write output to this file, truncate if exists.
12
13     -a file
14         Write output to this file, append if exists.
15
16     -d dir
17         Write output to a newly created file in the specified directory.
18
19  When one or more pattern (shell wildcards) are specified, then only counters
20  matching at least one pattern are printed.
21
22
23  It is also possible to instruct Yosys to print the coverage counters on program
24  exit to a file using environment variables:
25
26     YOSYS_COVER_DIR="{dir-name}" yosys {args}
27
28         This will create a file (with an auto-generated name) in this
29         directory and write the coverage counters to it.
30
31     YOSYS_COVER_FILE="{file-name}" yosys {args}
32
33         This will append the coverage counters to the specified file.
34
35
36  Hint: Use the following AWK command to consolidate Yosys coverage files:
37
38     gawk '{ p[$3] = $1; c[$3] += $2; } END { for (i in p)
39       printf "%-60s %10d %s\n", p[i], c[i], i; }' {files} | sort -k3
40
41
42  Coverage counters are only available in Yosys for Linux.
```

C.34 cutpoint – adds formal cut points to the design

```
1    cutpoint [options] [selection]
2
3  This command adds formal cut points to the design.
4
5    -undef
6        set cupoint nets to undef (x). the default behavior is to create a
7        $anyseq cell and drive the cutpoint net from that
```

C.35 debug – run command with debug log messages enabled

```
1    debug cmd
2
3  Execute the specified command with debug log messages enabled
```

C.36 delete – delete objects in the design

```
1    delete [selection]
2
3  Deletes the selected objects. This will also remove entire modules, if the
4  whole module is selected.
5
6
7    delete {-input|-output|-port} [selection]
8
9  Does not delete any object but removes the input and/or output flag on the
10 selected wires, thus 'deleting' module ports.
```

C.37 deminout – demote inout ports to input or output

```
1    deminout [options] [selection]
2
3  "Demote" inout ports to input or output ports, if possible.
```

C.38 demuxmap – transform $demux cells to $eq + $mux cells

```
1    demuxmap [selection]
2
3  This pass transforms $demux cells to a bunch of equality comparisons.
```

C.39   design – save, restore and reset current design

```
 1       design -reset
 2
 3   Clear the current design.
 4
 5
 6       design -save <name>
 7
 8   Save the current design under the given name.
 9
10
11       design -stash <name>
12
13   Save the current design under the given name and then clear the current design.
14
15
16       design -push
17
18   Push the current design to the stack and then clear the current design.
19
20
21       design -push-copy
22
23   Push the current design to the stack without clearing the current design.
24
25
26       design -pop
27
28   Reset the current design and pop the last design from the stack.
29
30
31       design -load <name>
32
33   Reset the current design and load the design previously saved under the given
34   name.
35
36
37       design -copy-from <name> [-as <new_mod_name>] <selection>
38
39   Copy modules from the specified design into the current one. The selection is
40   evaluated in the other design.
41
42
43       design -copy-to <name> [-as <new_mod_name>] [selection]
44
45   Copy modules from the current design into the specified one.
46
47
48       design -import <name> [-as <new_top_name>] [selection]
49
50   Import the specified design into the current design. The source design must
51   either have a selected top module or the selection must contain exactly one
52   module that is then used as top module for this command.
```

```
53
54
55       design -reset-vlog
56
57  The Verilog front-end remembers defined macros and top-level declarations
58  between calls to 'read_verilog'. This command resets this memory.
59
60       design -delete <name>
61
62  Delete the design previously saved under the given name.
```

C.40   dffinit – set INIT param on FF cells

```
1        dffinit [options] [selection]
2
3   This pass sets an FF cell parameter to the the initial value of the net it
4   drives. (This is primarily used in FPGA flows.)
5
6       -ff <cell_name> <output_port> <init_param>
7           operate on the specified cell type. this option can be used
8           multiple times.
9
10      -highlow
11          use the string values "high" and "low" to represent a single-bit
12          initial value of 1 or 0. (multi-bit values are not supported in this
13          mode.)
14
15      -strinit <string for high> <string for low>
16          use string values in the command line to represent a single-bit
17          initial value of 1 or 0. (multi-bit values are not supported in this
18          mode.)
19
20      -noreinit
21          fail if the FF cell has already a defined initial value set in other
22          passes and the initial value of the net it drives is not equal to
23          the already defined initial value.
```

C.41   dfflegalize – convert FFs to types supported by the target

```
1        dfflegalize [options] [selection]
2
3   Converts FFs to types supported by the target.
4
5       -cell <cell_type_pattern> <init_values>
6           specifies a supported group of FF cells.  <cell_type_pattern>
7           is a yosys internal fine cell name, where ? characters can be
8           as a wildcard matching any character.  <init_values> specifies
9           which initialization values these FF cells can support, and can
10          be one of:
11
```

```
          - x (no init value supported)
          - 0
          - 1
          - r (init value has to match reset value, only for some FF types)
          - 01 (both 0 and 1 supported).

      -mince <num>
          specifies a minimum number of FFs that should be using any given
          clock enable signal.  If a clock enable signal doesn't meet this
          threshold, it is unmapped into soft logic.

      -minsrst <num>
          specifies a minimum number of FFs that should be using any given
          sync set/reset signal.  If a sync set/reset signal doesn't meet this
          threshold, it is unmapped into soft logic.

The following cells are supported by this pass (ie. will be ingested,
and can be specified as allowed targets):

- $_DFF_[NP]_
- $_DFFE_[NP][NP]_
- $_DFF_[NP][NP][01]_
- $_DFFE_[NP][NP][01][NP]_
- $_ALDFF_[NP][NP]_
- $_ALDFFE_[NP][NP][NP]_
- $_DFFSR_[NP][NP][NP]_
- $_DFFSRE_[NP][NP][NP][NP]_
- $_SDFF_[NP][NP][01]_
- $_SDFFE_[NP][NP][01][NP]_
- $_SDFFCE_[NP][NP][01][NP]_
- $_SR_[NP][NP]_
- $_DLATCH_[NP]_
- $_DLATCH_[NP][NP][01]_
- $_DLATCHSR_[NP][NP][NP]_

The following transformations are performed by this pass:

- upconversion from a less capable cell to a more capable cell, if the less
  capable cell is not supported (eg. dff -> dffe, or adff -> dffsr)
- unmapping FFs with clock enable (due to unsupported cell type or -mince)
- unmapping FFs with sync reset (due to unsupported cell type or -minsrst)
- adding inverters on the control pins (due to unsupported polarity)
- adding inverters on the D and Q pins and inverting the init/reset values
  (due to unsupported init or reset value)
- converting sr into adlatch (by tying D to 1 and using E as set input)
- emulating unsupported dffsr cell by adff + adff + sr + mux
- emulating unsupported dlatchsr cell by adlatch + adlatch + sr + mux
- emulating adff when the (reset, init) value combination is unsupported by
  dff + adff + dlatch + mux
- emulating adlatch when the (reset, init) value combination is unsupported by
- dlatch + adlatch + dlatch + mux
If the pass is unable to realize a given cell type (eg. adff when only plain dff
is available), an error is raised.
```

C.42   dfflibmap – technology mapping of flip-flops

```
 1      dfflibmap [-prepare] [-map-only] [-info] -liberty <file> [selection]
 2
 3   Map internal flip-flop cells to the flip-flop cells in the technology
 4   library specified in the given liberty file.
 5
 6   This pass may add inverters as needed. Therefore it is recommended to
 7   first run this pass and then map the logic paths to the target technology.
 8
 9   When called with -prepare, this command will convert the internal FF cells
10   to the internal cell types that best match the cells found in the given
11   liberty file, but won't actually map them to the target cells.
12
13   When called with -map-only, this command will only map internal cell
14   types that are already of exactly the right type to match the target
15   cells, leaving remaining internal cells untouched.
16
17   When called with -info, this command will only print the target cell
18   list, along with their associated internal cell types, and the arguments
19   that would be passed to the dfflegalize pass.  The design will not be
20   changed.
```

C.43   dffunmap – unmap clock enable and synchronous reset from FFs

```
 1      dffunmap [options] [selection]
 2
 3   This pass transforms FF types with clock enable and/or synchronous reset into
 4   their base type (with neither clock enable nor sync reset) by emulating the
 5   clock enable and synchronous reset with multiplexers on the cell input.
 6
 7      -ce-only
 8          unmap only clock enables, leave synchronous resets alone.
 9
10      -srst-only
11          unmap only synchronous resets, leave clock enables alone.
```

C.44   dump – print parts of the design in RTLIL format

```
 1      dump [options] [selection]
 2
 3   Write the selected parts of the design to the console or specified file in
 4   RTLIL format.
 5
 6      -m
 7          also dump the module headers, even if only parts of a single
 8          module is selected
 9
```

```
10      -n
11          only dump the module headers if the entire module is selected
12
13      -o <filename>
14          write to the specified file.
15
16      -a <filename>
17          like -outfile but append instead of overwrite
```

C.45   echo – turning echoing back of commands on and off

```
1      echo on
2
3  Print all commands to log before executing them.
4
5
6      echo off
7
8  Do not print all commands to log before executing them. (default)
```

C.46   ecp5_gsr – ECP5: handle GSR

```
1      ecp5_gsr [options] [selection]
2
3  Trim active low async resets connected to GSR and resolve GSR parameter,
4  if a GSR or SGSR primitive is used in the design.
5
6  If any cell has the GSR parameter set to "AUTO", this will be resolved
7  to "ENABLED" if a GSR primitive is present and the (* nogsr *) attribute
8  is not set, otherwise it will be resolved to "DISABLED".
```

C.47   edgetypes – list all types of edges in selection

```
1      edgetypes [options] [selection]
2
3  This command lists all unique types of 'edges' found in the selection. An 'edge'
4  is a 4-tuple of source and sink cell type and port name.
```

C.48   efinix_fixcarry – Efinix: fix carry chain

```
1      efinix_fixcarry [options] [selection]
2
3  Add Efinix adders to fix carry chain if needed.
```

C.49   equiv_add – add a $equiv cell

```
1      equiv_add [-try] gold_sig gate_sig
2
3  This command adds an $equiv cell for the specified signals.
4
5
6      equiv_add [-try] -cell gold_cell gate_cell
7
8  This command adds $equiv cells for the ports of the specified cells.
```

C.50   equiv_induct – proving $equiv cells using temporal induction

```
1      equiv_induct [options] [selection]
2
3  Uses a version of temporal induction to prove $equiv cells.
4
5  Only selected $equiv cells are proven and only selected cells are used to
6  perform the proof.
7
8      -undef
9          enable modelling of undef states
10
11     -seq <N>
12         the max. number of time steps to be considered (default = 4)
13
14 This command is very effective in proving complex sequential circuits, when
15 the internal state of the circuit quickly propagates to $equiv cells.
16
17 However, this command uses a weak definition of 'equivalence': This command
18 proves that the two circuits will not diverge after they produce equal
19 outputs (observable points via $equiv) for at least <N> cycles (the <N>
20 specified via -seq).
21
22 Combined with simulation this is very powerful because simulation can give
23 you confidence that the circuits start out synced for at least <N> cycles
24 after reset.
```

C.51   equiv_make – prepare a circuit for equivalence checking

```
1      equiv_make [options] gold_module gate_module equiv_module
2
3  This creates a module annotated with $equiv cells from two presumably
4  equivalent modules. Use commands such as 'equiv_simple' and 'equiv_status'
5  to work with the created equivalent checking module.
6
7      -inames
8          Also match cells and wires with $... names.
```

```
 9
10      -blacklist <file>
11          Do not match cells or signals that match the names in the file.
12
13      -encfile <file>
14          Match FSM encodings using the description from the file.
15          See 'help fsm_recode' for details.
16
17  Note: The circuit created by this command is not a miter (with something like
18  a trigger output), but instead uses $equiv cells to encode the equivalence
19  checking problem. Use 'miter -equiv' if you want to create a miter circuit.
```

C.52   equiv_mark – mark equivalence checking regions

```
1      equiv_mark [options] [selection]
2
3  This command marks the regions in an equivalence checking module. Region 0 is
4  the proven part of the circuit. Regions with higher numbers are connected
5  unproven subcricuits. The integer attribute 'equiv_region' is set on all
6  wires and cells.
```

C.53   equiv_miter – extract miter from equiv circuit

```
1      equiv_miter [options] miter_module [selection]
2
3  This creates a miter module for further analysis of the selected $equiv cells.
4
5      -trigger
6          Create a trigger output
7
8      -cmp
9          Create cmp_* outputs for individual unproven $equiv cells
10
11      -assert
12          Create a $assert cell for each unproven $equiv cell
13
14      -undef
15          Create compare logic that handles undefs correctly
```

C.54   equiv_opt – prove equivalence for optimized circuit

```
1      equiv_opt [options] [command]
2
3  This command uses temporal induction to check circuit equivalence before and
4  after an optimization pass.
5
6      -run <from_label>:<to_label>
```

```
   7          only run the commands between the labels (see below). an empty
   8          from label is synonymous to the start of the command list, and empty to
   9          label is synonymous to the end of the command list.
  10
  11      -map <filename>
  12          expand the modules in this file before proving equivalence. this is
  13          useful for handling architecture-specific primitives.
  14
  15      -blacklist <file>
  16          Do not match cells or signals that match the names in the file
  17          (passed to equiv_make).
  18
  19      -assert
  20          produce an error if the circuits are not equivalent.
  21
  22      -multiclock
  23          run clk2fflogic before equivalence checking.
  24
  25      -async2sync
  26          run async2sync before equivalence checking.
  27
  28      -undef
  29          enable modelling of undef states during equiv_induct.
  30
  31      -nocheck
  32          disable running check before and after the command under test.
  33
  34  The following commands are executed by this verification command:
  35
  36      run_pass:
  37          hierarchy -auto-top
  38          design -save preopt
  39          check -assert    (unless -nocheck)
  40          [command]
  41          check -assert    (unless -nocheck)
  42          design -stash postopt
  43
  44      prepare:
  45          design -copy-from preopt  -as gold A:top
  46          design -copy-from postopt -as gate A:top
  47
  48      techmap:   (only with -map)
  49          techmap -wb -D EQUIV -autoproc -map <filename> ...
  50
  51      prove:
  52          clk2fflogic     (only with -multiclock)
  53          async2sync      (only with -async2sync)
  54          equiv_make -blacklist <filename> ... gold gate equiv
  55          equiv_induct [-undef] equiv
  56          equiv_status [-assert] equiv
  57
  58      restore:
  59          design -load preopt
```

C.55   equiv_purge – purge equivalence checking module

```
1      equiv_purge [options] [selection]
2
3  This command removes the proven part of an equivalence checking module, leaving
4  only the unproven segments in the design. This will also remove and add module
5  ports as needed.
```

C.56   equiv_remove – remove $equiv cells

```
1      equiv_remove [options] [selection]
2
3  This command removes the selected $equiv cells. If neither -gold nor -gate is
4  used then only proven cells are removed.
5
6      -gold
7          keep gold circuit
8
9      -gate
10         keep gate circuit
```

C.57   equiv_simple – try proving simple $equiv instances

```
1      equiv_simple [options] [selection]
2
3  This command tries to prove $equiv cells using a simple direct SAT approach.
4
5      -v
6          verbose output
7
8      -undef
9          enable modelling of undef states
10
11     -short
12         create shorter input cones that stop at shared nodes. This yields
13         simpler SAT problems but sometimes fails to prove equivalence.
14
15     -nogroup
16         disabling grouping of $equiv cells by output wire
17
18     -seq <N>
19         the max. number of time steps to be considered (default = 1)
```

C.58   equiv_status – print status of equivalent checking module

```
1      equiv_status [options] [selection]
2
3  This command prints status information for all selected $equiv cells.
4
5      -assert
6          produce an error if any unproven $equiv cell is found
```

C.59   equiv_struct – structural equivalence checking

```
1      equiv_struct [options] [selection]
2
3  This command adds additional $equiv cells based on the assumption that the
4  gold and gate circuit are structurally equivalent. Note that this can introduce
5  bad $equiv cells in cases where the netlists are not structurally equivalent,
6  for example when analyzing circuits with cells with commutative inputs. This
7  command will also de-duplicate gates.
8
9      -fwd
10         by default this command performans forward sweeps until nothing can
11         be merged by forwards sweeps, then backward sweeps until forward
12         sweeps are effective again. with this option set only forward sweeps
13         are performed.
14
15     -fwonly <cell_type>
16         add the specified cell type to the list of cell types that are only
17         merged in forward sweeps and never in backward sweeps. $equiv is in
18         this list automatically.
19
20     -icells
21         by default, the internal RTL and gate cell types are ignored. add
22         this option to also process those cell types with this command.
23
24     -maxiter <N>
25         maximum number of iterations to run before aborting
```

C.60   eval – evaluate the circuit given an input

```
1      eval [options] [selection]
2
3  This command evaluates the value of a signal given the value of all required
4  inputs.
5
6      -set <signal> <value>
7          set the specified signal to the specified value.
8
9      -set-undef
10         set all unspecified source signals to undef (x)
11
12     -table <signal>
```

```
13          create a truth table using the specified input signals
14
15      -show <signal>
16          show the value for the specified signal. if no -show option is passed
17          then all output ports of the current module are used.
```

C.61   exec − execute commands in the operating system shell

```
1       exec [options] -- [command]
2
3   Execute a command in the operating system shell.  All supplied arguments are
4   concatenated and passed as a command to popen(3).  Whitespace is not guaranteed
5   to be preserved, even if quoted.  stdin and stderr are not connected, while
6   stdout is logged unless the "-q" option is specified.
7
8
9       -q
10          Suppress stdout and stderr from subprocess
11
12      -expect-return <int>
13          Generate an error if popen() does not return specified value.
14          May only be specified once; the final specified value is controlling
15          if specified multiple times.
16
17      -expect-stdout <regex>
18          Generate an error if the specified regex does not match any line
19          in subprocess's stdout.  May be specified multiple times.
20
21      -not-expect-stdout <regex>
22          Generate an error if the specified regex matches any line
23          in subprocess's stdout.  May be specified multiple times.
24
25
26      Example: exec -q -expect-return 0 -- echo "bananapie" | grep "nana"
```

C.62   expose − convert internal signals to module ports

```
1       expose [options] [selection]
2
3   This command exposes all selected internal signals of a module as additional
4   outputs.
5
6       -dff
7           only consider wires that are directly driven by register cell.
8
9       -cut
10          when exposing a wire, create an input/output pair and cut the internal
11          signal path at that wire.
12
13      -input
```

```
14          when exposing a wire, create an input port and disconnect the internal
15          driver.
16
17      -shared
18          only expose those signals that are shared among the selected modules.
19          this is useful for preparing modules for equivalence checking.
20
21      -evert
22          also turn connections to instances of other modules to additional
23          inputs and outputs and remove the module instances.
24
25      -evert-dff
26          turn flip-flops to sets of inputs and outputs.
27
28      -sep <separator>
29          when creating new wire/port names, the original object name is suffixed
30          with this separator (default: '.') and the port name or a type
31          designator for the exposed signal.
```

C.63   extract – find subcircuits and replace them with cells

```
1       extract -map <map_file> [options] [selection]
2       extract -mine <out_file> [options] [selection]
3
4  This pass looks for subcircuits that are isomorphic to any of the modules
5  in the given map file and replaces them with instances of this modules. The
6  map file can be a Verilog source file (*.v) or an RTLIL source file (*.il).
7
8       -map <map_file>
9           use the modules in this file as reference. This option can be used
10          multiple times.
11
12      -map %<design-name>
13          use the modules in this in-memory design as reference. This option can
14          be used multiple times.
15
16      -verbose
17          print debug output while analyzing
18
19      -constports
20          also find instances with constant drivers. this may be much
21          slower than the normal operation.
22
23      -nodefaultswaps
24          normally builtin port swapping rules for internal cells are used per
25          default. This turns that off, so e.g. 'a^b' does not match 'b^a'
26          when this option is used.
27
28      -compat <needle_type> <haystack_type>
29          Per default, the cells in the map file (needle) must have the
30          type as the cells in the active design (haystack). This option
31          can be used to register additional pairs of types that should
```

```
32              match. This option can be used multiple times.
33
34      -swap <needle_type> <port1>,<port2>[,...]
35          Register a set of swappable ports for a needle cell type.
36          This option can be used multiple times.
37
38      -perm <needle_type> <port1>,<port2>[,...] <portA>,<portB>[,...]
39          Register a valid permutation of swappable ports for a needle
40          cell type. This option can be used multiple times.
41
42      -cell_attr <attribute_name>
43          Attributes on cells with the given name must match.
44
45      -wire_attr <attribute_name>
46          Attributes on wires with the given name must match.
47
48      -ignore_parameters
49          Do not use parameters when matching cells.
50
51      -ignore_param <cell_type> <parameter_name>
52          Do not use this parameter when matching cells.
53
54  This pass does not operate on modules with unprocessed processes in it.
55  (I.e. the 'proc' pass should be used first to convert processes to netlists.)
56
57  This pass can also be used for mining for frequent subcircuits. In this mode
58  the following options are to be used instead of the -map option.
59
60      -mine <out_file>
61          mine for frequent subcircuits and write them to the given RTLIL file
62
63      -mine_cells_span <min> <max>
64          only mine for subcircuits with the specified number of cells
65          default value: 3 5
66
67      -mine_min_freq <num>
68          only mine for subcircuits with at least the specified number of matches
69          default value: 10
70
71      -mine_limit_matches_per_module <num>
72          when calculating the number of matches for a subcircuit, don't count
73          more than the specified number of matches per module
74
75      -mine_max_fanout <num>
76          don't consider internal signals with more than <num> connections
77
78  The modules in the map file may have the attribute 'extract_order' set to an
79  integer value. Then this value is used to determine the order in which the pass
80  tries to map the modules to the design (ascending, default value is 0).
81
82  See 'help techmap' for a pass that does the opposite thing.
```

C.64   extract_counter – Extract GreenPak4 counter cells

```
   extract_counter [options] [selection]

This pass converts non-resettable or async resettable down counters to
counter cells. Use a target-specific 'techmap' map file to convert those cells
to the actual target cells.

   -maxwidth N
       Only extract counters up to N bits wide (default 64)

   -minwidth N
       Only extract counters at least N bits wide (default 2)

   -allow_arst yes|no
       Allow counters to have async reset (default yes)

   -dir up|down|both
       Look for up-counters, down-counters, or both (default down)

   -pout X,Y,...
       Only allow parallel output from the counter to the listed cell types
       (if not specified, parallel outputs are not restricted)
```

C.65   extract_fa – find and extract full/half adders

```
   extract_fa [options] [selection]

This pass extracts full/half adders from a gate-level design.

   -fa, -ha
       Enable cell types (fa=full adder, ha=half adder)
       All types are enabled if none of this options is used

   -d <int>
       Set maximum depth for extracted logic cones (default=20)

   -b <int>
       Set maximum breadth for extracted logic cones (default=6)

   -v
       Verbose output
```

C.66   extract_reduce – converts gate chains into $reduce_* cells

```
   extract_reduce [options] [selection]

converts gate chains into $reduce_* cells
```

```
 5   This command finds chains of $_AND_, $_OR_, and $_XOR_ cells and replaces them
 6   with their corresponding $reduce_* cells. Because this command only operates on
 7   these cell types, it is recommended to map the design to only these cell types
 8   using the `abc -g` command. Note that, in some cases, it may be more effective
 9   to map the design to only $_AND_ cells, run extract_reduce, map the remaining
10   parts of the design to AND/OR/XOR cells, and run extract_reduce a second time.
11
12       -allow-off-chain
13           Allows matching of cells that have loads outside the chain. These cells
14           will be replicated and folded into the $reduce_* cell, but the original
15           cell will remain, driving its original loads.
```

C.67 extractinv – extract explicit inverter cells for invertible cell pins

```
 1       extractinv [options] [selection]
 2
 3   Searches the design for all cells with invertible pins controlled by a cell
 4   parameter (eg. IS_CLK_INVERTED on many Xilinx cells) and removes the parameter.
 5   If the parameter was set to 1, inserts an explicit inverter cell in front of
 6   the pin instead.  Normally used for output to ISE, which does not support the
 7   inversion parameters.
 8
 9   To mark a cell port as invertible, use (* invertible_pin = "param_name" *)
10   on the wire in the blackbox module.  The parameter value should have
11   the same width as the port, and will be effectively XORed with it.
12
13       -inv <celltype> <portname_out>:<portname_in>
14           Specifies the cell type to use for the inverters and its port names.
15           This option is required.
```

C.68 flatten – flatten design

```
 1       flatten [options] [selection]
 2
 3   This pass flattens the design by replacing cells by their implementation. This
 4   pass is very similar to the 'techmap' pass. The only difference is that this
 5   pass is using the current design as mapping library.
 6
 7   Cells and/or modules with the 'keep_hierarchy' attribute set will not be
 8   flattened by this command.
 9
10       -wb
11           Ignore the 'whitebox' attribute on cell implementations.
```

C.69 flowmap – pack LUTs with FlowMap

```
 1      flowmap [options] [selection]
 2
 3  This pass uses the FlowMap technology mapping algorithm to pack logic gates
 4  into k-LUTs with optimal depth. It allows mapping any circuit elements that can
 5  be evaluated with the `eval` pass, including cells with multiple output ports
 6  and multi-bit input and output ports.
 7
 8      -maxlut k
 9          perform technology mapping for a k-LUT architecture. if not specified,
10          defaults to 3.
11
12      -minlut n
13          only produce n-input or larger LUTs. if not specified, defaults to 1.
14
15      -cells <cell>[,<cell>,...]
16          map specified cells. if not specified, maps $_NOT_, $_AND_, $_OR_,
17          $_XOR_ and $_MUX_, which are the outputs of the `simplemap` pass.
18
19      -relax
20          perform depth relaxation and area minimization.
21
22      -r-alpha n, -r-beta n, -r-gamma n
23          parameters of depth relaxation heuristic potential function.
24          if not specified, alpha=8, beta=2, gamma=1.
25
26      -optarea n
27          optimize for area by trading off at most n logic levels for fewer LUTs.
28          n may be zero, to optimize for area without increasing depth.
29          implies -relax.
30
31      -debug
32          dump intermediate graphs.
33
34      -debug-relax
35          explain decisions performed during depth relaxation.
```

C.70   fmcombine – combine two instances of a cell into one

```
 1      fmcombine [options] module_name gold_cell gate_cell
 2
 3  This pass takes two cells, which are instances of the same module, and replaces
 4  them with one instance of a special 'combined' module, that effectively
 5  contains two copies of the original module, plus some formal properties.
 6
 7  This is useful for formal test benches that check what differences in behavior
 8  a slight difference in input causes in a module.
 9
10      -initeq
11          Insert assumptions that initially all FFs in both circuits have the
12          same initial values.
13
```

```
14        -anyeq
15            Do not duplicate $anyseq/$anyconst cells.
16
17        -fwd
18            Insert forward hint assumptions into the combined module.
19
20        -bwd
21            Insert backward hint assumptions into the combined module.
22            (Backward hints are logically equivalend to fordward hits, but
23            some solvers are faster with bwd hints, or even both -bwd and -fwd.)
24
25        -nop
26            Don't insert hint assumptions into the combined module.
27            (This should not provide any speedup over the original design, but
28            strangely sometimes it does.)
29
30    If none of -fwd, -bwd, and -nop is given, then -fwd is used as default.
```

C.71   fminit – set init values/sequences for formal

```
1        fminit [options] <selection>
2
3    This pass creates init constraints (for example for reset sequences) in a formal
4    model.
5
6        -seq <signal> <sequence>
7            Set sequence using comma-separated list of values, use 'z for
8            unconstrained bits. The last value is used for the remainder of the
9            trace.
10
11        -set <signal> <value>
12            Add constant value constraint
13
14        -posedge <signal>
15        -negedge <signal>
16            Set clock for init sequences
```

C.72   formalff – prepare FFs for formal

```
1        formalff [options] [selection]
2
3    This pass transforms clocked flip-flops to prepare a design for formal
4    verification. If a design contains latches and/or multiple different clocks run
5    the async2sync or clk2fflogic passes before using this pass.
6
7        -clk2ff
8            Replace all clocked flip-flops with $ff cells that use the implicit
9            global clock. This assumes, without checking, that the design uses a
10            single global clock. If that is not the case, the clk2fflogic pass
11            should be used instead.
```

```
12
13      -ff2anyinit
14          Replace uninitialized bits of $ff cells with $anyinit cells. An
15          $anyinit cell behaves exactly like an $ff cell with an undefined
16          initialization value. The difference is that $anyinit inhibits
17          don't-care optimizations and is used to track solver-provided values
18          in witness traces.
19
20          If combined with -clk2ff this also affects newly created $ff cells.
21
22      -anyinit2ff
23          Replaces $anyinit cells with uninitialized $ff cells. This performs the
24          reverse of -ff2anyinit and can be used, before running a backend pass
25          (or similar) that is not yet aware of $anyinit cells.
26
27          Note that after running -anyinit2ff, in general, performing don't-care
28          optimizations is not sound in a formal verification setting.
29
30      -fine
31          Emit fine-grained $_FF_ cells instead of coarse-grained $ff cells for
32          -anyinit2ff. Cannot be combined with -clk2ff or -ff2anyinit.
33
34      -setundef
35          Find FFs with undefined initialization values for which changing the
36          initialization does not change the observable behavior and initialize
37          them. For -ff2anyinit, this reduces the number of generated $anyinit
38          cells that drive wires with private names.
```

C.73   freduce – perform functional reduction

```
1       freduce [options] [selection]
2
3  This pass performs functional reduction in the circuit. I.e. if two nodes are
4  equivalent, they are merged to one node and one of the redundant drivers is
5  disconnected. A subsequent call to 'clean' will remove the redundant drivers.
6
7      -v, -vv
8          enable verbose or very verbose output
9
10      -inv
11          enable explicit handling of inverted signals
12
13      -stop <n>
14          stop after <n> reduction operations. this is mostly used for
15          debugging the freduce command itself.
16
17      -dump <prefix>
18          dump the design to <prefix>_<module>_<num>.il after each reduction
19          operation. this is mostly used for debugging the freduce command.
20
21  This pass is undef-aware, i.e. it considers don't-care values for detecting
22  equivalent nodes.
```

```
23
24  All selected wires are considered for rewiring. The selected cells cover the
25  circuit that is analyzed.
```

C.74 fsm – extract and optimize finite state machines

```
 1      fsm [options] [selection]
 2
 3  This pass calls all the other fsm_* passes in a useful order. This performs
 4  FSM extraction and optimization. It also calls opt_clean as needed:
 5
 6      fsm_detect          unless got option -nodetect
 7      fsm_extract
 8
 9      fsm_opt
10      opt_clean
11      fsm_opt
12
13      fsm_expand          if got option -expand
14      opt_clean           if got option -expand
15      fsm_opt             if got option -expand
16
17      fsm_recode          unless got option -norecode
18
19      fsm_info
20
21      fsm_export          if got option -export
22      fsm_map             unless got option -nomap
23
24  Options:
25
26      -expand, -norecode, -export, -nomap
27          enable or disable passes as indicated above
28
29      -fullexpand
30          call expand with -full option
31
32      -encoding type
33      -fm_set_fsm_file file
34      -encfile file
35          passed through to fsm_recode pass
```

C.75 fsm_detect – finding FSMs in design

```
 1      fsm_detect [selection]
 2
 3  This pass detects finite state machines by identifying the state signal.
 4  The state signal is then marked by setting the attribute 'fsm_encoding'
 5  on the state signal to "auto".
 6
```

```
7   Existing 'fsm_encoding' attributes are not changed by this pass.
8
9   Signals can be protected from being detected by this pass by setting the
10  'fsm_encoding' attribute to "none".
```

C.76 fsm_expand – expand FSM cells by merging logic into it

```
1       fsm_expand [-full] [selection]
2
3   The fsm_extract pass is conservative about the cells that belong to a finite
4   state machine. This pass can be used to merge additional auxiliary gates into
5   the finite state machine.
6
7   By default, fsm_expand is still a bit conservative regarding merging larger
8   word-wide cells. Call with -full to consider all cells for merging.
```

C.77 fsm_export – exporting FSMs to KISS2 files

```
1       fsm_export [-noauto] [-o filename] [-origenc] [selection]
2
3   This pass creates a KISS2 file for every selected FSM. For FSMs with the
4   'fsm_export' attribute set, the attribute value is used as filename, otherwise
5   the module and cell name is used as filename. If the parameter '-o' is given,
6   the first exported FSM is written to the specified filename. This overwrites
7   the setting as specified with the 'fsm_export' attribute. All other FSMs are
8   exported to the default name as mentioned above.
9
10      -noauto
11          only export FSMs that have the 'fsm_export' attribute set
12
13      -o filename
14          filename of the first exported FSM
15
16      -origenc
17          use binary state encoding as state names instead of s0, s1, ...
```

C.78 fsm_extract – extracting FSMs in design

```
1       fsm_extract [selection]
2
3   This pass operates on all signals marked as FSM state signals using the
4   'fsm_encoding' attribute. It consumes the logic that creates the state signal
5   and uses the state signal to generate control signal and replaces it with an
6   FSM cell.
7
8   The generated FSM cell still generates the original state signal with its
9   original encoding. The 'fsm_opt' pass can be used in combination with the
10  'opt_clean' pass to eliminate this signal.
```

C.79 fsm_info – print information on finite state machines

```
1      fsm_info [selection]
2
3   This pass dumps all internal information on FSM cells. It can be useful for
4   analyzing the synthesis process and is called automatically by the 'fsm'
5   pass so that this information is included in the synthesis log file.
```

C.80 fsm_map – mapping FSMs to basic logic

```
1      fsm_map [selection]
2
3   This pass translates FSM cells to flip-flops and logic.
```

C.81 fsm_opt – optimize finite state machines

```
1      fsm_opt [selection]
2
3   This pass optimizes FSM cells. It detects which output signals are actually
4   not used and removes them from the FSM. This pass is usually used in
5   combination with the 'opt_clean' pass (see also 'help fsm').
```

C.82 fsm_recode – recoding finite state machines

```
1      fsm_recode [options] [selection]
2
3   This pass reassign the state encodings for FSM cells. At the moment only
4   one-hot encoding and binary encoding is supported.
5       -encoding <type>
6           specify the encoding scheme used for FSMs without the
7           'fsm_encoding' attribute or with the attribute set to `auto'.
8
9       -fm_set_fsm_file <file>
10          generate a file containing the mapping from old to new FSM encoding
11          in form of Synopsys Formality set_fsm_* commands.
12
13      -encfile <file>
14          write the mappings from old to new FSM encoding to a file in the
15          following format:
16
17              .fsm <module_name> <state_signal>
18              .map <old_bitpattern> <new_bitpattern>
```

C.83   fst2tb – generate testbench out of fst file

```
 1       fst2tb [options] [top-level]
 2
 3    This command generates testbench for the circuit using the given top-level
 4    module and simulus signal from FST file
 5
 6        -tb <name>
 7            generated testbench name.
 8            files <name>.v and <name>.txt are created as result.
 9
10        -r <filename>
11            read simulation FST file
12
13        -clock <portname>
14            name of top-level clock input
15
16        -clockn <portname>
17            name of top-level clock input (inverse polarity)
18
19        -scope <name>
20            scope of simulation top model
21
22        -start <time>
23            start co-simulation in arbitary time (default 0)
24
25        -stop <time>
26            stop co-simulation in arbitary time (default END)
27
28        -n <integer>
29            number of clock cycles to simulate (default: 20)
```

C.84   gatemate_foldinv – fold inverters into Gatemate LUT trees

```
 1       gatemate_foldinv [selection]
 2
 3
 4    This pass searches for $__CC_NOT cells and folds them into CC_LUT2, CC_L2T4
 5    and CC_L2T5 cells as created by LUT tree mapping.
```

C.85   glift – create GLIFT models and optimization problems

```
 1       glift <command> [options] [selection]
 2
 3    Augments the current or specified module with gate-level information flow
 4    tracking (GLIFT) logic using the "constructive mapping" approach. Also can set
 5    up QBF-SAT optimization problems in order to optimize GLIFT models or trade off
 6    precision and complexity.
 7
```

```
Commands:

  -create-precise-model
    Replaces the current or specified module with one that has corresponding
    "taint" inputs, outputs, and internal nets along with precise taint
    tracking logic. For example, precise taint tracking logic for an AND gate
    is:

      y_t = a & b_t | b & a_t | a_t & b_t


  -create-imprecise-model
    Replaces the current or specified module with one that has corresponding
    "taint" inputs, outputs, and internal nets along with imprecise "All OR"
    taint tracking logic:

      y_t = a_t | b_t


  -create-instrumented-model
    Replaces the current or specified module with one that has corresponding
    "taint" inputs, outputs, and internal nets along with 4 varying-precision
    versions of taint tracking logic. Which version of taint tracking logic is
    used for a given gate is determined by a MUX selected by an $anyconst cell.
     By default, unless the `-no-cost-model` option is provided, an additional
    wire named `__glift_weight` with the `keep` and `minimize` attributes is
    added to the module along with pmuxes and adders to calculate a rough
    estimate of the number of logic gates in the GLIFT model given an assignment
    for the $anyconst cells. The four versions of taint tracking logic for an
    AND gate are:
      y_t = a & b_t | b & a_t | a_t & b_t        (like `-create-precise-model`)
      y_t = a_t | a & b_t
      y_t = b_t | b & a_t
      y_t = a_t | b_t                                (like `-create-imprecise-model`)


Options:

  -taint-constants
    Constant values in the design are labeled as tainted.
    (default: label constants as un-tainted)

  -keep-outputs
    Do not remove module outputs. Taint tracking outputs will appear in the
    module ports alongside the orignal outputs.
    (default: original module outputs are removed)

  -simple-cost-model
    Do not model logic area. Instead model the number of non-zero assignments to
    $anyconsts. Taint tracking logic versions vary in their size, but all
    reduced-precision versions are significantly smaller than the fully-precise
    version. A non-zero $anyconst assignment means that reduced-precision taint
    tracking logic was chosen for some gate. Only applicable in combination with
```

```
62        `-create-instrumented-model`. (default: use a complex model and give that
63         wire the "keep" and "minimize" attributes)
64
65      -no-cost-model
66        Do not model taint tracking logic area and do not create a `__glift_weight`
67        wire. Only applicable in combination with `-create-instrumented-model`.
68        (default: model area and give that wire the "keep" and "minimize"
69        attributes)
70
71      -instrument-more
72        Allow choice from more versions of (even simpler) taint tracking logic. A
73        total of 8 versions of taint tracking logic will be added per gate,
74        including the 4 versions from `-create-instrumented-model` and these
75        additional versions:
76
77          y_t = a_t
78          y_t = b_t
79          y_t = 1
80          y_t = 0
81
82        Only applicable in combination with `-create-instrumented-model`.
83        (default: do not add more versions of taint tracking logic.
```

C.86 greenpak4_dffinv – merge greenpak4 inverters and DF-F/latches

```
1      greenpak4_dffinv [options] [selection]
2
3  Merge GP_INV cells with GP_DFF* and GP_DLATCH* cells.
```

C.87 help – display help messages

```
1      help  ...............  list all commands
2      help <command>  ......  print help message for given command
3      help -all  ..........  print complete command reference
4
5      help -cells ..........  list all cell types
6      help <celltype>  .....  print help message for given cell type
7      help <celltype>+  ....  print verilog code for given cell type
```

C.88 hierarchy – check, expand and clean up design hierarchy

```
1      hierarchy [-check] [-top <module>]
2      hierarchy -generate <cell-types> <port-decls>
3
4  In parametric designs, a module might exists in several variations with
5  different parameter values. This pass looks at all modules in the current
```

```
 6  design and re-runs the language frontends for the parametric modules as
 7  needed. It also resolves assignments to wired logic data types (wand/wor),
 8  resolves positional module parameters, unrolls array instances, and more.
 9
10      -check
11          also check the design hierarchy. this generates an error when
12          an unknown module is used as cell type.
13
14      -simcheck
15          like -check, but also throw an error if blackbox modules are
16          instantiated, and throw an error if the design has no top module.
17
18      -smtcheck
19          like -simcheck, but allow smtlib2_module modules.
20
21      -purge_lib
22          by default the hierarchy command will not remove library (blackbox)
23          modules. use this option to also remove unused blackbox modules.
24
25      -libdir <directory>
26          search for files named <module_name>.v in the specified directory
27          for unknown modules and automatically run read_verilog for each
28          unknown module.
29
30      -keep_positionals
31          per default this pass also converts positional arguments in cells
32          to arguments using port names. This option disables this behavior.
33
34      -keep_portwidths
35          per default this pass adjusts the port width on cells that are
36          module instances when the width does not match the module port. This
37          option disables this behavior.
38
39      -nodefaults
40          do not resolve input port default values
41
42      -nokeep_asserts
43          per default this pass sets the "keep" attribute on all modules
44          that directly or indirectly contain one or more formal properties.
45          This option disables this behavior.
46
47      -top <module>
48          use the specified top module to build the design hierarchy. Modules
49          outside this tree (unused modules) are removed.
50
51          when the -top option is used, the 'top' attribute will be set on the
52          specified top module. otherwise a module with the 'top' attribute set
53          will implicitly be used as top module, if such a module exists.
54
55      -auto-top
56          automatically determine the top of the design hierarchy and mark it.
57
58      -chparam name value
59          elaborate the top module using this parameter value. Modules on which
```

```
60         this parameter does not exist may cause a warning message to be output.
61         This option can be specified multiple times to override multiple
62         parameters. String values must be passed in double quotes (").
63
64  In -generate mode this pass generates blackbox modules for the given cell
65  types (wildcards supported). For this the design is searched for cells that
66  match the given types and then the given port declarations are used to
67  determine the direction of the ports. The syntax for a port declaration is:
68
69      {i|o|io}[@<num>]:<portname>
70
71  Input ports are specified with the 'i' prefix, output ports with the 'o'
72  prefix and inout ports with the 'io' prefix. The optional <num> specifies
73  the position of the port in the parameter list (needed when instantiated
74  using positional arguments). When <num> is not specified, the <portname> can
75  also contain wildcard characters.
76
77  This pass ignores the current selection and always operates on all modules
78  in the current design.
```

C.89 hilomap – technology mapping of constant hi- and/or lo-drivers

```
1      hilomap [options] [selection]
2
3  Map constants to 'tielo' and 'tiehi' driver cells.
4
5      -hicell <celltype> <portname>
6          Replace constant hi bits with this cell.
7
8      -locell <celltype> <portname>
9          Replace constant lo bits with this cell.
10
11     -singleton
12         Create only one hi/lo cell and connect all constant bits
13         to that cell. Per default a separate cell is created for
14         each constant bit.
```

C.90 history – show last interactive commands

```
1      history
2
3  This command prints all commands in the shell history buffer. This are
4  all commands executed in an interactive session, but not the commands
5  from executed scripts.
```

C.91   ice40_braminit – iCE40: perform SB_RAM40_4K initialization from file

```
1     ice40_braminit
2
3  This command processes all SB_RAM40_4K blocks with a non-empty INIT_FILE
4  parameter and converts it into the required INIT_x attributes
```

C.92   ice40_dsp – iCE40: map multipliers

```
1     ice40_dsp [options] [selection]
2
3  Map multipliers ($mul/SB_MAC16) and multiply-accumulate ($mul/SB_MAC16 + $add)
4  cells into iCE40 DSP resources.
5  Currently, only the 16x16 multiply mode is supported and not the 2 x 8x8 mode.
6
7  Pack input registers (A, B, {C,D}; with optional hold), pipeline registers
8  ({F,J,K,G}, H), output registers (O -- full 32-bits or lower 16-bits only; with
9  optional hold), and post-adder into into the SB_MAC16 resource.
10
11  Multiply-accumulate operations using the post-adder with feedback on the {C,D}
12  input will be folded into the DSP. In this scenario only, resetting the
13  the accumulator to an arbitrary value can be inferred to use the {C,D} input.
```

C.93   ice40_opt – iCE40: perform simple optimizations

```
1     ice40_opt [options] [selection]
2
3  This command executes the following script:
4
5      do
6          <ice40 specific optimizations>
7          opt_expr -mux_undef -undriven [-full]
8          opt_merge
9          opt_dff
10          opt_clean
11      while <changed design>
```

C.94   ice40_wrapcarry – iCE40: wrap carries

```
1     ice40_wrapcarry [selection]
2
3  Wrap manually instantiated SB_CARRY cells, along with their associated SB_LUT4s,
4  into an internal $__ICE40_CARRY_WRAPPER cell for preservation across technology
5  mapping.
6
```

```
 7  Attributes on both cells will have their names prefixed with 'SB_CARRY.' or
 8  'SB_LUT4.' and attached to the wrapping cell.
 9  A (* keep *) attribute on either cell will be logically OR-ed together.
10
11      -unwrap
12          unwrap $__ICE40_CARRY_WRAPPER cells back into SB_CARRYs and SB_LUT4s,
13          including restoring their attributes.
```

C.95 insbuf – insert buffer cells for connected wires

```
 1      insbuf [options] [selection]
 2
 3  Insert buffer cells into the design for directly connected wires.
 4
 5      -buf <celltype> <in-portname> <out-portname>
 6          Use the given cell type instead of $_BUF_. (Notice that the next
 7          call to "clean" will remove all $_BUF_ in the design.)
```

C.96 iopadmap – technology mapping of i/o pads (or buffers)

```
 1      iopadmap [options] [selection]
 2
 3  Map module inputs/outputs to PAD cells from a library. This pass
 4  can only map to very simple PAD cells. Use 'techmap' to further map
 5  the resulting cells to more sophisticated PAD cells.
 6
 7      -inpad <celltype> <in_port>[:<ext_port>]
 8          Map module input ports to the given cell type with the
 9          given output port name. if a 2nd portname is given, the
10          signal is passed through the pad cell, using the 2nd
11          portname as the port facing the module port.
12
13      -outpad <celltype> <out_port>[:<ext_port>]
14      -inoutpad <celltype> <io_port>[:<ext_port>]
15          Similar to -inpad, but for output and inout ports.
16
17      -toutpad <celltype> <oe_port>:<out_port>[:<ext_port>]
18          Merges $_TBUF_ cells into the output pad cell. This takes precedence
19          over the other -outpad cell. The first portname is the enable input
20          of the tristate driver, which can be prefixed with `~` for negative
21          polarity enable.
22
23      -tinoutpad <celltype> <oe_port>:<in_port>:<out_port>[:<ext_port>]
24          Merges $_TBUF_ cells into the inout pad cell. This takes precedence
25          over the other -inoutpad cell. The first portname is the enable input
26          of the tristate driver and the 2nd portname is the internal output
27          buffering the external signal.  Like with `-toutpad`, the enable can
28          be marked as negative polarity by prefixing the name with `~`.
29
30      -ignore <celltype> <portname>[:<portname>]*
```

```
31          Skips mapping inputs/outputs that are already connected to given
32          ports of the given cell.  Can be used multiple times.  This is in
33          addition to the cells specified as mapping targets.
34
35      -widthparam <param_name>
36          Use the specified parameter name to set the port width.
37
38      -nameparam <param_name>
39          Use the specified parameter to set the port name.
40
41      -bits
42          create individual bit-wide buffers even for ports that
43          are wider. (the default behavior is to create word-wide
44          buffers using -widthparam to set the word size on the cell.)
45
46  Tristate PADS (-toutpad, -tinoutpad) always operate in -bits mode.
```

C.97   jny – write design and metadata

```
1       jny [options] [selection]
2
3   Write JSON netlist metadata for the current design
4
5       -o <filename>
6           write to the specified file.
7
8       -no-connections
9           Don't include connection information in the netlist output.
10
11      -no-attributes
12          Don't include attributed information in the netlist output.
13
14      -no-properties
15          Don't include property information in the netlist output.
16
17  See 'help write_jny' for a description of the JSON format used.
```

C.98   json – write design in JSON format

```
1       json [options] [selection]
2
3   Write a JSON netlist of all selected objects.
4
5       -o <filename>
6           write to the specified file.
7
8       -aig
9           also include AIG models for the different gate types
10
11      -compat-int
```

```
12          emit 32-bit or smaller fully-defined parameter values directly
13          as JSON numbers (for compatibility with old parsers)
14
15  See 'help write_json' for a description of the JSON format used.
```

C.99   log – print text and log files

```
 1      log string
 2
 3  Print the given string to the screen and/or the log file. This is useful for TCL
 4  scripts, because the TCL command "puts" only goes to stdout but not to
 5  logfiles.
 6
 7      -stdout
 8          Print the output to stdout too. This is useful when all Yosys is
 9          executed with a script and the -q (quiet operation) argument to notify
10          the user.
11
12      -stderr
13          Print the output to stderr too.
14
15      -nolog
16          Don't use the internal log() command. Use either -stdout or -stderr,
17          otherwise no output will be generated at all.
18
19      -n
20          do not append a newline
```

C.100   logger – set logger properties

```
 1      logger [options]
 2
 3  This command sets global logger properties, also available using command line
 4  options.
 5
 6      -[no]time
 7          enable/disable display of timestamp in log output.
 8
 9      -[no]stderr
10          enable/disable logging errors to stderr.
11
12      -warn regex
13          print a warning for all log messages matching the regex.
14
15      -nowarn regex
16          if a warning message matches the regex, it is printed as regular
17          message instead.
18
19      -werror regex
20          if a warning message matches the regex, it is printed as error
```

```
21          message instead and the tool terminates with a nonzero return code.
22
23      -[no]debug
24          globally enable/disable debug log messages.
25
26      -experimental <feature>
27          do not print warnings for the specified experimental feature
28
29      -expect <type> <regex> <expected_count>
30          expect log, warning or error to appear. matched errors will terminate
31          with exit code 0.
32
33      -expect-no-warnings
34          gives error in case there is at least one warning that is not expected.
35
36      -check-expected
37          verifies that the patterns previously set up by -expect have actually
38          been met, then clears the expected log list.  If this is not called
39          manually, the check will happen at yosys exist time instead.
```

C.101   ls – list modules or objects in modules

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

C.102   ltp – print longest topological path

```
1      ltp [options] [selection]
2
3  This command prints the longest topological path in the design. (Only considers
4  paths within a single module, so the design must be flattened.)
5
6      -noff
7          automatically exclude FF cell types
```

C.103   lut2mux – convert $lut to $_MUX_

```
1      lut2mux [options] [selection]
2
3  This pass converts $lut cells to $_MUX_ gates.
```

C.104   maccmap − mapping macc cells

```
1 |     maccmap [-unmap] [selection]
2 |
3 | This pass maps $macc cells to yosys $fa and $alu cells. When the -unmap option
4 | is used then the $macc cell is mapped to $add, $sub, etc. cells instead.
```

C.105   memory − translate memories to basic cells

```
1 |     memory [-norom] [-nomap] [-nordff] [-nowiden] [-nosat] [-memx] [-no-rw-check] [-bram <bram_rules>] [sel
2 |
3 | This pass calls all the other memory_* passes in a useful order:
4 |
5 |     opt_mem
6 |     opt_mem_priority
7 |     opt_mem_feedback
8 |     memory_bmux2rom                (skipped if called with -norom)
9 |     memory_dff [-no-rw-check]      (skipped if called with -nordff or -memx)
10|     opt_clean
11|     memory_share [-nowiden] [-nosat]
12|     opt_mem_widen
13|     memory_memx                   (when called with -memx)
14|     opt_clean
15|     memory_collect
16|     memory_bram -rules <bram_rules>   (when called with -bram)
17|     memory_map                    (skipped if called with -nomap)
18|
19| This converts memories to word-wide DFFs and address decoders
20| or multiport memory blocks if called with the -nomap option.
```

C.106   memory_bmux2rom − convert muxes to ROMs

```
1 |     memory_bmux2rom [options] [selection]
2 |
3 | This pass converts $bmux cells with constant A input to ROMs.
```

C.107   memory_bram − map memories to block rams

```
1 |     memory_bram -rules <rule_file> [selection]
2 |
3 | This pass converts the multi-port $mem memory cells into block ram instances.
4 | The given rules file describes the available resources and how they should be
5 | used.
6 |
7 | The rules file contains configuration options, a set of block ram description
8 | and a sequence of match rules.
9 |
```

```
10  The option 'attr_icase' configures how attribute values are matched. The value 0
11  means case-sensitive, 1 means case-insensitive.
12
13  A block ram description looks like this:
14
15      bram RAMB1024X32      # name of BRAM cell
16        init 1             # set to '1' if BRAM can be initialized
17        abits 10           # number of address bits
18        dbits 32           # number of data bits
19        groups 2           # number of port groups
20        ports  1 1         # number of ports in each group
21        wrmode 1 0         # set to '1' if this groups is write ports
22        enable 4 1         # number of enable bits
23        transp 0 2         # transparent (for read ports)
24        clocks 1 2         # clock configuration
25        clkpol 2 2         # clock polarity configuration
26      endbram
27
28  For the option 'transp' the value 0 means non-transparent, 1 means transparent
29  and a value greater than 1 means configurable. All groups with the same
30  value greater than 1 share the same configuration bit.
31
32  For the option 'clocks' the value 0 means non-clocked, and a value greater
33  than 0 means clocked. All groups with the same value share the same clock
34  signal.
35
36  For the option 'clkpol' the value 0 means negative edge, 1 means positive edge
37  and a value greater than 1 means configurable. All groups with the same value
38  greater than 1 share the same configuration bit.
39
40  Using the same bram name in different bram blocks will create different variants
41  of the bram. Verilog configuration parameters for the bram are created as
42  needed.
43
44  It is also possible to create variants by repeating statements in the bram block
45  and appending '@<label>' to the individual statements.
46
47  A match rule looks like this:
48
49      match RAMB1024X32
50        max waste 16384    # only use this bram if <= 16k ram bits are unused
51        min efficiency 80  # only use this bram if efficiency is at least 80%
52      endmatch
53
54  It is possible to match against the following values with min/max rules:
55
56      words  ........  number of words in memory in design
57      abits  ........  number of address bits on memory in design
58      dbits  ........  number of data bits on memory in design
59      wports  .......  number of write ports on memory in design
60      rports  .......  number of read ports on memory in design
61      ports  ........  number of ports on memory in design
62      bits  .........  number of bits in memory in design
63      dups ..........  number of duplications for more read ports
```

```
64
65       awaste  .......   number of unused address slots for this match
66       dwaste  .......   number of unused data bits for this match
67       bwaste  .......   number of unused bram bits for this match
68       waste  ........   total number of unused bram bits (bwaste*dups)
69       efficiency  ...   total percentage of used and non-duplicated bits
70
71       acells  .......   number of cells in 'address-direction'
72       dcells  .......   number of cells in 'data-direction'
73       cells  .......    total number of cells (acells*dcells*dups)
74
75  A match containing the command 'attribute' followed by a list of space
76  separated 'name[=string_value]' values requires that the memory contains any
77  one of the given attribute name and string values (where specified), or name
78  and integer 1 value (if no string_value given, since Verilog will interpret
79  '(* attr *)' as '(* attr=1 *)').
80  A name prefixed with '!' indicates that the attribute must not exist.
81
82  The interface for the created bram instances is derived from the bram
83  description. Use 'techmap' to convert the created bram instances into
84  instances of the actual bram cells of your target architecture.
85
86  A match containing the command 'or_next_if_better' is only used if it
87  has a higher efficiency than the next match (and the one after that if
88  the next also has 'or_next_if_better' set, and so forth).
89
90  A match containing the command 'make_transp' will add external circuitry
91  to simulate 'transparent read', if necessary.
92
93  A match containing the command 'make_outreg' will add external flip-flops
94  to implement synchronous read ports, if necessary.
95
96  A match containing the command 'shuffle_enable A' will re-organize
97  the data bits to accommodate the enable pattern of port A.
```

C.108   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.109   memory_dff – merge input/output DFFs into memory read ports

```
1       memory_dff [-no-rw-check] [selection]
2
3  This pass detects DFFs at memory read ports and merges them into the memory
4  port. I.e. it consumes an asynchronous memory port and the flip-flops at its
```

```
 5   interface and yields a synchronous memory port.
 6
 7       -no-rw-check
 8           marks all recognized read ports as "return don't-care value on
 9           read/write collision" (same result as setting the no_rw_check
10           attribute on all memories).
```

C.110 memory_libmap – map memories to cells

```
 1       memory_libmap -lib <library_file> [-D <condition >] [selection]
 2
 3   This pass takes a description of available RAM cell types and maps
 4   all selected memories to one of them, or leaves them  to be mapped to FFs.
 5
 6     -lib <library_file >
 7       Selects a library file containing RAM cell definitions. This option
 8       can be passed more than once to select multiple libraries.
 9       See passes/memory/memlib.md for description of the library format.
10
11     -D <condition >
12       Enables a condition that can be checked within the library file
13       to eg. select between slightly different hardware variants.
14       This option can be passed any number of times.
15
16     -logic-cost-rom <num>
17     -logic-cost-ram <num>
18       Sets the cost of a single bit for memory lowered to soft logic.
19
20     -no-auto-distributed
21     -no-auto-block
22     -no-auto-huge
23       Disables automatic mapping of given kind of RAMs.  Manual mapping
24       (using ram_style or other attributes) is still supported.
```

C.111 memory_map – translate multiport memories to basic cells

```
 1       memory_map [options] [selection]
 2
 3   This pass converts multiport memory cells as generated by the memory_collect
 4   pass to word-wide DFFs and address decoders.
 5
 6       -attr !<name>
 7           do not map memories that have attribute <name> set.
 8
 9       -attr <name>[=<value >]
10           for memories that have attribute <name> set, only map them if its value
11           is a string <value> (if specified), or an integer 1 (otherwise). if this
12           option is specified multiple times, map the memory if the attribute is
```

```
13          to any of the values.
14
15      -iattr
16          for -attr, ignore case of <value>.
17
18      -rom-only
19          only perform conversion for ROMs (memories with no write ports).
20
21      -keepdc
22          when mapping ROMs, keep x-bits shared across read ports.
23
24      -formal
25          map memories for a global clock based formal verification flow.
26          This implies -keepdc, uses $ff cells for ROMs and sets hdlname
27          attributes. It also has limited support for async write ports
28          as generated by clk2fflogic.
```

C.112   memory_memx – emulate vlog sim behavior for mem ports

```
1      memory_memx [selection]
2
3  This pass adds additional circuitry that emulates the Verilog simulation
4  behavior for out-of-bounds memory reads and writes.
```

C.113   memory_narrow – split up wide memory ports

```
1      memory_narrow [options] [selection]
2
3  This pass splits up wide memory ports into several narrow ports.
```

C.114   memory_nordff – extract read port FFs from memories

```
1      memory_nordff [options] [selection]
2
3  This pass extracts FFs from memory read ports. This results in a netlist
4  similar to what one would get from not calling memory_dff.
```

C.115   memory_share – consolidate memory ports

```
1      memory_share [-nosat] [-nowiden] [selection]
2
3  This pass merges share-able memory ports into single memory ports.
4
```

```
5   The following methods are used to consolidate the number of memory ports:
6
7     - When multiple write ports access the same address then this is converted
8       to a single write port with a more complex data and/or enable logic path.
9
10    - When multiple read or write ports access adjacent aligned addresses, they
11      are merged to a single wide read or write port.  This transformation can be
12      disabled with the "-nowiden" option.
13
14    - When multiple write ports are never accessed at the same time (a SAT
15      solver is used to determine this), then the ports are merged into a single
16      write port.  This transformation can be disabled with the "-nosat" option.
17
18  Note that in addition to the algorithms implemented in this pass, the $memrd
19  and $memwr cells are also subject to generic resource sharing passes (and other
20  optimizations) such as "share" and "opt_merge".
```

C.116   memory_unpack – unpack multi-port memory cells

```
1       memory_unpack [selection]
2
3   This pass converts the multi-port $mem memory cells into individual $memrd and
4   $memwr cells. It is the counterpart to the memory_collect pass.
```

C.117   miter – automatically create a miter circuit

```
1       miter -equiv [options] gold_name gate_name miter_name
2
3   Creates a miter circuit for equivalence checking. The gold- and gate- modules
4   must have the same interfaces. The miter circuit will have all inputs of the
5   two source modules, prefixed with 'in_'. The miter circuit has a 'trigger'
6   output that goes high if an output mismatch between the two source modules is
7   detected.
8
9       -ignore_gold_x
10          a undef (x) bit in the gold module output will match any value in
11          the gate module output.
12
13      -make_outputs
14          also route the gold- and gate-outputs to 'gold_*' and 'gate_*' outputs
15          on the miter circuit.
16
17      -make_outcmp
18          also create a cmp_* output for each gold/gate output pair.
19
20      -make_assert
21          also create an 'assert' cell that checks if trigger is always low.
22
23      -flatten
24          call 'flatten -wb; opt_expr -keepdc -undriven;;' on the miter circuit.
```

```
25
26
27     -cross
28         allow output ports on the gold module to match input ports on the
29         gate module. This is useful when the gold module contains additional
30         logic to drive some of the gate module inputs.
31
32
33     miter -assert [options] module [miter_name]
34
35 Creates a miter circuit for property checking. All input ports are kept,
36 output ports are discarded. An additional output 'trigger' is created that
37 goes high when an assert is violated. Without a miter_name, the existing
38 module is modified.
39
40     -make_outputs
41         keep module output ports.
42
43     -flatten
44         call 'flatten -wb; opt_expr -keepdc -undriven;;' on the miter circuit.
```

C.118   mutate – generate or apply design mutations

```
1      mutate -list N [options] [selection]
2
3 Create a list of N mutations using an even sampling.
4
5     -o filename
6         Write list to this file instead of console output
7
8     -s filename
9         Write a list of all src tags found in the design to the specified file
10
11     -seed N
12         RNG seed for selecting mutations
13
14     -none
15         Include a "none" mutation in the output
16
17     -ctrl name width value
18         Add -ctrl options to the output. Use 'value' for first mutation, then
19         simply count up from there.
20
21     -mode name
22     -module name
23     -cell name
24     -port name
25     -portbit int
26     -ctrlbit int
27     -wire name
28     -wirebit int
29     -src string
```

```
30          Filter list of mutation candidates to those matching
31          the given parameters.
32
33      -cfg option int
34          Set a configuration option. Options available:
35            weight_pq_w weight_pq_b weight_pq_c weight_pq_s
36            weight_pq_mw weight_pq_mb weight_pq_mc weight_pq_ms
37            weight_cover pick_cover_prcnt
38
39
40      mutate -mode MODE [options]
41
42  Apply the given mutation.
43
44      -ctrl name width value
45          Add a control signal with the given name and width. The mutation is
46          activated if the control signal equals the given value.
47
48      -module name
49      -cell name
50      -port name
51      -portbit int
52      -ctrlbit int
53          Mutation parameters, as generated by 'mutate -list N'.
54
55      -wire name
56      -wirebit int
57      -src string
58          Ignored. (They are generated by -list for documentation purposes.)
```

C.119   muxcover – cover trees of MUX cells with wider MUXes

```
1       muxcover [options] [selection]
2
3   Cover trees of $_MUX_ cells with $_MUX{4,8,16}_ cells
4
5       -mux4[=cost], -mux8[=cost], -mux16[=cost]
6           Cover $_MUX_ trees using the specified types of MUXes (with optional
7           integer costs). If none of these options are given, the effect is the
8           same as if all of them are.
9           Default costs: $_MUX4_ = 220, $_MUX8_ = 460,
10                          $_MUX16_ = 940
11
12      -mux2=cost
13          Use the specified cost for $_MUX_ cells when making covering decisions.
14          Default cost: $_MUX_ = 100
15
16      -dmux=cost
17          Use the specified cost for $_MUX_ cells used in decoders.
18          Default cost: 90
19
20      -nodecode
```

```
21          Do not insert decoder logic. This reduces the number of possible
22          substitutions , but guarantees that the resulting circuit is not
23          less efficient than the original circuit.
24
25      -nopartial
26          Do not consider mappings that use $_MUX<N>_ to select from less
27          than <N> different signals.
```

C.120 muxpack – $mux/$pmux cascades to $pmux

```
1       muxpack [selection]
2
3   This pass converts cascaded chains of $pmux cells (e.g. those create from case
4   constructs) and $mux cells (e.g. those created by if-else constructs) into
5   $pmux cells.
6
7   This optimisation is conservative --- it will only pack $mux or $pmux cells
8   whose select lines are driven by '$eq' cells with other such cells if it can be
9   certain that their select inputs are mutually exclusive.
```

C.121 nlutmap – map to LUTs of different sizes

```
1       nlutmap [options] [selection]
2
3   This pass uses successive calls to 'abc' to map to an architecture. That
4   provides a small number of differently sized LUTs.
5
6       -luts N_1,N_2,N_3,...
7           The number of LUTs with 1, 2, 3, ... inputs that are
8           available in the target architecture.
9
10      -assert
11          Create an error if not all logic can be mapped
12
13  Excess logic that does not fit into the specified LUTs is mapped back
14  to generic logic gates ($_AND_, etc.).
```

C.122 onehot – optimize $eq cells for onehot signals

```
1       onehot [options] [selection]
2
3   This pass optimizes $eq cells that compare one-hot signals against constants
4
5       -v, -vv
6           verbose output
```

C.123 opt – perform simple optimizations

```
1      opt [options] [selection]
2
3   This pass calls all the other opt_* passes in a useful order. This performs
4   a series of trivial optimizations and cleanups. This pass executes the other
5   passes in the following order:
6
7       opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-keepdc]
8       opt_merge [-share_all] -nomux
9
10      do
11          opt_muxtree
12          opt_reduce [-fine] [-full]
13          opt_merge [-share_all]
14          opt_share   (-full only)
15          opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat]   (except when called with -noff)
16          opt_clean [-purge]
17          opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-keepdc]
18      while <changed design>
19
20  When called with -fast the following script is used instead:
21
22      do
23          opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-keepdc]
24          opt_merge [-share_all]
25          opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat]   (except when called with -noff)
26          opt_clean [-purge]
27      while <changed design in opt_dff>
28
29  Note: Options in square brackets (such as [-keepdc]) are passed through to
30  the opt_* commands when given to 'opt'.
```

C.124 opt_clean – remove unused cells and wires

```
1      opt_clean [options] [selection]
2
3   This pass identifies wires and cells that are unused and removes them. Other
4   passes often remove cells but leave the wires in the design or reconnect the
5   wires but leave the old cells in the design. This pass can be used to clean up
6   after the passes that do the actual work.
7
8   This pass only operates on completely selected modules without processes.
9
10      -purge
11          also remove internal nets if they have a public name
```

C.125 opt_demorgan – Optimize reductions with DeMorgan equivalents

```
1        opt_demorgan [selection]
2
3   This pass pushes inverters through $reduce_* cells if this will reduce the
4   overall gate count of the circuit
```

C.126   opt_dff – perform DFF optimizations

```
1        opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat] [selection]
2
3   This pass converts flip-flops to a more suitable type by merging clock enables
4   and synchronous reset multiplexers, removing unused control inputs, or
5   potentially removes the flip-flop altogether, converting it to a constant
6   driver.
7
8       -nodffe
9           disables dff -> dffe conversion, and other transforms recognizing clock
10          enable
11
12      -nosdff
13          disables dff -> sdff conversion, and other transforms recognizing sync
14          resets
15
16      -simple-dffe
17          only enables clock enable recognition transform for obvious cases
18
19      -sat
20          additionally invoke SAT solver to detect and remove flip-flops (with
21          non-constant inputs) that can also be replaced with a constant driver
22
23      -keepdc
24          some optimizations change the behavior of the circuit with respect to
25          don't-care bits. for example in 'a+0' a single x-bit in 'a' will cause
26          all result bits to be set to x. this behavior changes when 'a+0' is
27          replaced by 'a'. the -keepdc option disables all such optimizations.
```

C.127   opt_expr – perform const folding and simple expression rewriting

```
1        opt_expr [options] [selection]
2
3   This pass performs const folding on internal cell types with constant inputs.
4   It also performs some simple expression rewriting.
5
6       -mux_undef
7           remove 'undef' inputs from $mux, $pmux and $_MUX_ cells
8
9       -mux_bool
10          replace $mux cells with inverters or buffers when possible
```

```
11
12      -undriven
13          replace undriven nets with undef (x) constants
14
15      -noclkinv
16          do not optimize clock inverters by changing FF types
17
18      -fine
19          perform fine-grain optimizations
20
21      -full
22          alias for -mux_undef -mux_bool -undriven -fine
23
24      -keepdc
25          some optimizations change the behavior of the circuit with respect to
26          don't-care bits. for example in 'a+0' a single x-bit in 'a' will cause
27          all result bits to be set to x. this behavior changes when 'a+0' is
28          replaced by 'a'. the -keepdc option disables all such optimizations.
```

C.128   opt_ffinv – push inverters through FFs

```
1       opt_ffinv [selection]
2
3  This pass pushes inverters to the other side of a FF when they can be merged
4  into LUTs on the other side.
```

C.129   opt_lut – optimize LUT cells

```
1       opt_lut [options] [selection]
2
3  This pass combines cascaded $lut cells with unused inputs.
4
5      -dlogic <type>:<cell-port>=<LUT-input>[:<cell-port>=<LUT-input>...]
6          preserve connections to dedicated logic cell <type> that has ports
7          <cell-port> connected to LUT inputs <LUT-input>. this includes
8          the case where both LUT and dedicated logic input are connected to
9          the same constant.
10
11      -limit N
12          only perform the first N combines, then stop. useful for debugging.
```

C.130   opt_lut_ins – discard unused LUT inputs

```
1       opt_lut_ins [options] [selection]
2
3  This pass removes unused inputs from LUT cells (that is, inputs that can not
4  influence the output signal given this LUT's value).  While such LUTs cannot
```

```
 5   be directly emitted by ABC , they can be a result of various post - ABC
 6   transformations , such as mapping wide LUTs ( not all sub - LUTs will use the
 7   full set of inputs ) or optimizations such as xilinx_dffopt .
 8
 9       -tech < technology >
10           Instead of generic $lut cells , operate on LUT cells specific
11           to the given technology .  Valid values are : xilinx , ecp5 , gowin .
```

C.131   opt_mem – optimize memories

```
1       opt_mem [ options ] [ selection ]
2
3   This pass performs various optimizations on memories in the design .
```

C.132   opt_mem_feedback – convert memory read-to-write port feedback paths to write enables

```
1       opt_mem_feedback [ selection ]
2
3   This pass detects cases where an asynchronous read port is only connected via
4   a mux tree to a write port with the same address .  When such a connection is
5   found , it is replaced with a new condition on an enable signal , allowing
6   for removal of the read port .
```

C.133   opt_mem_priority – remove priority relations between write ports that can never collide

```
1       opt_mem_priority [ selection ]
2
3   This pass detects cases where one memory write port has priority over another
4   even though they can never collide with each other -- ie. there can never be
5   a situation where a given memory bit is written by both ports at the same
6   time , for example because of always - different addresses , or mutually exclusive
7   enable signals . In such cases , the priority relation is removed .
```

C.134   opt_mem_widen – optimize memories where all ports are wide

```
1       opt_mem_widen [ options ] [ selection ]
2
3   This pass looks for memories where all ports are wide and adjusts the base
4   memory width up until that stops being the case .
```

C.135 opt_merge – consolidate identical cells

```
opt_merge [options] [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.

    -share_all
        Operate on all cell types, not just built-in types.

    -keepdc
        Do not merge flipflops with don't-care bits in their initial value.
```

C.136 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.137 opt_reduce – simplify large MUXes and AND/OR gates

```
opt_reduce [options] [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.

    -fine
      perform fine-grain optimizations

    -full
      alias for -fine
```

C.138 opt_share – merge mutually exclusive cells of the same type that share an input signal

```
1       opt_share [selection]
2
3  This pass identifies mutually exclusive cells of the same type that:
4       (a) share an input signal,
5       (b) drive the same $mux, $_MUX_, or $pmux multiplexing cell,
6
7  allowing the cell to be merged and the multiplexer to be moved from
8  multiplexing its output to multiplexing the non-shared input signals.
```

C.139   paramap – renaming cell parameters

```
1       paramap [options] [selection]
2
3  This command renames cell parameters and/or maps key/value pairs to
4  other key/value pairs.
5
6       -tocase <name>
7           Match attribute names case-insensitively and set it to the specified
8           name.
9
10      -rename <old_name> <new_name>
11          Rename attributes as specified
12
13      -map <old_name>=<old_value> <new_name>=<new_value>
14          Map key/value pairs as indicated.
15
16      -imap <old_name>=<old_value> <new_name>=<new_value>
17          Like -map, but use case-insensitive match for <old_value> when
18          it is a string value.
19
20      -remove <name>=<value>
21          Remove attributes matching this pattern.
22
23  For example, mapping Diamond-style ECP5 "init" attributes to Yosys-style:
24
25      paramap -tocase INIT t:LUT4
```

C.140   peepopt – collection of peephole optimizers

```
1       peepopt [options] [selection]
2
3  This pass applies a collection of peephole optimizers to the current design.
```

C.141   plugin – load and list loaded plugins

```
1       plugin [options]
2
3   Load and list loaded plugins.
4
5       -i <plugin_filename>
6           Load (install) the specified plugin.
7
8       -a <alias_name>
9           Register the specified alias name for the loaded plugin
10
11      -l
12          List loaded plugins
```

C.142   pmux2shiftx – transform $pmux cells to $shiftx cells

```
1       pmux2shiftx [options] [selection]
2
3   This pass transforms $pmux cells to $shiftx cells.
4
5       -v, -vv
6           verbose output
7
8       -min_density <percentage>
9           specifies the minimum density for the shifter
10          default: 50
11
12      -min_choices <int>
13          specified the minimum number of choices for a control signal
14          default: 3
15
16      -onehot ignore|pmux|shiftx
17          select strategy for one-hot encoded control signals
18          default: pmux
19
20      -norange
21          disable $sub inference for "range decoders"
```

C.143   pmuxtree – transform $pmux cells to trees of $mux cells

```
1       pmuxtree [selection]
2
3   This pass transforms $pmux cells to trees of $mux cells.
```

C.144   portlist – list (top-level) ports

```
1       portlist [options] [selection]
2
3  This command lists all module ports found in the selected modules.
4
5  If no selection is provided then it lists the ports on the top module.
6
7    -m
8      print verilog blackbox module definitions instead of port lists
```

C.145 prep – generic synthesis script

```
1       prep [options]
2
3  This command runs a conservative RTL synthesis. A typical application for this
4  is the preparation stage of a verification flow. This command does not operate
5  on partly selected designs.
6
7      -top <module>
8          use the specified module as top module (default='top')
9
10     -auto-top
11         automatically determine the top of the design hierarchy
12
13     -flatten
14         flatten the design before synthesis. this will pass '-auto-top' to
15         'hierarchy' if no top module is specified.
16
17     -ifx
18         passed to 'proc'. uses verilog simulation behavior for verilog if/case
19         undef handling. this also prevents 'wreduce' from being run.
20
21     -memx
22         simulate verilog simulation behavior for out-of-bounds memory accesses
23         using the 'memory_memx' pass.
24
25     -nomem
26         do not run any of the memory_* passes
27
28     -rdff
29         call 'memory_dff'. This enables merging of FFs into
30         memory read ports.
31
32     -nokeepdc
33         do not call opt_* with -keepdc
34
35     -run <from_label>[:<to_label>]
36         only run the commands between the labels (see below). an empty
37         from label is synonymous to 'begin', and empty to label is
38         synonymous to the end of the command list.
39
40
```

```
41    The following commands are executed by this synthesis command:
42
43        begin:
44            hierarchy -check [-top <top> | -auto-top]
45
46        coarse:
47            proc [-ifx]
48            flatten    (if -flatten)
49            opt_expr -keepdc
50            opt_clean
51            check
52            opt -noff -keepdc
53            wreduce -keepdc [-memx]
54            memory_dff   (if -rdff)
55            memory_memx   (if -memx)
56            opt_clean
57            memory_collect
58            opt -noff -keepdc -fast
59
60        check:
61            stat
62            check
```

C.146   printattrs – print attributes of selected objects

```
1        printattrs [selection]
2
3    Print all attributes of the selected objects.
```

C.147   proc – translate processes to netlists

```
1        proc [options] [selection]
2
3    This pass calls all the other proc_* passes in the most common order.
4
5        proc_clean
6        proc_rmdead
7        proc_prune
8        proc_init
9        proc_arst
10       proc_rom
11       proc_mux
12       proc_dlatch
13       proc_dff
14       proc_memwr
15       proc_clean
16       opt_expr -keepdc
17
18   This replaces the processes in the design with multiplexers,
19   flip-flops and latches.
```

```
20
21   The following options are supported:
22
23       -nomux
24           Will omit the proc_mux pass.
25
26       -norom
27           Will omit the proc_rom pass.
28
29       -global_arst [!]<netname>
30           This option is passed through to proc_arst.
31
32       -ifx
33           This option is passed through to proc_mux. proc_rmdead is not
34           executed in -ifx mode.
35
36       -noopt
37           Will omit the opt_expr pass.
```

C.148 proc_arst – detect asynchronous resets

```
1       proc_arst [-global_arst [!]<netname>] [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.
6
7       -global_arst [!]<netname>
8           In modules that have a net with the given name, use this net as async
9           reset for registers that have been assign initial values in their
10          declaration ('reg foobar = constant_value;'). Use the '!' modifier for
11          active low reset signals. Note: the frontend stores the default value
12          in the 'init' attribute on the net.
```

C.149 proc_clean – remove empty parts of processes

```
1       proc_clean [options] [selection]
2
3       -quiet
4           do not print any messages.
5
6   This pass removes empty parts of processes and ultimately removes a process
7   if it contains only empty structures.
```

C.150 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.151   proc_dlatch – extract latches from processes

```
1     proc_dlatch [selection]
2
3  This pass identifies latches in the processes and converts them to
4  d-type latches.
```

C.152   proc_init – convert initial block to init attributes

```
1     proc_init [selection]
2
3  This pass extracts the 'init' actions from processes (generated from Verilog
4  'initial' blocks) and sets the initial value to the 'init' attribute on the
5  respective wire.
```

C.153   proc_memwr – extract memory writes from processes

```
1     proc_memwr [selection]
2
3  This pass converts memory writes in processes into $memwr cells.
```

C.154   proc_mux – convert decision trees to multiplexers

```
1     proc_mux [options] [selection]
2
3  This pass converts the decision trees in processes (originating from if-else
4  and case statements) to trees of multiplexer cells.
5
6     -ifx
7         Use Verilog simulation behavior with respect to undef values in
8         'case' expressions and 'if' conditions.
```

C.155   proc_prune – remove redundant assignments

```
1     proc_prune [selection]
2
3  This pass identifies assignments in processes that are always overwritten by
4  a later assignment to the same signal and removes them.
```

C.156   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.157   proc_rom – convert switches to ROMs

```
1      proc_rom [selection]
2
3  This pass converts switches into read-only memories when appropriate.
```

C.158   qbfsat – solve a 2QBF-SAT problem in the circuit

```
1      qbfsat [options] [selection]
2
3  This command solves an "exists-forall" 2QBF-SAT problem defined over the
4  currently selected module. Existentially-quantified variables are declared by
5  assigning a wire "$anyconst". Universally-quantified variables may be
6  explicitly declared by assigning a wire "$allconst", but module inputs will be
7  treated as universally-quantified variables by default.
8
9      -nocleanup
10         Do not delete temporary files and directories. Useful for debugging.
11
12     -dump-final-smt2 <file>
13         Pass the --dump-smt2 option to yosys-smtbmc.
14
15     -assume-outputs
16         Add an "$assume" cell for the conjunction of all one-bit module output
17         wires.
18
19     -assume-negative-polarity
20         When adding $assume cells for one-bit module output wires, assume they
21         are negative polarity signals and should always be low, for example like
22         the miters created with the `miter` command.
23
24     -nooptimize
25         Ignore "\minimize" and "\maximize" attributes, do not emit
26         "(maximize)" or "(minimize)" in the SMT-LIBv2, and generally make no
27         attempt to optimize anything.
28
29     -nobisection
30         If a wire is marked with the "\minimize" or "\maximize" attribute,
31         do not attempt to optimize that value with the default iterated solving
32         and threshold bisection approach. Instead, have yosys-smtbmc emit a
33         "(minimize)" or "(maximize)" command in the SMT-LIBv2 output and
34         hope that the solver supports optimizing quantified bitvector problems.
35
```

```
36        -solver <solver>
37            Use a particular solver. Choose one of: "z3", "yices", and "cvc4".
38            (default: yices)
39
40        -solver-option <name> <value>
41            Set the specified solver option in the SMT-LIBv2 problem file.
42
43        -timeout <value>
44            Set the per-iteration timeout in seconds.
45            (default: no timeout)
46
47        -O0, -O1, -O2
48            Control the use of ABC to simplify the QBF-SAT problem before solving.
49
50        -sat
51            Generate an error if the solver does not return "sat".
52
53        -unsat
54            Generate an error if the solver does not return "unsat".
55
56        -show-smtbmc
57            Print the output from yosys-smtbmc.
58
59        -specialize
60            If the problem is satisfiable, replace each "$anyconst" cell with its
61            corresponding constant value from the model produced by the solver.
62
63        -specialize-from-file <solution file>
64            Do not run the solver, but instead only attempt to replace each
65            "$anyconst" cell in the current module with a constant value provided
66            by the specified file.
67
68        -write-solution <solution file>
69            If the problem is satisfiable, write the corresponding constant value
70            for each "$anyconst" cell from the model produced by the solver to the
71            specified file.
```

C.159   qwp – quadratic wirelength placer

```
1        qwp [options] [selection]
2
3   This command runs quadratic wirelength placement on the selected modules and
4   annotates the cells in the design with 'qwp_position' attributes.
5
6        -ltr
7            Add left-to-right constraints: constrain all inputs on the left border
8            outputs to the right border.
9
10        -alpha
11            Add constraints for inputs/outputs to be placed in alphanumerical
12            order along the y-axis (top-to-bottom).
13
```

```
14      -grid N
15          Number of grid divisions in x- and y-direction. (default=16)
16
17      -dump <html_file_name>
18          Dump a protocol of the placement algorithm to the html file.
19
20      -v
21          Verbose solver output for profiling or debugging
22
23  Note: This implementation of a quadratic wirelength placer uses exact
24  dense matrix operations. It is only a toy-placer for small circuits.
```

C.160   read − load HDL designs

```
1       read {-vlog95|-vlog2k|-sv2005|-sv2009|-sv2012|-sv|-formal} <verilog-file>..
2
3   Load the specified Verilog/SystemVerilog files. (Full SystemVerilog support
4   is only available via Verific.)
5
6   Additional -D<macro>[=<value>] options may be added after the option indicating
7   the language version (and before file names) to set additional verilog defines.
8
9
10      read {-vhdl87|-vhdl93|-vhdl2k|-vhdl2008|-vhdl} <vhdl-file>..
11
12  Load the specified VHDL files. (Requires Verific.)
13
14
15      read {-edif} <edif-file>..
16
17  Load the specified EDIF files. (Requires Verific.)
18
19
20      read {-liberty} <liberty-file>..
21
22  Load the specified Liberty files.
23
24      -lib
25          only create empty blackbox modules
26
27
28      read {-f|-F} <command-file>
29
30  Load and execute the specified command file. (Requires Verific.)
31  Check verific command for more information about supported commands in file.
32
33
34      read -define <macro>[=<value>]..
35
36  Set global Verilog/SystemVerilog defines.
37
38
```

```
39      read -undef <macro>..
40
41  Unset global Verilog/SystemVerilog defines.
42
43
44      read -incdir <directory>
45
46  Add directory to global Verilog/SystemVerilog include directories.
47
48
49      read -verific
50      read -noverific
51
52  Subsequent calls to 'read' will either use or not use Verific. Calling 'read'
53  with -verific will result in an error on Yosys binaries that are built without
54  Verific support. The default is to use Verific if it is available.
```

C.161   read_aiger – read AIGER file

```
1       read_aiger [options] [filename]
2
3   Load module from an AIGER file into the current design.
4
5       -module_name <module_name>
6           name of module to be created (default: <filename>)
7
8       -clk_name <wire_name>
9           if specified, AIGER latches to be transformed into $_DFF_P_ cells
10          clocked by wire of this name. otherwise, $_FF_ cells will be used
11
12      -map <filename>
13          read file with port and latch symbols
14
15      -wideports
16          merge ports that match the pattern 'name[int]' into a single
17          multi-bit port 'name'
18
19      -xaiger
20          read XAIGER extensions
```

C.162   read_blif – read BLIF file

```
1       read_blif [options] [filename]
2
3   Load modules from a BLIF file into the current design.
4
5       -sop
6           Create $sop cells instead of $lut cells
7
8       -wideports
```

```
 9          Merge ports that match the pattern 'name[int]' into a single
10          multi-bit port 'name'.
```

C.163   read_ilang – (deprecated) alias of read_rtlil

```
1  See `help read_rtlil`.
```

C.164   read_json – read JSON file

```
1      read_json [filename]
2
3  Load modules from a JSON file into the current design See "help write_json"
4  for a description of the file format.
```

C.165   read_liberty – read cells from liberty file

```
 1      read_liberty [filename]
 2
 3  Read cells from liberty file as modules into current design.
 4
 5      -lib
 6          only create empty blackbox modules
 7
 8      -wb
 9          mark imported cells as whiteboxes
10
11      -nooverwrite
12          ignore re-definitions of modules. (the default behavior is to
13          create an error message if the existing module is not a blackbox
14          module, and overwrite the existing module if it is  a blackbox module.)
15
16      -overwrite
17          overwrite existing modules with the same name
18
19      -ignore_miss_func
20          ignore cells with missing function specification of outputs
21
22      -ignore_miss_dir
23          ignore cells with a missing or invalid direction
24          specification on a pin
25
26      -ignore_miss_data_latch
27          ignore latches with missing data and/or enable pins
28
29      -setattr <attribute_name>
30          set the specified attribute (to the value 1) on all loaded modules
```

C.166   read_rtlil – read modules from RTLIL file

```
 1      read_rtlil [filename]
 2
 3  Load modules from an RTLIL file to the current design. (RTLIL is a text
 4  representation of a design in yosys's internal format.)
 5
 6      -nooverwrite
 7          ignore re-definitions of modules. (the default behavior is to
 8          create an error message if the existing module is not a blackbox
 9          module, and overwrite the existing module if it is a blackbox module.)
10
11      -overwrite
12          overwrite existing modules with the same name
13
14      -lib
15          only create empty blackbox modules
```

C.167   read_verilog – read modules from Verilog file

```
 1      read_verilog [options] [filename]
 2
 3  Load modules from a Verilog file to the current design. A large subset of
 4  Verilog-2005 is supported.
 5
 6      -sv
 7          enable support for SystemVerilog features. (only a small subset
 8          of SystemVerilog is supported)
 9
10      -formal
11          enable support for SystemVerilog assertions and some Yosys extensions
12          replace the implicit -D SYNTHESIS with -D FORMAL
13
14      -nosynthesis
15          don't add implicit -D SYNTHESIS
16
17      -noassert
18          ignore assert() statements
19
20      -noassume
21          ignore assume() statements
22
23      -norestrict
24          ignore restrict() statements
25
26      -assume-asserts
27          treat all assert() statements like assume() statements
28
29      -assert-assumes
30          treat all assume() statements like assert() statements
31
32      -debug
```

```
33          alias for -dump_ast1 -dump_ast2 -dump_vlog1 -dump_vlog2 -yydebug
34
35     -dump_ast1
36          dump abstract syntax tree (before simplification)
37
38     -dump_ast2
39          dump abstract syntax tree (after simplification)
40
41     -no_dump_ptr
42          do not include hex memory addresses in dump (easier to diff dumps)
43
44     -dump_vlog1
45          dump ast as Verilog code (before simplification)
46
47     -dump_vlog2
48          dump ast as Verilog code (after simplification)
49
50     -dump_rtlil
51          dump generated RTLIL netlist
52
53     -yydebug
54          enable parser debug output
55
56     -nolatches
57          usually latches are synthesized into logic loops
58          this option prohibits this and sets the output to 'x'
59          in what would be the latches hold condition
60
61          this behavior can also be achieved by setting the
62          'nolatches' attribute on the respective module or
63          always block.
64
65     -nomem2reg
66          under certain conditions memories are converted to registers
67          early during simplification to ensure correct handling of
68          complex corner cases. this option disables this behavior.
69
70          this can also be achieved by setting the 'nomem2reg'
71          attribute on the respective module or register.
72
73          This is potentially dangerous. Usually the front-end has good
74          reasons for converting an array to a list of registers.
75          Prohibiting this step will likely result in incorrect synthesis
76          results.
77
78     -mem2reg
79          always convert memories to registers. this can also be
80          achieved by setting the 'mem2reg' attribute on the respective
81          module or register.
82
83     -nomeminit
84          do not infer $meminit cells and instead convert initialized
85          memories to registers directly in the front-end.
86
```

```
 87         -ppdump
 88             dump Verilog code after pre-processor
 89
 90         -nopp
 91             do not run the pre-processor
 92
 93         -nodpi
 94             disable DPI-C support
 95
 96         -noblackbox
 97             do not automatically add a (* blackbox *) attribute to an
 98             empty module.
 99
100         -lib
101             only create empty blackbox modules. This implies -DBLACKBOX.
102             modules with the (* whitebox *) attribute will be preserved.
103             (* lib_whitebox *) will be treated like (* whitebox *).
104
105         -nowb
106             delete (* whitebox *) and (* lib_whitebox *) attributes from
107             all modules.
108
109         -specify
110             parse and import specify blocks
111
112         -noopt
113             don't perform basic optimizations (such as const folding) in the
114             high-level front-end.
115
116         -icells
117             interpret cell types starting with '$' as internal cell types
118
119         -pwires
120             add a wire for each module parameter
121
122         -nooverwrite
123             ignore re-definitions of modules. (the default behavior is to
124             create an error message if the existing module is not a black box
125             module, and overwrite the existing module otherwise.)
126
127         -overwrite
128             overwrite existing modules with the same name
129
130         -defer
131             only read the abstract syntax tree and defer actual compilation
132             to a later 'hierarchy' command. Useful in cases where the default
133             parameters of modules yield invalid or not synthesizable code.
134
135         -noautowire
136             make the default of `default_nettype be "none" instead of "wire".
137
138         -setattr <attribute_name>
139             set the specified attribute (to the value 1) on all loaded modules
140
```

```
141      -Dname[=definition]
142          define the preprocessor symbol 'name' and set its optional value
143          'definition'
144
145      -Idir
146          add 'dir' to the directories which are used when searching include
147          files
148
149  The command 'verilog_defaults' can be used to register default options for
150  subsequent calls to 'read_verilog'.
151
152  Note that the Verilog frontend does a pretty good job of processing valid
153  verilog input, but has not very good error reporting. It generally is
154  recommended to use a simulator (for example Icarus Verilog) for checking
155  the syntax of the code, rather than to rely on read_verilog for that.
156
157  Depending on if read_verilog is run in -formal mode, either the macro
158  SYNTHESIS or FORMAL is defined automatically, unless -nosynthesis is used.
159  In addition, read_verilog always defines the macro YOSYS.
160
161  See the Yosys README file for a list of non-standard Verilog features
162  supported by the Yosys Verilog front-end.
```

C.168   rename – rename object in the design

```
1        rename old_name new_name
2
3   Rename the specified object. Note that selection patterns are not supported
4   by this command.
5
6
7
8        rename -output old_name new_name
9
10  Like above, but also make the wire an output. This will fail if the object is
11  not a wire.
12
13
14       rename -src [selection]
15
16  Assign names auto-generated from the src attribute to all selected wires and
17  cells with private names.
18
19
20       rename -wire [selection] [-suffix <suffix>]
21
22  Assign auto-generated names based on the wires they drive to all selected
23  cells with private names. Ignores cells driving privatly named wires.
24  By default, the cell is named after the wire with the cell type as suffix.
25  The -suffix option can be used to set the suffix to the given string instead.
26
27
```

```
28        rename -enumerate [-pattern <pattern>] [selection]
29
30    Assign short auto-generated names to all selected wires and cells with private
31    names. The -pattern option can be used to set the pattern for the new names.
32    The character % in the pattern is replaced with a integer number. The default
33    pattern is '_%_'.
34
35
36        rename -witness
37
38    Assigns auto-generated names to all $any*/$all* output wires and containing
39    cells that do not have a public name. This ensures that, during formal
40    verification, a solver-found trace can be fully specified using a public
41    hierarchical names.
42
43
44        rename -hide [selection]
45
46    Assign private names (the ones with $-prefix) to all selected wires and cells
47    with public names. This ignores all selected ports.
48
49
50        rename -top new_name
51
52    Rename top module.
53
54
55        rename -scramble-name [-seed <seed>] [selection]
56
57    Assign randomly-generated names to all selected wires and cells. The seed option
58    can be used to change the random number generator seed from the default, but it
59    must be non-zero.
```

C.169   rmports – remove module ports with no connections

```
1        rmports [selection]
2
3    This pass identifies ports in the selected modules which are not used or
4    driven and removes them.
```

C.170   sat – solve a SAT problem in the circuit

```
1        sat [options] [selection]
2
3    This command solves a SAT problem defined over the currently selected circuit
4    and additional constraints passed as parameters.
5
6        -all
7            show all solutions to the problem (this can grow exponentially, use
8            -max <N> instead to get <N> solutions)
```

```
 9
10      -max <N>
11          like -all, but limit number of solutions to <N>
12
13      -enable_undef
14          enable modeling of undef value (aka 'x-bits')
15          this option is implied by -set-def, -set-undef et. cetera
16
17      -max_undef
18          maximize the number of undef bits in solutions, giving a better
19          picture of which input bits are actually vital to the solution.
20
21      -set <signal> <value>
22          set the specified signal to the specified value.
23
24      -set-def <signal>
25          add a constraint that all bits of the given signal must be defined
26
27      -set-any-undef <signal>
28          add a constraint that at least one bit of the given signal is undefined
29
30      -set-all-undef <signal>
31          add a constraint that all bits of the given signal are undefined
32
33      -set-def-inputs
34          add -set-def constraints for all module inputs
35
36      -show <signal>
37          show the model for the specified signal. if no -show option is
38          passed then a set of signals to be shown is automatically selected.
39
40      -show-inputs, -show-outputs, -show-ports
41          add all module (input/output) ports to the list of shown signals
42
43      -show-regs, -show-public, -show-all
44          show all registers, show signals with 'public' names, show all signals
45
46      -ignore_div_by_zero
47          ignore all solutions that involve a division by zero
48
49      -ignore_unknown_cells
50          ignore all cells that can not be matched to a SAT model
51
52  The following options can be used to set up a sequential problem:
53
54      -seq <N>
55          set up a sequential problem with <N> time steps. The steps will
56          be numbered from 1 to N.
57
58          note: for large <N> it can be significantly faster to use
59          -tempinduct-baseonly -maxsteps <N> instead of -seq <N>.
60
61      -set-at <N> <signal> <value>
62      -unset-at <N> <signal>
```

```
 63          set or unset the specified signal to the specified value in the
 64          given timestep. this has priority over a -set for the same signal.
 65
 66      -set-assumes
 67          set all assumptions provided via $assume cells
 68
 69      -set-def-at <N> <signal>
 70      -set-any-undef-at <N> <signal>
 71      -set-all-undef-at <N> <signal>
 72          add undef constraints in the given timestep.
 73
 74      -set-init <signal> <value>
 75          set the initial value for the register driving the signal to the value
 76
 77      -set-init-undef
 78          set all initial states (not set using -set-init) to undef
 79
 80      -set-init-def
 81          do not force a value for the initial state but do not allow undef
 82
 83      -set-init-zero
 84          set all initial states (not set using -set-init) to zero
 85
 86      -dump_vcd <vcd-file-name>
 87          dump SAT model (counter example in proof) to VCD file
 88
 89      -dump_json <json-file-name>
 90          dump SAT model (counter example in proof) to a WaveJSON file.
 91
 92      -dump_cnf <cnf-file-name>
 93          dump CNF of SAT problem (in DIMACS format). in temporal induction
 94          proofs this is the CNF of the first induction step.
 95
 96  The following additional options can be used to set up a proof. If also -seq
 97  is passed, a temporal induction proof is performed.
 98
 99      -tempinduct
100          Perform a temporal induction proof. In a temporal induction proof it is
101          proven that the condition holds forever after the number of time steps
102          specified using -seq.
103
104      -tempinduct-def
105          Perform a temporal induction proof. Assume an initial state with all
106          registers set to defined values for the induction step.
107
108      -tempinduct-baseonly
109          Run only the basecase half of temporal induction (requires -maxsteps)
110
111      -tempinduct-inductonly
112          Run only the induction half of temporal induction
113
114      -tempinduct-skip <N>
115          Skip the first <N> steps of the induction proof.
116
```

```
117          note: this will assume that the base case holds for <N> steps.
118          this must be proven independently with "-tempinduct-baseonly
119          -maxsteps <N>". Use -initsteps if you just want to set a
120          minimal induction length.
121
122      -prove <signal> <value>
123          Attempt to proof that <signal> is always <value>.
124
125      -prove-x <signal> <value>
126          Like -prove, but an undef (x) bit in the lhs matches any value on
127          the right hand side. Useful for equivalence checking.
128
129      -prove-asserts
130          Prove that all asserts in the design hold.
131
132      -prove-skip <N>
133          Do not enforce the prove-condition for the first <N> time steps.
134
135      -maxsteps <N>
136          Set a maximum length for the induction.
137
138      -initsteps <N>
139          Set initial length for the induction.
140          This will speed up the search of the right induction length
141          for deep induction proofs.
142
143      -stepsize <N>
144          Increase the size of the induction proof in steps of <N>.
145          This will speed up the search of the right induction length
146          for deep induction proofs.
147
148      -timeout <N>
149          Maximum number of seconds a single SAT instance may take.
150
151      -verify
152          Return an error and stop the synthesis script if the proof fails.
153
154      -verify-no-timeout
155          Like -verify but do not return an error for timeouts.
156
157      -falsify
158          Return an error and stop the synthesis script if the proof succeeds.
159
160      -falsify-no-timeout
161          Like -falsify but do not return an error for timeouts.
```

C.171   scatter – add additional intermediate nets

```
1       scatter [selection]
2
3  This command adds additional intermediate nets on all cell ports. This is used
4  for testing the correct use of the SigMap helper in passes. If you don't know
```

```
5   what this means: don't worry -- you only need this pass when testing your own
6   extensions to Yosys.
7
8   Use the opt_clean command to get rid of the additional nets.
```

C.172   scc – detect strongly connected components (logic loops)

```
1       scc [options] [selection]
2
3   This command identifies strongly connected components (aka logic loops) in the
4   design.
5
6       -expect <num>
7           expect to find exactly <num> SCCs. A different number of SCCs will
8           produce an error.
9
10      -max_depth <num>
11          limit to loops not longer than the specified number of cells. This
12          can e.g. be useful in identifying small local loops in a module that
13          implements one large SCC.
14
15      -nofeedback
16          do not count cells that have their output fed back into one of their
17          inputs as single-cell scc.
18
19      -all_cell_types
20          Usually this command only considers internal non-memory cells. With
21          this option set, all cells are considered. For unknown cells all ports
22          are assumed to be bidirectional 'inout' ports.
23
24      -set_attr <name> <value>
25          set the specified attribute on all cells that are part of a logic
26          loop. the special token {} in the value is replaced with a unique
27          identifier for the logic loop.
28
29      -select
30          replace the current selection with a selection of all cells and wires
31          that are part of a found logic loop
32
33      -specify
34          examine specify rules to detect logic loops in whitebox/blackbox cells
```

C.173   scratchpad – get/set values in the scratchpad

```
1       scratchpad [options]
2
3   This pass allows to read and modify values from the scratchpad of the current
4   design. Options:
5
6       -get <identifier>
```

```
 7          print the value saved in the scratchpad under the given identifier.
 8
 9      -set <identifier> <value>
10          save the given value in the scratchpad under the given identifier.
11
12      -unset <identifier>
13          remove the entry for the given identifier from the scratchpad.
14
15      -copy <identifier_from> <identifier_to>
16          copy the value of the first identifier to the second identifier.
17
18      -assert <identifier> <value>
19          assert that the entry for the given identifier is set to the given
20          value.
21
22      -assert-set <identifier>
23          assert that the entry for the given identifier exists.
24
25      -assert-unset <identifier>
26          assert that the entry for the given identifier does not exist.
27
28  The identifier may not contain whitespace. By convention, it is usually prefixed
29  by the name of the pass that uses it, e.g. 'opt.did_something'. If the value
30  contains whitespace, it must be enclosed in double quotes.
```

C.174   script – execute commands from file or wire

```
 1      script <filename> [<from_label>:<to_label>]
 2      script -scriptwire [selection]
 3
 4  This command executes the yosys commands in the specified file (default
 5  behaviour), or commands embedded in the constant text value connected to the
 6  selected wires.
 7
 8  In the default (file) case, the 2nd argument can be used to only execute the
 9  section of the file between the specified labels. An empty from label is
10  synonymous with the beginning of the file and an empty to label is synonymous
11  with the end of the file.
12
13  If only one label is specified (without ':') then only the block
14  marked with that label (until the next label) is executed.
15
16  In "-scriptwire" mode, the commands on the selected wire(s) will be executed
17  in the scope of (and thus, relative to) the wires' owning module(s). This
18  '-module' mode can be exited by using the 'cd' command.
```

C.175   select – modify and view the list of selected objects

```
 1      select [ -add | -del | -set <name> ] {-read <filename> | <selection>}
 2      select [ -unset <name> ]
```

```
   3        select [ <assert_option> ] {-read <filename> | <selection>}
   4        select [ -list | -write <filename> | -count | -clear ]
   5        select -module <modname>
   6
   7  Most commands use the list of currently selected objects to determine which part
   8  of the design to operate on. This command can be used to modify and view this
   9  list of selected objects.
  10
  11  Note that many commands support an optional [selection] argument that can be
  12  used to override the global selection for the command. The syntax of this
  13  optional argument is identical to the syntax of the <selection> argument
  14  described here.
  15
  16      -add, -del
  17          add or remove the given objects to the current selection.
  18          without this options the current selection is replaced.
  19
  20      -set <name>
  21          do not modify the current selection. instead save the new selection
  22          under the given name (see @<name> below). to save the current selection,
  23          use "select -set <name> %"
  24
  25      -unset <name>
  26          do not modify the current selection. instead remove a previously saved
  27          selection under the given name (see @<name> below).
  28
  29      -assert-none
  30          do not modify the current selection. instead assert that the given
  31          selection is empty. i.e. produce an error if any object or module
  32          matching the selection is found.
  33
  34      -assert-any
  35          do not modify the current selection. instead assert that the given
  36          selection is non-empty. i.e. produce an error if no object or module
  37          matching the selection is found.
  38
  39      -assert-count N
  40          do not modify the current selection. instead assert that the given
  41          selection contains exactly N objects.
  42
  43      -assert-max N
  44          do not modify the current selection. instead assert that the given
  45          selection contains less than or exactly N objects.
  46
  47      -assert-min N
  48          do not modify the current selection. instead assert that the given
  49          selection contains at least N objects.
  50
  51      -list
  52          list all objects in the current selection
  53
  54      -write <filename>
  55          like -list but write the output to the specified file
  56
```

```
 57        -read <filename>
 58            read the specified file (written by -write)
 59
 60        -count
 61            count all objects in the current selection
 62
 63        -clear
 64            clear the current selection. this effectively selects the whole
 65            design. it also resets the selected module (see -module). use the
 66            command 'select *' to select everything but stay in the current module.
 67
 68        -none
 69            create an empty selection. the current module is unchanged.
 70
 71        -module <modname>
 72            limit the current scope to the specified module.
 73            the difference between this and simply selecting the module
 74            is that all object names are interpreted relative to this
 75            module after this command until the selection is cleared again.
 76
 77  When this command is called without an argument, the current selection
 78  is displayed in a compact form (i.e. only the module name when a whole module
 79  is selected).
 80
 81  The <selection> argument itself is a series of commands for a simple stack
 82  machine. Each element on the stack represents a set of selected objects.
 83  After this commands have been executed, the union of all remaining sets
 84  on the stack is computed and used as selection for the command.
 85
 86  Pushing (selecting) object when not in -module mode:
 87
 88        <mod_pattern>
 89            select the specified module(s)
 90
 91        <mod_pattern>/<obj_pattern>
 92            select the specified object(s) from the module(s)
 93
 94  Pushing (selecting) object when in -module mode:
 95
 96        <obj_pattern>
 97            select the specified object(s) from the current module
 98
 99  By default, patterns will not match black/white-box modules or their
100  contents. To include such objects, prefix the pattern with '='.
101
102  A <mod_pattern> can be a module name, wildcard expression (*, ?, [..])
103  matching module names, or one of the following:
104
105        A:<pattern>, A:<pattern>=<pattern>
106            all modules with an attribute matching the given pattern
107            in addition to = also <, <=, >=, and > are supported
108
109        N:<pattern>
110            all modules with a name matching the given pattern
```

174

```
111            (i.e. 'N:' is optional as it is the default matching rule)
112
113  An <obj_pattern> can be an object name, wildcard expression, or one of
114  the following:
115
116      w:<pattern>
117          all wires with a name matching the given wildcard pattern
118
119      i:<pattern>, o:<pattern>, x:<pattern>
120          all inputs (i:), outputs (o:) or any ports (x:) with matching names
121
122      s:<size>, s:<min>:<max>
123          all wires with a matching width
124
125      m:<pattern>
126          all memories with a name matching the given pattern
127
128      c:<pattern>
129          all cells with a name matching the given pattern
130
131      t:<pattern>
132          all cells with a type matching the given pattern
133
134      p:<pattern>
135          all processes with a name matching the given pattern
136
137      a:<pattern>
138          all objects with an attribute name matching the given pattern
139
140      a:<pattern>=<pattern>
141          all objects with a matching attribute name-value-pair.
142          in addition to = also <, <=, >=, and > are supported
143
144      r:<pattern>, r:<pattern>=<pattern>
145          cells with matching parameters. also with <, <=, >= and >.
146
147      n:<pattern>
148          all objects with a name matching the given pattern
149          (i.e. 'n:' is optional as it is the default matching rule)
150
151      @<name>
152          push the selection saved prior with 'select -set <name> ...'
153
154  The following actions can be performed on the top sets on the stack:
155
156      %
157          push a copy of the current selection to the stack
158
159      %%
160          replace the stack with a union of all elements on it
161
162      %n
163          replace top set with its invert
164
```

175

```
165     %u
166         replace the two top sets on the stack with their union
167
168     %i
169         replace the two top sets on the stack with their intersection
170
171     %d
172         pop the top set from the stack and subtract it from the new top
173
174     %D
175         like %d but swap the roles of two top sets on the stack
176
177     %c
178         create a copy of the top set from the stack and push it
179
180     %x[<num1>|*][.<num2>][:<rule>[:<rule>..]]
181         expand top set <num1> num times according to the specified rules.
182         (i.e. select all cells connected to selected wires and select all
183         wires connected to selected cells) The rules specify which cell
184         ports to use for this. the syntax for a rule is a '-' for exclusion
185         and a '+' for inclusion, followed by an optional comma separated
186         list of cell types followed by an optional comma separated list of
187         cell ports in square brackets. a rule can also be just a cell or wire
188         name that limits the expansion (is included but does not go beyond).
189         select at most <num2> objects. a warning message is printed when this
190         limit is reached. When '*' is used instead of <num1> then the process
191         is repeated until no further object are selected.
192
193     %ci[<num1>|*][.<num2>][:<rule>[:<rule>..]]
194     %co[<num1>|*][.<num2>][:<rule>[:<rule>..]]
195         similar to %x, but only select input (%ci) or output cones (%co)
196
197     %xe[...] %cie[...] %coe
198         like %x, %ci, and %co but only consider combinatorial cells
199
200     %a
201         expand top set by selecting all wires that are (at least in part)
202         aliases for selected wires.
203
204     %s
205         expand top set by adding all modules that implement cells in selected
206         modules
207
208     %m
209         expand top set by selecting all modules that contain selected objects
210
211     %M
212         select modules that implement selected cells
213
214     %C
215         select cells that implement selected modules
216
217     %R[<num>]
218         select <num> random objects from top selection (default 1)
```

```
219
220  Example: the following command selects all wires that are connected to a
221  'GATE' input of a 'SWITCH' cell:
222
223      select */t:SWITCH %x:+[GATE] */t:SWITCH %d
```

C.176   setattr – set/unset attributes on objects

```
1      setattr [ -mod ] [ -set name value | -unset name ]... [selection]
2
3  Set/unset the given attributes on the selected objects. String values must be
4  passed in double quotes (").
5
6  When called with -mod, this command will set and unset attributes on modules
7  instead of objects within modules.
```

C.177   setparam – set/unset parameters on objects

```
1      setparam [ -type cell_type ] [ -set name value | -unset name ]... [selection]
2
3  Set/unset the given parameters on the selected cells. String values must be
4  passed in double quotes (").
5
6  The -type option can be used to change the cell type of the selected cells.
```

C.178   setundef – replace undef values with defined constants

```
1      setundef [options] [selection]
2
3  This command replaces undef (x) constants with defined (0/1) constants.
4
5      -undriven
6          also set undriven nets to constant values
7
8      -expose
9          also expose undriven nets as inputs (use with -undriven)
10
11     -zero
12         replace with bits cleared (0)
13
14     -one
15         replace with bits set (1)
16
17     -undef
18         replace with undef (x) bits, may be used with -undriven
19
20     -anyseq
```

```
21          replace with $anyseq drivers (for formal)
22
23      -anyconst
24          replace with $anyconst drivers (for formal)
25
26      -random <seed>
27          replace with random bits using the specified integer as seed
28          value for the random number generator.
29
30      -init
31          also create/update init values for flip-flops
32
33      -params
34          replace undef in cell parameters
```

C.179   share – perform sat-based resource sharing

```
1       share [options] [selection]
2
3  This pass merges shareable resources into a single resource. A SAT solver
4  is used to determine if two resources are share-able.
5
6    -force
7      Per default the selection of cells that is considered for sharing is
8      narrowed using a list of cell types. With this option all selected
9      cells are considered for resource sharing.
10
11     IMPORTANT NOTE: If the -all option is used then no cells with internal
12     state must be selected!
13
14   -aggressive
15     Per default some heuristics are used to reduce the number of cells
16     considered for resource sharing to only large resources. This options
17     turns this heuristics off, resulting in much more cells being considered
18     for resource sharing.
19
20   -fast
21     Only consider the simple part of the control logic in SAT solving, resulting
22     in much easier SAT problems at the cost of maybe missing some opportunities
23     for resource sharing.
24
25   -limit N
26     Only perform the first N merges, then stop. This is useful for debugging.
```

C.180   shell – enter interactive command mode

```
1       shell
2
3  This command enters the interactive command mode. This can be useful
4  in a script to interrupt the script at a certain point and allow for
```

```
 5   interactive inspection or manual synthesis of the design at this point.
 6
 7   The command prompt of the interactive shell indicates the current
 8   selection (see 'help select'):
 9
10       yosys>
11           the entire design is selected
12
13       yosys*>
14           only part of the design is selected
15
16       yosys [modname]>
17           the entire module 'modname' is selected using 'select -module modname'
18
19       yosys [modname]*>
20           only part of current module 'modname' is selected
21
22   When in interactive shell, some errors (e.g. invalid command arguments)
23   do not terminate yosys but return to the command prompt.
24
25   This command is the default action if nothing else has been specified
26   on the command line.
27
28   Press Ctrl-D or type 'exit' to leave the interactive shell.
```

C.181   show – generate schematics using graphviz

```
 1       show [options] [selection]
 2
 3   Create a graphviz DOT file for the selected part of the design and compile it
 4   to a graphics file (usually SVG or PostScript).
 5
 6       -viewer <viewer>
 7           Run the specified command with the graphics file as parameter.
 8           On Windows, this pauses yosys until the viewer exits.
 9
10       -format <format>
11           Generate a graphics file in the specified format. Use 'dot' to just
12           generate a .dot file, or other <format> strings such as 'svg' or 'ps'
13           to generate files in other formats (this calls the 'dot' command).
14
15       -lib <verilog_or_rtlil_file>
16           Use the specified library file for determining whether cell ports are
17           inputs or outputs. This option can be used multiple times to specify
18           more than one library.
19
20           note: in most cases it is better to load the library before calling
21           show with 'read_verilog -lib <filename>'. it is also possible to
22           load liberty files with 'read_liberty -lib <filename>'.
23
24       -prefix <prefix>
25           generate <prefix>.* instead of ~/.yosys_show.*
```

```
 26
 27      -color <color> <object>
 28          assign the specified color to the specified object. The object can be
 29          a single selection wildcard expressions or a saved set of objects in
 30          the @<name> syntax (see "help select" for details).
 31
 32      -label <text> <object>
 33          assign the specified label text to the specified object. The object can
 34          be a single selection wildcard expressions or a saved set of objects in
 35          the @<name> syntax (see "help select" for details).
 36
 37      -colors <seed>
 38          Randomly assign colors to the wires. The integer argument is the seed
 39          for the random number generator. Change the seed value if the colored
 40          graph still is ambiguous. A seed of zero deactivates the coloring.
 41
 42      -colorattr <attribute_name>
 43          Use the specified attribute to assign colors. A unique color is
 44          assigned to each unique value of this attribute.
 45
 46      -width
 47          annotate buses with a label indicating the width of the bus.
 48
 49      -signed
 50          mark ports (A, B) that are declared as signed (using the [AB]_SIGNED
 51          cell parameter) with an asterisk next to the port name.
 52
 53      -stretch
 54          stretch the graph so all inputs are on the left side and all outputs
 55          (including inout ports) are on the right side.
 56
 57      -pause
 58          wait for the user to press enter to before returning
 59
 60      -enum
 61          enumerate objects with internal ($-prefixed) names
 62
 63      -long
 64          do not abbreviate objects with internal ($-prefixed) names
 65
 66      -notitle
 67          do not add the module name as graph title to the dot file
 68
 69      -nobg
 70          don't run viewer in the background, IE wait for the viewer tool to
 71          exit before returning
 72
 73  When no <format> is specified, 'dot' is used. When no <format> and <viewer> is
 74  specified, 'xdot' is used to display the schematic (POSIX systems only).
 75
 76  The generated output files are '~/.yosys_show.dot' and '~/.yosys_show.<format>',
 77  unless another prefix is specified using -prefix <prefix>.
 78
 79  Yosys on Windows and YosysJS use different defaults: The output is written
```

```
80 │ to 'show.dot' in the current directory and new viewer is launched each time
81 │ the 'show' command is executed.
```

C.182   shregmap – map shift registers

```
 1 │     shregmap [options] [selection]
 2 │
 3 │ This pass converts chains of $_DFF_[NP]_ gates to target specific shift register
 4 │ primitives. The generated shift register will be of type $__SHREG_DFF_[NP]_ and
 5 │ will use the same interface as the original $_DFF_*_ cells. The cell parameter
 6 │ 'DEPTH' will contain the depth of the shift register. Use a target-specific
 7 │ 'techmap' map file to convert those cells to the actual target cells.
 8 │
 9 │     -minlen N
10 │         minimum length of shift register (default = 2)
11 │         (this is the length after -keep_before and -keep_after)
12 │
13 │     -maxlen N
14 │         maximum length of shift register (default = no limit)
15 │         larger chains will be mapped to multiple shift register instances
16 │
17 │     -keep_before N
18 │         number of DFFs to keep before the shift register (default = 0)
19 │
20 │     -keep_after N
21 │         number of DFFs to keep after the shift register (default = 0)
22 │
23 │     -clkpol pos|neg|any
24 │         limit match to only positive or negative edge clocks. (default = any)
25 │
26 │     -enpol pos|neg|none|any_or_none|any
27 │         limit match to FFs with the specified enable polarity. (default = none)
28 │
29 │     -match <cell_type>[:<d_port_name>:<q_port_name>]
30 │         match the specified cells instead of $_DFF_N_ and $_DFF_P_. If
31 │         ':<d_port_name>:<q_port_name>' is omitted then 'D' and 'Q' is used
32 │         by default. E.g. the option '-clkpol pos' is just an alias for
33 │         '-match $_DFF_P_', which is an alias for '-match $_DFF_P_:D:Q'.
34 │
35 │     -params
36 │         instead of encoding the clock and enable polarity in the cell name by
37 │         deriving from the original cell name, simply name all generated cells
38 │         $__SHREG_ and use CLKPOL and ENPOL parameters. An ENPOL value of 2 is
39 │         used to denote cells without enable input. The ENPOL parameter is
40 │         omitted when '-enpol none' (or no -enpol option) is passed.
41 │
42 │     -zinit
43 │         assume the shift register is automatically zero-initialized, so it
44 │         becomes legal to merge zero initialized FFs into the shift register.
45 │
46 │     -init
47 │         map initialized registers to the shift reg, add an INIT parameter to
```

```
48         generated cells with the initialization value. (first bit to shift out
49         in LSB position)
50
51     -tech greenpak4
52         map to greenpak4 shift registers.
```

C.183   sim − simulate the circuit

```
 1     sim [options] [top-level]
 2
 3 This command simulates the circuit using the given top-level module.
 4
 5     -vcd <filename>
 6         write the simulation results to the given VCD file
 7
 8     -fst <filename>
 9         write the simulation results to the given FST file
10
11     -aiw <filename>
12         write the simulation results to an AIGER witness file
13         (requires a *.aim file via -map)
14
15     -hdlname
16         use the hdlname attribute when writing simulation results
17         (preserves hierarchy in a flattened design)
18
19     -x
20         ignore constant x outputs in simulation file.
21
22     -date
23         include date and full version info in output.
24
25     -clock <portname>
26         name of top-level clock input
27
28     -clockn <portname>
29         name of top-level clock input (inverse polarity)
30
31     -multiclock
32         mark that witness file is multiclock.
33
34     -reset <portname>
35         name of top-level reset input (active high)
36
37     -resetn <portname>
38         name of top-level inverted reset input (active low)
39
40     -rstlen <integer>
41         number of cycles reset should stay active (default: 1)
42
43     -zinit
44         zero-initialize all uninitialized regs and memories
```

```
45

46      -timescale <string>
47          include the specified timescale declaration in the vcd
48

49      -n <integer>
50          number of clock cycles to simulate (default: 20)
51

52      -a
53          use all nets in VCD/FST operations, not just those with public names
54

55      -w
56          writeback mode: use final simulation state as new init state
57

58      -r
59          read simulation results file
60              File formats supported: FST, VCD, AIW and WIT
61                  VCD support requires vcd2fst external tool to be present
62

63      -map <filename>
64          read file with port and latch symbols, needed for AIGER witness input
65

66      -scope <name>
67          scope of simulation top model
68

69      -at <time>
70          sets start and stop time
71

72      -start <time>
73          start co-simulation in arbitary time (default 0)
74

75      -stop <time>
76          stop co-simulation in arbitary time (default END)
77

78      -sim
79          simulation with stimulus from FST (default)
80

81      -sim-cmp
82          co-simulation expect exact match
83

84      -sim-gold
85          co-simulation, x in simulation can match any value in FST
86

87      -sim-gate
88          co-simulation, x in FST can match any value in simulation
89

90      -q
91          disable per-cycle/sample log message
92

93      -d
94          enable debug output
```

C.184   simplemap – mapping simple coarse-grain cells

```
     simplemap [selection]

This pass maps a small selection of simple coarse-grain cells to yosys gate
primitives. The following internal cell types are mapped by this pass:

  $not, $pos, $and, $or, $xor, $xnor
  $reduce_and, $reduce_or, $reduce_xor, $reduce_xnor, $reduce_bool
  $logic_not, $logic_and, $logic_or, $mux, $tribuf
  $sr, $ff, $dff, $dffe, $dffsr, $dffsre, $adff, $adffe, $aldff, $aldffe, $sdff,
  $sdffe, $sdffce, $dlatch, $adlatch, $dlatchsr
```

C.185   splice – create explicit splicing cells

```
     splice [options] [selection]

This command adds $slice and $concat cells to the design to make the splicing
of multi-bit signals explicit. This for example is useful for coarse grain
synthesis, where dedicated hardware is needed to splice signals.

    -sel_by_cell
        only select the cell ports to rewire by the cell. if the selection
        contains a cell, than all cell inputs are rewired, if necessary.

    -sel_by_wire
        only select the cell ports to rewire by the wire. if the selection
        contains a wire, than all cell ports driven by this wire are wired,
        if necessary.

    -sel_any_bit
        it is sufficient if the driver of any bit of a cell port is selected.
        by default all bits must be selected.

    -wires
        also add $slice and $concat cells to drive otherwise unused wires.

    -no_outputs
        do not rewire selected module outputs.

    -port <name>
        only rewire cell ports with the specified name. can be used multiple
        times. implies -no_output.

    -no_port <name>
        do not rewire cell ports with the specified name. can be used multiple
        times. can not be combined with -port <name>.

By default selected output wires and all cell ports of selected cells driven
by selected wires are rewired.
```

C.186   splitnets – split up multi-bit nets

```
 1      splitnets [options] [selection]
 2
 3  This command splits multi-bit nets into single-bit nets.
 4
 5      -format char1[char2[char3]]
 6          the first char is inserted between the net name and the bit index, the
 7          second char is appended to the netname. e.g. -format () creates net
 8          names like 'mysignal(42)'. the 3rd character is the range separation
 9          character when creating multi-bit wires. the default is '[]:'.
10
11      -ports
12          also split module ports. per default only internal signals are split.
13
14      -driver
15          don't blindly split nets in individual bits. instead look at the driver
16          and split nets so that no driver drives only part of a net.
```

C.187   sta – perform static timing analysis

```
 1      sta [options] [selection]
 2
 3  This command performs static timing analysis on the design. (Only considers
 4  paths within a single module, so the design must be flattened.)
```

C.188   stat – print some statistics

```
 1      stat [options] [selection]
 2
 3  Print some statistics (number of objects) on the selected portion of the
 4  design.
 5
 6      -top <module>
 7          print design hierarchy with this module as top. if the design is fully
 8          selected and a module has the 'top' attribute set, this module is used
 9          default value for this option.
10
11      -liberty <liberty_file>
12          use cell area information from the provided liberty file
13
14      -tech <technology>
15          print area estemate for the specified technology. Currently supported
16          values for <technology>: xilinx, cmos
17
18      -width
19          annotate internal cell types with their word width.
20          e.g. $add_8 for an 8 bit wide $add cell.
21
```

```
22        -json
23            output the statistics in a machine-readable JSON format.
24            this is output to the console; use "tee" to output to a file.
```

C.189   submod – moving part of a module to a new submodule

```
 1        submod [options] [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       -copy
14           by default the cells are 'moved' from the source module and the source
15           module will use an instance of the new module after this command is
16           finished. call with -copy to not modify the source module.
17
18       -name <name>
19           don't use the 'submod' attribute but instead use the selection. only
20           objects from one module might be selected. the value of the -name option
21           is used as the value of the 'submod' attribute instead.
22
23       -hidden
24           instead of creating submodule ports with public names, create ports with
25           private names so that a subsequent 'flatten; clean' call will restore
26           the original module with original public names.
```

C.190   supercover – add hi/lo cover cells for each wire bit

```
 1        supercover [options] [selection]
 2
 3   This command adds two cover cells for each bit of each selected wire, one
 4   checking for a hi signal level and one checking for lo level.
```

C.191   synth – generic synthesis script

```
 1        synth [options]
 2
 3   This command runs the default synthesis script. This command does not operate
 4   on partly selected designs.
 5
```

```
 6        -top <module>
 7            use the specified module as top module (default='top')
 8
 9        -auto-top
10            automatically determine the top of the design hierarchy
11
12        -flatten
13            flatten the design before synthesis. this will pass '-auto-top' to
14            'hierarchy' if no top module is specified.
15
16        -encfile <file>
17            passed to 'fsm_recode' via 'fsm'
18
19        -lut <k>
20            perform synthesis for a k-LUT architecture.
21
22        -nofsm
23            do not run FSM optimization
24
25        -noabc
26            do not run abc (as if yosys was compiled without ABC support)
27
28        -noalumacc
29            do not run 'alumacc' pass. i.e. keep arithmetic operators in
30            their direct form ($add, $sub, etc.).
31
32        -nordff
33            passed to 'memory'. prohibits merging of FFs into memory read ports
34
35        -noshare
36            do not run SAT-based resource sharing
37
38        -run <from_label>[:<to_label>]
39            only run the commands between the labels (see below). an empty
40            from label is synonymous to 'begin', and empty to label is
41            synonymous to the end of the command list.
42
43        -abc9
44            use new ABC9 flow (EXPERIMENTAL)
45
46        -flowmap
47            use FlowMap LUT techmapping instead of ABC
48
49        -no-rw-check
50            marks all recognized read ports as "return don't-care value on
51            read/write collision" (same result as setting the no_rw_check
52            attribute on all memories).
53
54
55    The following commands are executed by this synthesis command:
56
57        begin:
58            hierarchy -check [-top <top> | -auto-top]
59
```

```
60      coarse:
61          proc
62          flatten      (if -flatten)
63          opt_expr
64          opt_clean
65          check
66          opt -nodffe -nosdff
67          fsm          (unless -nofsm)
68          opt
69          wreduce
70          peepopt
71          opt_clean
72          techmap -map +/cmp2lut.v -map +/cmp2lcu.v      (if -lut)
73          alumacc      (unless -noalumacc)
74          share        (unless -noshare)
75          opt
76          memory -nomap
77          opt_clean
78
79      fine:
80          opt -fast -full
81          memory_map
82          opt -full
83          techmap
84          techmap -map +/gate2lut.v    (if -noabc and -lut)
85          clean; opt_lut               (if -noabc and -lut)
86          flowmap -maxlut K            (if -flowmap and -lut)
87          opt -fast
88          abc -fast             (unless -noabc, unless -lut)
89          abc -fast -lut k      (unless -noabc, if -lut)
90          opt -fast             (unless -noabc)
91
92      check:
93          hierarchy -check
94          stat
95          check
```

C.192 synth_achronix – synthesis for Acrhonix Speedster22i FP-GAs.

```
1       synth_achronix [options]
2
3  This command runs synthesis for Achronix Speedster eFPGAs. This work is still experimental.
4
5      -top <module>
6          use the specified module as top module (default='top')
7
8      -vout <file>
9          write the design to the specified Verilog netlist file. writing of an
10         output file is omitted if this parameter is not specified.
11
12     -run <from_label>:<to_label>
```

```
13          only run the commands between the labels (see below). an empty
14          from label is synonymous to 'begin', and empty to label is
15          synonymous to the end of the command list.
16
17      -noflatten
18          do not flatten design before synthesis
19
20      -retime
21          run 'abc' with '-dff -D 1' options
22
23
24  The following commands are executed by this synthesis command:
25
26      begin:
27          read_verilog -sv -lib +/achronix/speedster22i/cells_sim.v
28          hierarchy -check -top <top>
29
30      flatten:    (unless -noflatten)
31          proc
32          flatten
33          tribuf -logic
34          deminout
35
36      coarse:
37          synth -run coarse
38
39      fine:
40          opt -fast -mux_undef -undriven -fine -full
41          memory_map
42          opt -undriven -fine
43          opt -fine
44          techmap -map +/techmap.v
45          opt -full
46          clean -purge
47          setundef -undriven -zero
48          dfflegalize -cell $_DFF_P_ x
49          abc -markgroups -dff -D 1    (only if -retime)
50
51      map_luts:
52          abc -lut 4
53          clean
54
55      map_cells:
56          iopadmap -bits -outpad $__outpad I:O -inpad $__inpad O:I
57          techmap -map +/achronix/speedster22i/cells_map.v
58          clean -purge
59
60      check:
61          hierarchy -check
62          stat
63          check -noinit
64          blackbox =A:whitebox
65
66      vout:
```

```
67         write_verilog -nodec -attr2comment -defparam -renameprefix syn_ <file-name>
```

C.193 synth_anlogic – synthesis for Anlogic FPGAs

```
 1      synth_anlogic [options]
 2
 3  This command runs synthesis for Anlogic FPGAs.
 4
 5      -top <module>
 6          use the specified module as top module
 7
 8      -edif <file>
 9          write the design to the specified EDIF file. writing of an output file
10          is omitted if this parameter is not specified.
11
12      -json <file>
13          write the design to the specified JSON file. writing of an output file
14          is omitted if this parameter is not specified.
15
16      -run <from_label>:<to_label>
17          only run the commands between the labels (see below). an empty
18          from label is synonymous to 'begin', and empty to label is
19          synonymous to the end of the command list.
20
21      -noflatten
22          do not flatten design before synthesis
23
24      -retime
25          run 'abc' with '-dff -D 1' options
26
27      -nolutram
28          do not use EG_LOGIC_DRAM16X4 cells in output netlist
29
30      -nobram
31          do not use EG_PHY_BRAM or EG_PHY_BRAM32K cells in output netlist
32
33
34  The following commands are executed by this synthesis command:
35
36      begin:
37          read_verilog -lib +/anlogic/cells_sim.v +/anlogic/eagle_bb.v
38          hierarchy -check -top <top>
39
40      flatten:    (unless -noflatten)
41          proc
42          flatten
43          tribuf -logic
44          deminout
45
46      coarse:
47          synth -run coarse
48
```

```
49      map_ram:
50          memory_libmap -lib +/anlogic/lutrams.txt -lib +/anlogic/brams.txt [-no-auto-block] [-no-auto-distri
   (-no-auto-block if -nobram, -no-auto-distributed if -nolutram)
51          techmap -map +/anlogic/lutrams_map.v -map +/anlogic/brams_map.v
52
53      map_ffram:
54          opt -fast -mux_undef -undriven -fine
55          memory_map
56          opt -undriven -fine
57
58      map_gates:
59          techmap -map +/techmap.v -map +/anlogic/arith_map.v
60          opt -fast
61          abc -dff -D 1     (only if -retime)
62
63      map_ffs:
64          dfflegalize -cell $_DFFE_P??P_ r -cell $_SDFFE_P??P_ r -cell $_DLATCH_N??_ r
65          techmap -D NO_LUT -map +/anlogic/cells_map.v
66          opt_expr -mux_undef
67          simplemap
68
69      map_luts:
70          abc -lut 4:6
71          clean
72
73      map_cells:
74          techmap -map +/anlogic/cells_map.v
75          clean
76
77      map_anlogic:
78          anlogic_fixcarry
79          anlogic_eqn
80
81      check:
82          hierarchy -check
83          stat
84          check -noinit
85          blackbox =A:whitebox
86
87      edif:
88          write_edif <file-name>
89
90      json:
91          write_json <file-name>
```

C.194 synth_coolrunner2 – synthesis for Xilinx Coolrunner-II CPLDs

```
1      synth_coolrunner2 [options]
2
3   This command runs synthesis for Coolrunner-II CPLDs. This work is experimental.
4   It is intended to be used with https://github.com/azonenberg/openfpga as the
```

191

```
 5   place-and-route.
 6
 7       -top <module>
 8           use the specified module as top module (default='top')
 9
10       -json <file>
11           write the design to the specified JSON file. writing of an output file
12           is omitted if this parameter is not specified.
13
14       -run <from_label>:<to_label>
15           only run the commands between the labels (see below). an empty
16           from label is synonymous to 'begin', and empty to label is
17           synonymous to the end of the command list.
18
19       -noflatten
20           do not flatten design before synthesis
21
22       -retime
23           run 'abc' with '-dff -D 1' options
24
25
26   The following commands are executed by this synthesis command:
27
28       begin:
29           read_verilog -lib +/coolrunner2/cells_sim.v
30           hierarchy -check -top <top>
31
32       flatten:    (unless -noflatten)
33           proc
34           flatten
35           tribuf -logic
36
37       coarse:
38           synth -run coarse
39
40       fine:
41           extract_counter -dir up -allow_arst no
42           techmap -map +/coolrunner2/cells_counter_map.v
43           clean
44           opt -fast -full
45           techmap -map +/techmap.v -map +/coolrunner2/cells_latch.v
46           opt -fast
47           dfflibmap -prepare -liberty +/coolrunner2/xc2_dff.lib
48
49       map_tff:
50           abc -g AND,XOR
51           clean
52           extract -map +/coolrunner2/tff_extract.v
53
54       map_pla:
55           abc -sop -I 40 -P 56
56           clean
57
58       map_cells:
```

```
59          dfflibmap -liberty +/coolrunner2/xc2_dff.lib
60          dffinit -ff FDCP Q INIT
61          dffinit -ff FDCP_N Q INIT
62          dffinit -ff FTCP Q INIT
63          dffinit -ff FTCP_N Q INIT
64          dffinit -ff LDCP Q INIT
65          dffinit -ff LDCP_N Q INIT
66          coolrunner2_sop
67          clean
68          iopadmap -bits -inpad IBUF O:I -outpad IOBUFE I:IO -inoutpad IOBUFE O:IO -toutpad IOBUFE E:I:IO -ti
69          attrmvcp -attr src -attr LOC t:IOBUFE n:*
70          attrmvcp -attr src -attr LOC -driven t:IBUF n:*
71          coolrunner2_fixup
72          splitnets
73          clean
74
75      check:
76          hierarchy -check
77          stat
78          check -noinit
79          blackbox =A:whitebox
80
81      json:
82          write_json <file-name>
```

C.195 synth_easic – synthesis for eASIC platform

```
1       synth_easic [options]
2
3   This command runs synthesis for eASIC platform.
4
5       -top <module>
6           use the specified module as top module
7
8       -vlog <file>
9           write the design to the specified structural Verilog file. writing of
10          an output file is omitted if this parameter is not specified.
11
12      -etools <path>
13          set path to the eTools installation. (default=/opt/eTools)
14
15      -run <from_label>:<to_label>
16          only run the commands between the labels (see below). an empty
17          from label is synonymous to 'begin', and empty to label is
18          synonymous to the end of the command list.
19
20      -noflatten
21          do not flatten design before synthesis
22
23      -retime
24          run 'abc' with '-dff -D 1' options
25
```

```
26
27  The following commands are executed by this synthesis command:
28
29      begin:
30          read_liberty -lib <etools_phys_clk_lib>
31          read_liberty -lib <etools_logic_lut_lib>
32          hierarchy -check -top <top>
33
34      flatten:    (unless -noflatten)
35          proc
36          flatten
37
38      coarse:
39          synth -run coarse
40
41      fine:
42          opt -fast -mux_undef -undriven -fine
43          memory_map
44          opt -undriven -fine
45          techmap
46          opt -fast
47          abc -dff -D 1     (only if -retime)
48          opt_clean    (only if -retime)
49
50      map:
51          dfflibmap -liberty <etools_phys_clk_lib>
52          abc -liberty <etools_logic_lut_lib>
53          opt_clean
54
55      check:
56          hierarchy -check
57          stat
58          check -noinit
59          blackbox =A:whitebox
60
61      vlog:
62          write_verilog -noexpr -attr2comment <file-name>
```

C.196   synth_ecp5 – synthesis for ECP5 FPGAs

```
1       synth_ecp5 [options]
2
3   This command runs synthesis for ECP5 FPGAs.
4
5       -top <module>
6           use the specified module as top module
7
8       -blif <file>
9           write the design to the specified BLIF file. writing of an output file
10          is omitted if this parameter is not specified.
11
12      -edif <file>
```

```
13          write the design to the specified EDIF file. writing of an output file
14          is omitted if this parameter is not specified.
15
16      -json <file>
17          write the design to the specified JSON file. writing of an output file
18          is omitted if this parameter is not specified.
19
20      -run <from_label>:<to_label>
21          only run the commands between the labels (see below). an empty
22          from label is synonymous to 'begin', and empty to label is
23          synonymous to the end of the command list.
24
25      -noflatten
26          do not flatten design before synthesis
27
28      -dff
29          run 'abc'/'abc9' with -dff option
30
31      -retime
32          run 'abc' with '-dff -D 1' options
33
34      -noccu2
35          do not use CCU2 cells in output netlist
36
37      -nodffe
38          do not use flipflops with CE in output netlist
39
40      -nobram
41          do not use block RAM cells in output netlist
42
43      -nolutram
44          do not use LUT RAM cells in output netlist
45
46      -nowidelut
47          do not use PFU muxes to implement LUTs larger than LUT4s
48
49      -asyncprld
50          use async PRLD mode to implement ALDFF (EXPERIMENTAL)
51
52      -abc2
53          run two passes of 'abc' for slightly improved logic density
54
55      -abc9
56          use new ABC9 flow (EXPERIMENTAL)
57
58      -vpr
59          generate an output netlist (and BLIF file) suitable for VPR
60          (this feature is experimental and incomplete)
61
62      -nodsp
63          do not map multipliers to MULT18X18D
64
65      -no-rw-check
66          marks all recognized read ports as "return don't-care value on
```

```
 67          read/write collision" (same result as setting the no_rw_check
 68          attribute on all memories).
 69
 70
 71  The following commands are executed by this synthesis command:
 72
 73      begin:
 74          read_verilog -lib -specify +/ecp5/cells_sim.v +/ecp5/cells_bb.v
 75          hierarchy -check -top <top>
 76
 77      coarse:
 78          proc
 79          flatten
 80          tribuf -logic
 81          deminout
 82          opt_expr
 83          opt_clean
 84          check
 85          opt -nodffe -nosdff
 86          fsm
 87          opt
 88          wreduce
 89          peepopt
 90          opt_clean
 91          share
 92          techmap -map +/cmp2lut.v -D LUT_WIDTH=4
 93          opt_expr
 94          opt_clean
 95          techmap -map +/mul2dsp.v -map +/ecp5/dsp_map.v -D DSP_A_MAXWIDTH=18 -D DSP_B_MAXWIDTH=18
     -D DSP_A_MINWIDTH=2 -D DSP_B_MINWIDTH=2  -D DSP_NAME=$__MUL18X18    (unless -nodsp)
 96          chtype -set $mul t:$__soft_mul    (unless -nodsp)
 97          alumacc
 98          opt
 99          memory -nomap [-no-rw-check]
100          opt_clean
101
102      map_ram:
103          memory_libmap -lib +/ecp5/lutrams.txt -lib +/ecp5/brams.txt [-no-auto-block] [-no-auto-distributed]
     (-no-auto-block if -nobram, -no-auto-distributed if -nolutram)
104          techmap -map +/ecp5/lutrams_map.v -map +/ecp5/brams_map.v
105
106      map_ffram:
107          opt -fast -mux_undef -undriven -fine
108          memory_map
109          opt -undriven -fine
110
111      map_gates:
112          techmap -map +/techmap.v -map +/ecp5/arith_map.v
113          opt -fast
114          abc -dff -D 1    (only if -retime)
115
116      map_ffs:
117          opt_clean
```

```
118         dfflegalize -cell $_DFF_?_ 01 -cell $_DFF_?P?_ r -cell $_SDFF_?P?_ r [-cell $_DFFE_??_ 01 -cell $_D
    ($_ALDFF_*_ only if -asyncprld, $_DLATCH_* only if not -asyncprld, $_*DFFE_* only if not -nodffe)
119         zinit -all w:* t:$_DFF_?_ t:$_DFFE_??_ t:$_SDFF*    (only if -abc9 and -dff)
120         techmap -D NO_LUT -map +/ecp5/cells_map.v
121         opt_expr -undriven -mux_undef
122         simplemap
123         ecp5_gsr
124         attrmvcp -copy -attr syn_useioff
125         opt_clean
126
127    map_luts:
128         abc           (only if -abc2)
129         techmap -map +/ecp5/latches_map.v     (skip if -asyncprld)
130         abc -dress -lut 4:7
131         clean
132
133    map_cells:
134         techmap -map +/ecp5/cells_map.v     (skip if -vpr)
135         opt_lut_ins -tech ecp5
136         clean
137
138    check:
139         autoname
140         hierarchy -check
141         stat
142         check -noinit
143         blackbox =A:whitebox
144
145    blif:
146         opt_clean -purge                              (vpr mode)
147         write_blif -attr -cname -conn -param <file-name>    (vpr mode)
148         write_blif -gates -attr -param <file-name>          (non-vpr mode)
149
150    edif:
151         write_edif <file-name>
152
153    json:
154         write_json <file-name>
```

C.197   synth_efinix – synthesis for Efinix FPGAs

```
 1      synth_efinix [options]
 2
 3   This command runs synthesis for Efinix FPGAs.
 4
 5      -top <module>
 6          use the specified module as top module
 7
 8      -edif <file>
 9          write the design to the specified EDIF file. writing of an output file
10          is omitted if this parameter is not specified.
11
```

197

```
12      -json <file>
13          write the design to the specified JSON file. writing of an output file
14          is omitted if this parameter is not specified.
15
16      -run <from_label>:<to_label>
17          only run the commands between the labels (see below). an empty
18          from label is synonymous to 'begin', and empty to label is
19          synonymous to the end of the command list.
20
21      -noflatten
22          do not flatten design before synthesis
23
24      -retime
25          run 'abc' with '-dff -D 1' options
26
27      -nobram
28          do not use EFX_RAM_5K cells in output netlist
29
30
31  The following commands are executed by this synthesis command:
32
33      begin:
34          read_verilog -lib +/efinix/cells_sim.v
35          hierarchy -check -top <top>
36
37      flatten:    (unless -noflatten)
38          proc
39          flatten
40          tribuf -logic
41          deminout
42
43      coarse:
44          synth -run coarse
45
46      map_ram:
47          memory_libmap -lib +/efinix/brams.txt
48          techmap -map +/efinix/brams_map.v
49
50      map_ffram:
51          opt -fast -mux_undef -undriven -fine
52          memory_map
53          opt -undriven -fine
54
55      map_gates:
56          techmap -map +/techmap.v -map +/efinix/arith_map.v
57          opt -fast
58          abc -dff -D 1    (only if -retime)
59
60      map_ffs:
61          dfflegalize -cell $_DFFE_????_ 0 -cell $_SDFFE_????_ 0 -cell $_SDFFCE_????_ 0 -cell $_DLATCH_?_ x
62          techmap -D NO_LUT -map +/efinix/cells_map.v
63          opt_expr -mux_undef
64          simplemap
65
```

```
66      map_luts:
67          abc -lut 4
68          clean
69
70      map_cells:
71          techmap -map +/efinix/cells_map.v
72          clean
73
74      map_gbuf:
75          clkbufmap -buf $__EFX_GBUF O:I
76          techmap -map +/efinix/gbuf_map.v
77          efinix_fixcarry
78          clean
79
80      check:
81          hierarchy -check
82          stat
83          check -noinit
84          blackbox =A:whitebox
85
86      edif:
87          write_edif <file-name>
88
89      json:
90          write_json <file-name>
```

C.198 synth_gatemate – synthesis for Cologne Chip GateMate FPGAs

```
1       synth_gatemate [options]
2
3   This command runs synthesis for Cologne Chip AG GateMate FPGAs.
4
5       -top <module>
6           use the specified module as top module.
7
8       -vlog <file>
9           write the design to the specified verilog file. Writing of an output
10          file is omitted if this parameter is not specified.
11
12      -json <file>
13          write the design to the specified JSON file. Writing of an output file
14          is omitted if this parameter is not specified.
15
16      -run <from_label>:<to_label>
17          only run the commands between the labels (see below). An empty
18          from label is synonymous to 'begin', and empty to label is
19          synonymous to the end of the command list.
20
21      -noflatten
22          do not flatten design before synthesis.
23
```

```
24        -nobram
25            do not use CC_BRAM_20K or CC_BRAM_40K cells in output netlist.
26
27        -noaddf
28            do not use CC_ADDF full adder cells in output netlist.
29
30        -nomult
31            do not use CC_MULT multiplier cells in output netlist.
32
33        -nomx8, -nomx4
34            do not use CC_MX{8,4} multiplexer cells in output netlist.
35
36        -luttree
37            use new LUT tree mapping approach (EXPERIMENTAL).
38
39        -dff
40            run 'abc' with -dff option
41
42        -retime
43            run 'abc' with '-dff -D 1' options
44
45        -noiopad
46            disable I/O buffer insertion (useful for hierarchical or
47            out-of-context flows).
48
49        -noclkbuf
50            disable automatic clock buffer insertion.
51
52  The following commands are executed by this synthesis command:
53
54        begin:
55            read_verilog -lib -specify +/gatemate/cells_sim.v +/gatemate/cells_bb.v
56            hierarchy -check -top <top>
57
58        prepare:
59            proc
60            flatten
61            tribuf -logic
62            deminout
63            opt_expr
64            opt_clean
65            check
66            opt -nodffe -nosdff
67            fsm
68            opt
69            wreduce
70            peepopt
71            opt_clean
72            muxpack
73            share
74            techmap -map +/cmp2lut.v -D LUT_WIDTH=4
75            opt_expr
76            opt_clean
77
```

```
78      map_mult:    (skip if '-nomult')
79          techmap -map +/gatemate/mul_map.v
80
81      coarse:
82          alumacc
83          opt
84          memory -nomap
85          opt_clean
86
87      map_bram:    (skip if '-nobram')
88          memory_libmap -lib +/gatemate/brams.txt
89          techmap -map +/gatemate/brams_map.v
90
91      map_ffram:
92          opt -fast -mux_undef -undriven -fine
93          memory_map
94          opt -undriven -fine
95
96      map_gates:
97          techmap -map +/techmap.v  -map +/gatemate/arith_map.v
98          opt -fast
99
100     map_io:    (skip if '-noiopad')
101         iopadmap -bits -inpad CC_IBUF Y:I -outpad CC_OBUF A:O -toutpad CC_TOBUF ~T:A:O -tinoutpad CC_IOBUF
102         clean
103
104     map_regs:
105         opt_clean
106         dfflegalize -cell $_DFFE_????_ x -cell $_DLATCH_???_ x
107         techmap -map +/gatemate/reg_map.v
108         opt_expr -mux_undef
109         simplemap
110         opt_clean
111
112     map_muxs:
113         muxcover  -mux4 -mux8
114         opt -full
115         techmap -map +/gatemate/mux_map.v
116
117     map_luts:
118         abc  -genlib +/gatemate/lut_tree_cells.genlib    (with -luttree)
119         techmap -map +/gatemate/lut_tree_map.v    (with -luttree)
120         gatemate_foldinv    (with -luttree)
121         techmap -map +/gatemate/inv_map.v    (with -luttree)
122         abc  -dress -lut 4    (without -luttree)
123         clean
124
125     map_cells:
126         techmap -map +/gatemate/lut_map.v
127         clean
128
129     map_bufg:    (skip if '-noclkbuf')
130         clkbufmap -buf CC_BUFG O:I
131         clean
```

```
132
133        check:
134            hierarchy -check
135            stat -width
136            check -noinit
137            blackbox =A:whitebox
138
139        vlog:
140            opt_clean -purge
141            write_verilog -noattr <file-name>
142
143        json:
144            write_json <file-name>
```

C.199 synth_gowin – synthesis for Gowin FPGAs

```
1        synth_gowin [options]
2
3    This command runs synthesis for Gowin FPGAs. This work is experimental.
4
5        -top <module>
6            use the specified module as top module (default='top')
7
8        -vout <file>
9            write the design to the specified Verilog netlist file. writing of an
10           output file is omitted if this parameter is not specified.
11
12       -json <file>
13           write the design to the specified JSON netlist file. writing of an
14           output file is omitted if this parameter is not specified.
15           This disables features not yet supported by nexpnr-gowin.
16
17       -run <from_label>:<to_label>
18           only run the commands between the labels (see below). an empty
19           from label is synonymous to 'begin', and empty to label is
20           synonymous to the end of the command list.
21
22       -nodffe
23           do not use flipflops with CE in output netlist
24
25       -nobram
26           do not use BRAM cells in output netlist
27
28       -nolutram
29           do not use distributed RAM cells in output netlist
30
31       -noflatten
32           do not flatten design before synthesis
33
34       -retime
35           run 'abc' with '-dff -D 1' options
36
```

```
37      -nowidelut
38          do not use muxes to implement LUTs larger than LUT4s
39
40      -noiopads
41          do not emit IOB at top level ports
42
43      -noalu
44          do not use ALU cells
45
46      -abc9
47          use new ABC9 flow (EXPERIMENTAL)
48
49      -no-rw-check
50          marks all recognized read ports as "return don't-care value on
51          read/write collision" (same result as setting the no_rw_check
52          attribute on all memories).
53
54
55  The following commands are executed by this synthesis command:
56
57      begin:
58          read_verilog -specify -lib +/gowin/cells_sim.v
59          hierarchy -check -top <top>
60
61      flatten:    (unless -noflatten)
62          proc
63          flatten
64          tribuf -logic
65          deminout
66
67      coarse:
68          synth -run coarse [-no-rw-check]
69
70      map_ram:
71          memory_libmap -lib +/gowin/lutrams.txt -lib +/gowin/brams.txt [-no-auto-block] [-no-auto-distribute
    (-no-auto-block if -nobram, -no-auto-distributed if -nolutram)
72          techmap -map +/gowin/lutrams_map.v -map +/gowin/brams_map.v
73
74      map_ffram:
75          opt -fast -mux_undef -undriven -fine
76          memory_map
77          opt -undriven -fine
78
79      map_gates:
80          techmap -map +/techmap.v -map +/gowin/arith_map.v
81          opt -fast
82          abc -dff -D 1    (only if -retime)
83          iopadmap -bits -inpad IBUF O:I -outpad OBUF I:O -toutpad TBUF ~OEN:I:O -tinoutpad IOBUF ~OEN:O:I:IO
    (unless -noiopads)
84
85      map_ffs:
86          opt_clean
87          dfflegalize -cell $_DFF_?_ 0 -cell $_DFFE_?P_ 0 -cell $_SDFF_?P?_ r -cell $_SDFFE_?P?P_ r -cell $_D
88          techmap -map +/gowin/cells_map.v
```

```
 89           opt_expr -mux_undef
 90           simplemap
 91
 92       map_luts:
 93           abc -lut 4:8
 94           clean
 95
 96       map_cells:
 97           techmap -map +/gowin/cells_map.v
 98           opt_lut_ins -tech gowin
 99           setundef -undriven -params -zero
100           hilomap -singleton -hicell VCC V -locell GND G
101           splitnets -ports    (only if -vout used)
102           clean
103           autoname
104
105       check:
106           hierarchy -check
107           stat
108           check -noinit
109           blackbox =A:whitebox
110
111       vout:
112           write_verilog -simple-lhs -decimal -attr2comment -defparam -renameprefix gen <file-name>
113           write_json <file-name>
```

C.200 synth_greenpak4 – synthesis for GreenPAK4 FPGAs

```
 1       synth_greenpak4 [options]
 2
 3   This command runs synthesis for GreenPAK4 FPGAs. This work is experimental.
 4   It is intended to be used with https://github.com/azonenberg/openfpga as the
 5   place-and-route.
 6
 7       -top <module>
 8           use the specified module as top module (default='top')
 9
10       -part <part>
11           synthesize for the specified part. Valid values are SLG46140V,
12           SLG46620V, and SLG46621V (default).
13
14       -json <file>
15           write the design to the specified JSON file. writing of an output file
16           is omitted if this parameter is not specified.
17
18       -run <from_label>:<to_label>
19           only run the commands between the labels (see below). an empty
20           from label is synonymous to 'begin', and empty to label is
21           synonymous to the end of the command list.
22
23       -noflatten
24           do not flatten design before synthesis
```

```
25
26      -retime
27          run 'abc' with '-dff -D 1' options
28
29
30  The following commands are executed by this synthesis command:
31
32      begin:
33          read_verilog -lib +/greenpak4/cells_sim.v
34          hierarchy -check -top <top>
35
36      flatten:    (unless -noflatten)
37          proc
38          flatten
39          tribuf -logic
40
41      coarse:
42          synth -run coarse
43
44      fine:
45          extract_counter -pout GP_DCMP,GP_DAC -maxwidth 14
46          clean
47          opt -fast -mux_undef -undriven -fine
48          memory_map
49          opt -undriven -fine
50          techmap -map +/techmap.v -map +/greenpak4/cells_latch.v
51          dfflibmap -prepare -liberty +/greenpak4/gp_dff.lib
52          opt -fast -noclkinv -noff
53          abc -dff -D 1    (only if -retime)
54
55      map_luts:
56          nlutmap -assert -luts 0,6,8,2     (for -part SLG46140V)
57          nlutmap -assert -luts 2,8,16,2    (for -part SLG46620V)
58          nlutmap -assert -luts 2,8,16,2    (for -part SLG46621V)
59          clean
60
61      map_cells:
62          shregmap -tech greenpak4
63          dfflibmap -liberty +/greenpak4/gp_dff.lib
64          dffinit -ff GP_DFF Q INIT
65          dffinit -ff GP_DFFR Q INIT
66          dffinit -ff GP_DFFS Q INIT
67          dffinit -ff GP_DFFSR Q INIT
68          iopadmap -bits -inpad GP_IBUF OUT:IN -outpad GP_OBUF IN:OUT -inoutpad GP_OBUF OUT:IN -toutpad GP_OB
69          attrmvcp -attr src -attr LOC t:GP_OBUF t:GP_OBUFT t:GP_IOBUF n:*
70          attrmvcp -attr src -attr LOC -driven t:GP_IBUF n:*
71          techmap -map +/greenpak4/cells_map.v
72          greenpak4_dffinv
73          clean
74
75      check:
76          hierarchy -check
77          stat
78          check -noinit
```

```
79          blackbox =A:whitebox
80
81      json:
82          write_json <file-name>
```

C.201   synth_ice40 – synthesis for iCE40 FPGAs

```
1      synth_ice40 [options]
2
3  This command runs synthesis for iCE40 FPGAs.
4
5      -device < hx | lp | u >
6          relevant only for '-abc9' flow, optimise timing for the specified
7          device. default: hx
8
9      -top <module>
10          use the specified module as top module
11
12      -blif <file>
13          write the design to the specified BLIF file. writing of an output file
14          is omitted if this parameter is not specified.
15
16      -edif <file>
17          write the design to the specified EDIF file. writing of an output file
18          is omitted if this parameter is not specified.
19
20      -json <file>
21          write the design to the specified JSON file. writing of an output file
22          is omitted if this parameter is not specified.
23
24      -run <from_label>:<to_label>
25          only run the commands between the labels (see below). an empty
26          from label is synonymous to 'begin', and empty to label is
27          synonymous to the end of the command list.
28
29      -noflatten
30          do not flatten design before synthesis
31
32      -dff
33          run 'abc'/'abc9' with -dff option
34
35      -retime
36          run 'abc' with '-dff -D 1' options
37
38      -nocarry
39          do not use SB_CARRY cells in output netlist
40
41      -nodffe
42          do not use SB_DFFE* cells in output netlist
43
44      -dffe_min_ce_use <min_ce_use>
45          do not use SB_DFFE* cells if the resulting CE line would go to less
```

```
46          than min_ce_use SB_DFFE* in output netlist
47
48      -nobram
49          do not use SB_RAM40_4K* cells in output netlist
50
51      -spram
52          enable automatic inference of SB_SPRAM256KA
53
54      -dsp
55          use iCE40 UltraPlus DSP cells for large arithmetic
56
57      -noabc
58          use built-in Yosys LUT techmapping instead of abc
59
60      -abc2
61          run two passes of 'abc' for slightly improved logic density
62
63      -vpr
64          generate an output netlist (and BLIF file) suitable for VPR
65          (this feature is experimental and incomplete)
66
67      -abc9
68          use new ABC9 flow (EXPERIMENTAL)
69
70      -flowmap
71          use FlowMap LUT techmapping instead of abc (EXPERIMENTAL)
72
73      -no-rw-check
74          marks all recognized read ports as "return don't-care value on
75          read/write collision" (same result as setting the no_rw_check
76          attribute on all memories).
77
78
79  The following commands are executed by this synthesis command:
80
81      begin:
82          read_verilog -D ICE40_HX -lib -specify +/ice40/cells_sim.v
83          hierarchy -check -top <top>
84          proc
85
86      flatten:    (unless -noflatten)
87          flatten
88          tribuf -logic
89          deminout
90
91      coarse:
92          opt_expr
93          opt_clean
94          check
95          opt -nodffe -nosdff
96          fsm
97          opt
98          wreduce
99          peepopt
```

```
100         opt_clean
101         share
102         techmap -map +/cmp2lut.v -D LUT_WIDTH=4
103         opt_expr
104         opt_clean
105         memory_dff [-no-rw-check]
106         wreduce t:$mul
107         techmap -map +/mul2dsp.v -map +/ice40/dsp_map.v -D DSP_A_MAXWIDTH=16 -D DSP_B_MAXWIDTH=16 -D DSP_A_
    (if -dsp)
108         select a:mul2dsp              (if -dsp)
109         setattr -unset mul2dsp        (if -dsp)
110         opt_expr -fine                (if -dsp)
111         wreduce                       (if -dsp)
112         select -clear                 (if -dsp)
113         ice40_dsp                     (if -dsp)
114         chtype -set $mul t:$__soft_mul   (if -dsp)
115         alumacc
116         opt
117         memory -nomap [-no-rw-check]
118         opt_clean
119
120     map_ram:
121         memory_libmap -lib +/ice40/brams.txt -lib +/ice40/spram.txt -no-auto-huge [-no-auto-huge] [-no-auto
    (-no-auto-huge unless -spram, -no-auto-block if -nobram)
122         techmap -map +/ice40/brams_map.v -map +/ice40/spram_map.v
123         ice40_braminit
124
125     map_ffram:
126         opt -fast -mux_undef -undriven -fine
127         memory_map
128         opt -undriven -fine
129
130     map_gates:
131         ice40_wrapcarry
132         techmap -map +/techmap.v -map +/ice40/arith_map.v
133         opt -fast
134         abc -dff -D 1    (only if -retime)
135         ice40_opt
136
137     map_ffs:
138         dfflegalize -cell $_DFF_?_ 0 -cell $_DFFE_?P_ 0 -cell $_DFF_?P?_ 0 -cell $_DFFE_?P?P_ 0 -cell $_SDF
139         techmap -map +/ice40/ff_map.v
140         opt_expr -mux_undef
141         simplemap
142         ice40_opt -full
143
144     map_luts:
145         abc          (only if -abc2)
146         ice40_opt    (only if -abc2)
147         techmap -map +/ice40/latches_map.v
148         simplemap                                (if -noabc or -flowmap)
149         techmap -map +/gate2lut.v -D LUT_WIDTH=4    (only if -noabc)
150         flowmap -maxlut 4    (only if -flowmap)
151         abc -dress -lut 4    (skip if -noabc)
```

```
152        ice40_wrapcarry -unwrap
153        techmap -map +/ice40/ff_map.v
154        clean
155        opt_lut -dlogic SB_CARRY:I0=1:I1=2:CI=3 -dlogic SB_CARRY:CO=3
156
157    map_cells:
158        techmap -map +/ice40/cells_map.v    (skip if -vpr)
159        clean
160
161    check:
162        autoname
163        hierarchy -check
164        stat
165        check -noinit
166        blackbox =A:whitebox
167
168    blif:
169        opt_clean -purge                              (vpr mode)
170        write_blif -attr -cname -conn -param <file-name>    (vpr mode)
171        write_blif -gates -attr -param <file-name>          (non-vpr mode)
172
173    edif:
174        write_edif <file-name>
175
176    json:
177        write_json <file-name>
```

C.202 synth_intel – synthesis for Intel (Altera) FPGAs.

```
1      synth_intel [options]
2
3  This command runs synthesis for Intel FPGAs.
4
5      -family <max10 | cyclone10lp | cycloneiv | cycloneive>
6          generate the synthesis netlist for the specified family.
7          MAX10 is the default target if no family argument specified.
8          For Cyclone IV GX devices, use cycloneiv argument; for Cyclone IV E, use
9          cycloneive. For Cyclone V and Cyclone 10 GX, use the synth_intel_alm
10         backend instead.
11
12     -top <module>
13         use the specified module as top module (default='top')
14
15     -vqm <file>
16         write the design to the specified Verilog Quartus Mapping File. Writing
17         of an output file is omitted if this parameter is not specified.
18         Note that this backend has not been tested and is likely incompatible
19         with recent versions of Quartus.
20
21     -vpr <file>
22         write BLIF files for VPR flow experiments. The synthesized BLIF output
23         file is not compatible with the Quartus flow. Writing of an
```

```
24          output file is omitted if this parameter is not specified.
25
26      -run <from_label>:<to_label>
27          only run the commands between the labels (see below). an empty
28          from label is synonymous to 'begin', and empty to label is
29          synonymous to the end of the command list.
30
31      -iopads
32          use IO pad cells in output netlist
33
34      -nobram
35          do not use block RAM cells in output netlist
36
37      -noflatten
38          do not flatten design before synthesis
39
40      -retime
41          run 'abc' with '-dff -D 1' options
42
43  The following commands are executed by this synthesis command:
44
45      begin:
46
47      family:
48          read_verilog -sv -lib +/intel/max10/cells_sim.v
49          read_verilog -sv -lib +/intel/common/m9k_bb.v
50          read_verilog -sv -lib +/intel/common/altpll_bb.v
51          hierarchy -check -top <top>
52
53      flatten:    (unless -noflatten)
54          proc
55          flatten
56          tribuf -logic
57          deminout
58
59      coarse:
60          synth -run coarse
61
62      map_bram:    (skip if -nobram)
63          memory_bram -rules +/intel/common/brams_m9k.txt    (if applicable for family)
64          techmap -map +/intel/common/brams_map_m9k.v    (if applicable for family)
65
66      map_ffram:
67          opt -fast -mux_undef -undriven -fine -full
68          memory_map
69          opt -undriven -fine
70          techmap -map +/techmap.v
71          opt -full
72          clean -purge
73          setundef -undriven -zero
74          abc -markgroups -dff -D 1    (only if -retime)
75
76      map_ffs:
77          dfflegalize -cell $_DFFE_PN0P_ 01
```

```
78          techmap -map +/intel/common/ff_map.v
79
80      map_luts:
81          abc -lut 4
82          clean
83
84      map_cells:
85          iopadmap -bits -outpad $__outpad I:O -inpad $__inpad O:I    (if -iopads)
86          techmap -map +/intel/max10/cells_map.v
87          clean -purge
88
89      check:
90          hierarchy -check
91          stat
92          check -noinit
93          blackbox =A:whitebox
94
95      vqm:
96          write_verilog -attr2comment -defparam -nohex -decimal -renameprefix syn_ <file-name>
97
98      vpr:
99          opt_clean -purge
100         write_blif <file-name>
101
102
103 WARNING: THE 'synth_intel' COMMAND IS EXPERIMENTAL.
```

C.203   synth_intel_alm – synthesis for ALM-based Intel (Altera) FPGAs.

```
1       synth_intel_alm [options]
2
3   This command runs synthesis for ALM-based Intel FPGAs.
4
5       -top <module>
6           use the specified module as top module
7
8       -family <family>
9           target one of:
10          "cyclonev"   - Cyclone V (default)
11          "arriav"     - Arria V (non-GZ)
12          "cyclone10gx" - Cyclone 10GX
13
14      -vqm <file>
15          write the design to the specified Verilog Quartus Mapping File. Writing
16          of an output file is omitted if this parameter is not specified. Implies
17          -quartus.
18
19      -noflatten
20          do not flatten design before synthesis; useful for per-module area
21          statistics
22
```

```
23      -quartus
24          output a netlist using Quartus cells instead of MISTRAL_* cells
25
26      -dff
27          pass DFFs to ABC to perform sequential logic optimisations
28          (EXPERIMENTAL)
29
30      -run <from_label>:<to_label>
31          only run the commands between the labels (see below). an empty
32          from label is synonymous to 'begin', and empty to label is
33          synonymous to the end of the command list.
34
35      -nolutram
36          do not use LUT RAM cells in output netlist
37
38      -nobram
39          do not use block RAM cells in output netlist
40
41      -nodsp
42          do not map multipliers to MISTRAL_MUL cells
43
44      -noiopad
45          do not instantiate IO buffers
46
47      -noclkbuf
48          do not insert global clock buffers
49
50  The following commands are executed by this synthesis command:
51
52      begin:
53          read_verilog -specify -lib -D <family> +/intel_alm/common/alm_sim.v
54          read_verilog -specify -lib -D <family> +/intel_alm/common/dff_sim.v
55          read_verilog -specify -lib -D <family> +/intel_alm/common/dsp_sim.v
56          read_verilog -specify -lib -D <family> +/intel_alm/common/mem_sim.v
57          read_verilog -specify -lib -D <family> +/intel_alm/common/misc_sim.v
58          read_verilog -specify -lib -D <family> -icells +/intel_alm/common/abc9_model.v
59          read_verilog -lib +/intel/common/altpll_bb.v
60          read_verilog -lib +/intel_alm/common/megafunction_bb.v
61          hierarchy -check -top <top>
62
63      coarse:
64          proc
65          flatten    (skip if -noflatten)
66          tribuf -logic
67          deminout
68          opt_expr
69          opt_clean
70          check
71          opt -nodffe -nosdff
72          fsm
73          opt
74          wreduce
75          peepopt
76          opt_clean
```

```
 77            share
 78            techmap -map +/cmp2lut.v -D LUT_WIDTH=6
 79            opt_expr
 80            opt_clean
 81            techmap -map +/mul2dsp.v [...]    (unless -nodsp)
 82            alumacc
 83            iopadmap -bits -outpad MISTRAL_OB I:PAD -inpad MISTRAL_IB O:PAD -toutpad MISTRAL_IO OE:O:PAD -tinou
    (unless -noiopad)
 84            techmap -map +/intel_alm/common/arith_alm_map.v -map +/intel_alm/common/dsp_map.v
 85            opt
 86            memory -nomap
 87            opt_clean
 88
 89        map_bram:    (skip if -nobram)
 90            memory_bram -rules +/intel_alm/common/bram_<bram_type>.txt
 91            techmap -map +/intel_alm/common/bram_<bram_type>_map.v
 92
 93        map_lutram:    (skip if -nolutram)
 94            memory_bram -rules +/intel_alm/common/lutram_mlab.txt    (for Cyclone V / Cyclone 10GX)
 95
 96        map_ffram:
 97            memory_map
 98            opt -full
 99
100        map_ffs:
101            techmap
102            dfflegalize -cell $_DFFE_PN0P_ 0 -cell $_SDFFCE_PP0P_ 0
103            techmap -map +/intel_alm/common/dff_map.v
104            opt -full -undriven -mux_undef
105            clean -purge
106            clkbufmap -buf MISTRAL_CLKBUF Q:A    (unless -noclkbuf)
107
108        map_luts:
109            techmap -map +/intel_alm/common/abc9_map.v
110            abc9 [-dff] -maxlut 6 -W 600
111            techmap -map +/intel_alm/common/abc9_unmap.v
112            techmap -map +/intel_alm/common/alm_map.v
113            opt -fast
114            autoname
115            clean
116
117        check:
118            hierarchy -check
119            stat
120            check
121            blackbox =A:whitebox
122
123        quartus:
124            rename -hide w:*[* w:*]*
125            setundef -zero
126            hilomap -singleton -hicell __MISTRAL_VCC Q -locell __MISTRAL_GND Q
127            techmap -D <family> -map +/intel_alm/common/quartus_rename.v
128
129        vqm:
```

```
130        write_verilog -attr2comment -defparam -nohex -decimal <file-name>
```

C.204   synth_machxo2 – synthesis for MachXO2 FPGAs. This work is experimental.

```
 1      synth_machxo2 [options]
 2
 3  This command runs synthesis for MachXO2 FPGAs.
 4
 5      -top <module>
 6          use the specified module as top module
 7
 8      -blif <file>
 9          write the design to the specified BLIF file. writing of an output file
10          is omitted if this parameter is not specified.
11
12      -edif <file>
13          write the design to the specified EDIF file. writing of an output file
14          is omitted if this parameter is not specified.
15
16      -json <file>
17          write the design to the specified JSON file. writing of an output file
18          is omitted if this parameter is not specified.
19
20      -run <from_label>:<to_label>
21          only run the commands between the labels (see below). an empty
22          from label is synonymous to 'begin', and empty to label is
23          synonymous to the end of the command list.
24
25      -nobram
26          do not use block RAM cells in output netlist
27
28      -nolutram
29          do not use LUT RAM cells in output netlist
30
31      -noflatten
32          do not flatten design before synthesis
33
34      -noiopad
35          do not insert IO buffers
36
37      -vpr
38          generate an output netlist (and BLIF file) suitable for VPR
39          (this feature is experimental and incomplete)
40
41
42  The following commands are executed by this synthesis command:
43
44      begin:
45          read_verilog -lib -icells +/machxo2/cells_sim.v
46          hierarchy -check -top <top>
47
```

```
48      flatten:    (unless -noflatten)
49          proc
50          flatten
51          tribuf -logic
52          deminout
53
54      coarse:
55          synth -run coarse
56
57      map_ram:
58          memory_libmap -lib +/machxo2/lutrams.txt -lib +/machxo2/brams.txt [-no-auto-block] [-no-auto-distri
   (-no-auto-block if -nobram, -no-auto-distributed if -nolutram)
59          techmap -map +/machxo2/lutrams_map.v -map +/machxo2/brams_map.v
60
61      fine:
62          memory_map
63          opt -full
64          techmap -map +/techmap.v
65          opt -fast
66
67      map_ios:    (unless -noiopad)
68          iopadmap -bits -outpad $__FACADE_OUTPAD I:O -inpad $__FACADE_INPAD O:I -toutpad $__FACADE_TOUTPAD ~
69          attrmvcp -attr src -attr LOC t:$__FACADE_OUTPAD %x:+[O] t:$__FACADE_TOUTPAD %x:+[O] t:$__FACADE_TIN
70          attrmvcp -attr src -attr LOC -driven t:$__FACADE_INPAD %x:+[I]
71
72      map_ffs:
73          dfflegalize -cell $_DFF_P_ 0
74
75      map_luts:
76          abc -lut 4 -dress
77          clean
78
79      map_cells:
80          techmap -map +/machxo2/cells_map.v
81          clean
82
83      check:
84          hierarchy -check
85          stat
86          blackbox =A:whitebox
87
88      blif:
89          opt_clean -purge                                  (vpr mode)
90          write_blif -attr -cname -conn -param <file-name>  (vpr mode)
91          write_blif -gates -attr -param <file-name>        (non-vpr mode)
92
93      edif:
94          write_edif <file-name>
95
96      json:
97          write_json <file-name>
```

C.205   synth_nexus – synthesis for Lattice Nexus FPGAs

```
 1      synth_nexus [options]
 2
 3  This command runs synthesis for Lattice Nexus FPGAs.
 4
 5      -top <module>
 6          use the specified module as top module
 7
 8      -family <device>
 9          run synthesis for the specified Nexus device
10          supported values: lifcl, lfd2nx
11
12      -json <file>
13          write the design to the specified JSON file. writing of an output file
14          is omitted if this parameter is not specified.
15
16      -vm <file>
17          write the design to the specified structural Verilog file. writing of
18          an output file is omitted if this parameter is not specified.
19
20      -run <from_label>:<to_label>
21          only run the commands between the labels (see below). an empty
22          from label is synonymous to 'begin', and empty to label is
23          synonymous to the end of the command list.
24
25      -noflatten
26          do not flatten design before synthesis
27
28      -dff
29          run 'abc'/'abc9' with -dff option
30
31      -retime
32          run 'abc' with '-dff -D 1' options
33
34      -noccu2
35          do not use CCU2 cells in output netlist
36
37      -nodffe
38          do not use flipflops with CE in output netlist
39
40      -nolram
41          do not use large RAM cells in output netlist
42          note that large RAM must be explicitly requested with a (* lram *)
43          attribute on the memory.
44
45      -nobram
46          do not use block RAM cells in output netlist
47
48      -nolutram
49          do not use LUT RAM cells in output netlist
50
51      -nowidelut
52          do not use PFU muxes to implement LUTs larger than LUT4s
```

```
 53
 54      -noiopad
 55          do not insert IO buffers
 56
 57      -nodsp
 58          do not infer DSP multipliers
 59
 60      -abc9
 61          use new ABC9 flow (EXPERIMENTAL)
 62
 63  The following commands are executed by this synthesis command:
 64
 65      begin:
 66          read_verilog -lib -specify +/nexus/cells_sim.v +/nexus/cells_xtra.v
 67          hierarchy -check -top <top>
 68
 69      coarse:
 70          proc
 71          flatten
 72          tribuf -logic
 73          deminout
 74          opt_expr
 75          opt_clean
 76          check
 77          opt -nodffe -nosdff
 78          fsm
 79          opt
 80          wreduce
 81          peepopt
 82          opt_clean
 83          share
 84          techmap -map +/cmp2lut.v -D LUT_WIDTH=4
 85          opt_expr
 86          opt_clean
 87          techmap -map +/mul2dsp.v [...]    (unless -nodsp)
 88          techmap -map +/nexus/dsp_map.v    (unless -nodsp)
 89          alumacc
 90          opt
 91          memory -nomap
 92          opt_clean
 93
 94      map_ram:
 95          memory_libmap -lib +/nexus/lutrams.txt -lib +/nexus/brams.txt -lib +/nexus/lrams.txt -no-auto-huge
    (-no-auto-block if -nobram, -no-auto-distributed if -nolutram)
 96          techmap -map +/nexus/lutrams_map.v -map +/nexus/brams_map.v -map +/nexus/lrams_map.v
 97
 98      map_ffram:
 99          opt -fast -mux_undef -undriven -fine
100          memory_map
101          opt -undriven -fine
102
103      map_gates:
104          techmap -map +/techmap.v -map +/nexus/arith_map.v
```

```
105        iopadmap -bits -outpad OB I:O -inpad IB O:I -toutpad OBZ ~T:I:O -tinoutpad BB ~T:O:I:B A:top
    (skip if '-noiopad')
106        opt -fast
107        abc -dff -D 1    (only if -retime)
108
109    map_ffs:
110        opt_clean
111        dfflegalize -cell $_DFF_P_ 01 -cell $_DFF_PP?_ r -cell $_SDFF_PP?_ r -cell $_DLATCH_?_ x [-cell $_D
    ($_*DFFE_* only if not -nodffe)
112        zinit -all w:* t:$_DFF_?_ t:$_DFFE_??_ t:$_SDFF*    (only if -abc9 and -dff
113        techmap -D NO_LUT -map +/nexus/cells_map.v
114        opt_expr -undriven -mux_undef
115        simplemap
116        attrmvcp -copy -attr syn_useioff
117        opt_clean
118
119    map_luts:
120        techmap -map +/nexus/latches_map.v
121        abc -dress -lut 4:5
122        clean
123
124    map_cells:
125        techmap -map +/nexus/cells_map.v
126        setundef -zero
127        hilomap -singleton -hicell VHI Z -locell VLO Z
128        clean
129
130    check:
131        autoname
132        hierarchy -check
133        stat
134        check -noinit
135        blackbox =A:whitebox
136
137    json:
138        write_json <file-name>
139
140    vm:
141        write_verilog <file-name>
```

C.206   synth_quicklogic – Synthesis for QuickLogic FPGAs

```
1     synth_quicklogic [options]
2  This command runs synthesis for QuickLogic FPGAs
3
4     -top <module>
5         use the specified module as top module
6
7     -family <family>
8         run synthesis for the specified QuickLogic architecture
9         generate the synthesis netlist for the specified family.
10        supported values:
```

```
11          - pp3: PolarPro 3
12
13      -blif <file>
14          write the design to the specified BLIF file. writing of an output file
15          is omitted if this parameter is not specified.
16
17      -verilog <file>
18          write the design to the specified verilog file. writing of an output
19          file is omitted if this parameter is not specified.
20
21      -abc
22          use old ABC flow, which has generally worse mapping results but is less
23          likely to have bugs.
24
25  The following commands are executed by this synthesis command:
26
27      begin:
28          read_verilog -lib -specify +/quicklogic/cells_sim.v +/quicklogic/pp3_cells_sim.v
29          read_verilog -lib -specify +/quicklogic/lut_sim.v
30          hierarchy -check -top <top>
31
32      coarse:
33          proc
34          flatten
35          tribuf -logic
36          deminout
37          opt_expr
38          opt_clean
39          check
40          opt -nodffe -nosdff
41          fsm
42          opt
43          wreduce
44          peepopt
45          opt_clean
46          share
47          techmap -map +/cmp2lut.v -D LUT_WIDTH=4
48          opt_expr
49          opt_clean
50          alumacc
51          pmuxtree
52          opt
53          memory -nomap
54          opt_clean
55
56      map_ffram:
57          opt -fast -mux_undef -undriven -fine
58          memory_map -iattr -attr !ram_block -attr !rom_block -attr logic_block -attr syn_ramstyle=auto -attr
59          opt -undriven -fine
60
61      map_gates:
62          techmap
63          opt -fast
64          muxcover -mux8 -mux4
```

```
65
66      map_ffs:
67          opt_expr
68          dfflegalize -cell $_DFFSRE_PPPP_ 0 -cell $_DLATCH_?_ x
69          techmap -map +/quicklogic/pp3_cells_map.v -map +/quicklogic/pp3_ffs_map.v
70          opt_expr -mux_undef
71
72      map_luts:
73          techmap -map +/quicklogic/pp3_latches_map.v
74          read_verilog -lib -specify -icells +/quicklogic/abc9_model.v
75          techmap -map +/quicklogic/abc9_map.v
76          abc9 -maxlut 4 -dff
77          techmap -map +/quicklogic/abc9_unmap.v
78          clean
79
80      map_cells:
81          techmap -map +/quicklogic/pp3_lut_map.v
82          clean
83
84      check:
85          autoname
86          hierarchy -check
87          stat
88          check -noinit
89
90      iomap:
91          clkbufmap -inpad ckpad Q:P
92          iopadmap -bits -outpad outpad A:P -inpad inpad Q:P -tinoutpad bipad EN:Q:A:P A:top
93
94      finalize:
95          setundef -zero -params -undriven
96          hilomap -hicell logic_1 A -locell logic_0 A -singleton A:top
97          opt_clean -purge
98          check
99          blackbox =A:whitebox
100
101     blif:
102         write_blif -attr -param -auto-top
103
104     verilog:
105         write_verilog -noattr -nohex <file-name>
```

C.207   synth_sf2 – synthesis for SmartFusion2 and IGLOO2 FP-GAs

```
1       synth_sf2 [options]
2
3   This command runs synthesis for SmartFusion2 and IGLOO2 FPGAs.
4
5       -top <module>
6           use the specified module as top module
7
```

220

```
 8        -edif <file>
 9            write the design to the specified EDIF file. writing of an output file
10            is omitted if this parameter is not specified.
11
12        -vlog <file>
13            write the design to the specified Verilog file. writing of an output
14            file is omitted if this parameter is not specified.
15
16        -json <file>
17            write the design to the specified JSON file. writing of an output file
18            is omitted if this parameter is not specified.
19
20        -run <from_label>:<to_label>
21            only run the commands between the labels (see below). an empty
22            from label is synonymous to 'begin', and empty to label is
23            synonymous to the end of the command list.
24
25        -noflatten
26            do not flatten design before synthesis
27
28        -noiobs
29            run synthesis in "block mode", i.e. do not insert IO buffers
30
31        -clkbuf
32            insert direct PAD->global_net buffers
33
34        -discard-ffinit
35            discard FF init value instead of emitting an error
36
37        -retime
38            run 'abc' with '-dff -D 1' options
39
40
41  The following commands are executed by this synthesis command:
42
43        begin:
44            read_verilog -lib +/sf2/cells_sim.v
45            hierarchy -check -top <top>
46
47        flatten:    (unless -noflatten)
48            proc
49            flatten
50            tribuf -logic
51            deminout
52
53        coarse:
54            attrmap -remove init    (only if -discard-ffinit)
55            synth -run coarse
56
57        fine:
58            opt -fast -mux_undef -undriven -fine
59            memory_map
60            opt -undriven -fine
61            techmap -map +/techmap.v -map +/sf2/arith_map.v
```

221

```
62         opt -fast
63         abc -dff -D 1    (only if -retime)
64
65     map_ffs:
66         dfflegalize -cell $_DFFE_PN?P_ x -cell $_SDFFCE_PN?P_ x -cell $_DLATCH_PN?_ x
67         techmap -D NO_LUT -map +/sf2/cells_map.v
68         opt_expr -mux_undef
69         simplemap
70
71     map_luts:
72         abc -lut 4
73         clean
74
75     map_cells:
76         techmap -map +/sf2/cells_map.v
77         clean
78
79     map_iobs:
80         clkbufmap -buf CLKINT Y:A [-inpad CLKBUF Y:PAD]    (unless -noiobs, -inpad only passed if -clkbuf)
81         iopadmap -bits -inpad INBUF Y:PAD -outpad OUTBUF D:PAD -toutpad TRIBUFF E:D:PAD -tinoutpad BIBUF E:
    (unless -noiobs)
82         clean -purge
83
84     check:
85         hierarchy -check
86         stat
87         check -noinit
88         blackbox =A:whitebox
89
90     edif:
91         write_edif -gndvccy <file-name>
92
93     vlog:
94         write_verilog <file-name>
95
96     json:
97         write_json <file-name>
```

C.208   synth_xilinx – synthesis for Xilinx FPGAs

```
1     synth_xilinx [options]
2
3  This command runs synthesis for Xilinx FPGAs. This command does not operate on
4  partly selected designs. At the moment this command creates netlists that are
5  compatible with 7-Series Xilinx devices.
6
7      -top <module>
8          use the specified module as top module
9
10     -family <family>
11         run synthesis for the specified Xilinx architecture
12         generate the synthesis netlist for the specified family.
```

```
13          supported values:
14          - xcup: Ultrascale Plus
15          - xcu: Ultrascale
16          - xc7: Series 7 (default)
17          - xc6s: Spartan 6
18          - xc6v: Virtex 6
19          - xc5v: Virtex 5 (EXPERIMENTAL)
20          - xc4v: Virtex 4 (EXPERIMENTAL)
21          - xc3sda: Spartan 3A DSP (EXPERIMENTAL)
22          - xc3sa: Spartan 3A (EXPERIMENTAL)
23          - xc3se: Spartan 3E (EXPERIMENTAL)
24          - xc3s: Spartan 3 (EXPERIMENTAL)
25          - xc2vp: Virtex 2 Pro (EXPERIMENTAL)
26          - xc2v: Virtex 2 (EXPERIMENTAL)
27          - xcve: Virtex E, Spartan 2E (EXPERIMENTAL)
28          - xcv: Virtex, Spartan 2 (EXPERIMENTAL)
29
30      -edif <file>
31          write the design to the specified edif file. writing of an output file
32          is omitted if this parameter is not specified.
33
34      -blif <file>
35          write the design to the specified BLIF file. writing of an output file
36          is omitted if this parameter is not specified.
37
38      -ise
39          generate an output netlist suitable for ISE
40
41      -nobram
42          do not use block RAM cells in output netlist
43
44      -nolutram
45          do not use distributed RAM cells in output netlist
46
47      -nosrl
48          do not use distributed SRL cells in output netlist
49
50      -nocarry
51          do not use XORCY/MUXCY/CARRY4 cells in output netlist
52
53      -nowidelut
54          do not use MUXF[5-9] resources to implement LUTs larger than native for
55          the target
56
57      -nodsp
58          do not use DSP48*s to implement multipliers and associated logic
59
60      -noiopad
61          disable I/O buffer insertion (useful for hierarchical or
62          out-of-context flows)
63
64      -noclkbuf
65          disable automatic clock buffer insertion
66
```

```
67      -uram
68          infer URAM288s for large memories (xcup only)
69
70      -widemux <int>
71          enable inference of hard multiplexer resources (MUXF[78]) for muxes at
72          or above this number of inputs (minimum value 2, recommended value >= 5)
73          default: 0 (no inference)
74
75      -run <from_label>:<to_label>
76          only run the commands between the labels (see below). an empty
77          from label is synonymous to 'begin', and empty to label is
78          synonymous to the end of the command list.
79
80      -flatten
81          flatten design before synthesis
82
83      -dff
84          run 'abc'/'abc9' with -dff option
85
86      -retime
87          run 'abc' with '-D 1' option to enable flip-flop retiming.
88          implies -dff.
89
90      -abc9
91          use new ABC9 flow (EXPERIMENTAL)
92
93
94  The following commands are executed by this synthesis command:
95
96      begin:
97          read_verilog -lib -specify +/xilinx/cells_sim.v
98          read_verilog -lib +/xilinx/cells_xtra.v
99          hierarchy -check -auto-top
100
101     prepare:
102         proc
103         flatten     (with '-flatten')
104         tribuf -logic
105         deminout
106         opt_expr
107         opt_clean
108         check
109         opt -nodffe -nosdff
110         fsm
111         opt
112         wreduce [-keepdc]    (option for '-widemux')
113         peepopt
114         opt_clean
115         muxpack         ('-widemux' only)
116         pmux2shiftx     (skip if '-nosrl' and '-widemux=0')
117         clean           (skip if '-nosrl' and '-widemux=0')
118
119     map_dsp:    (skip if '-nodsp')
120         memory_dff
```

```
121         techmap -map +/mul2dsp.v -map +/xilinx/{family}_dsp_map.v {options}
122         select a:mul2dsp
123         setattr -unset mul2dsp
124         opt_expr -fine
125         wreduce
126         select -clear
127         xilinx_dsp -family <family>
128         chtype -set $mul t:$__soft_mul
129
130     coarse:
131         techmap -map +/cmp2lut.v -map +/cmp2lcu.v -D LUT_WIDTH=[46]
132         alumacc
133         share
134         opt
135         memory -nomap
136         opt_clean
137
138     map_memory:
139         memory_libmap [...]
140         techmap -map +/xilinx/lutrams_<family>_map.v
141         techmap -map +/xilinx/brams_<family>_map.v
142
143     map_ffram:
144         opt -fast -full
145         memory_map
146
147     fine:
148         simplemap t:$mux    ('-widemux' only)
149         muxcover <internal options>    ('-widemux' only)
150         opt -full
151         xilinx_srl -variable -minlen 3    (skip if '-nosrl')
152         techmap  -map +/techmap.v -D LUT_SIZE=[46] [-map +/xilinx/mux_map.v] -map +/xilinx/arith_map.v
153         opt -fast
154
155     map_cells:
156         iopadmap -bits -outpad OBUF I:O -inpad IBUF O:I -toutpad OBUFT ~T:I:O -tinoutpad IOBUF ~T:O:I:IO A:
   (skip if '-noiopad')
157         techmap -map +/techmap.v -map +/xilinx/cells_map.v
158         clean
159
160     map_ffs:
161         dfflegalize -cell $_DFFE_?P?P_ 01 -cell $_SDFFE_?P?P_ 01 -cell $_DLATCH_?P?_ 01    (for xc6v, xc7,
162         zinit -all w:* t:$_SDFFE_*    ('-dff' only)
163         techmap -map +/xilinx/ff_map.v    ('-abc9' only)
164
165     map_luts:
166         opt_expr -mux_undef -noclkinv
167         abc -luts 2:2,3,6:5[,10,20] [-dff] [-D 1]    (option for '-nowidelut', '-dff', '-retime')
168         clean
169         techmap -map +/xilinx/ff_map.v    (only if not '-abc9')
170         xilinx_srl -fixed -minlen 3    (skip if '-nosrl')
171         techmap -map +/xilinx/lut_map.v -map +/xilinx/cells_map.v -D LUT_WIDTH=[46]
172         xilinx_dffopt [-lut4]
173         opt_lut_ins -tech xilinx
```

```
174
175        finalize:
176            clkbufmap -buf BUFG O:I    (skip if '-noclkbuf')
177            extractinv -inv INV O:I    (only if '-ise')
178            clean
179
180        check:
181            hierarchy -check
182            stat -tech xilinx
183            check -noinit
184            blackbox =A:whitebox
185
186        edif:
187            write_edif -pvector bra
188
189        blif:
190            write_blif
```

C.209   tcl – execute a TCL script file

```
1        tcl <filename> [args]
2
3    This command executes the tcl commands in the specified file.
4    Use 'yosys cmd' to run the yosys command 'cmd' from tcl.
5
6    The tcl command 'yosys -import' can be used to import all yosys
7    commands directly as tcl commands to the tcl shell. Yosys commands
8    'proc' and 'rename' are wrapped to tcl commands 'procs' and 'renames'
9    in order to avoid a name collision with the built in commands.
10
11   If any arguments are specified, these arguments are provided to the script via
12   the standard $argc and $argv variables.
13
14   Note, tcl will not recieve the output of any yosys command. If the output
15   of the tcl commands are needed, use the yosys command 'tee' to redirect yosys's
16   output to a temporary file.
```

C.210   techmap – generic technology mapper

```
1        techmap [-map filename] [selection]
2
3    This pass implements a very simple technology mapper that replaces cells in
4    the design with implementations given in form of a Verilog or RTLIL source
5    file.
6
7        -map filename
8            the library of cell implementations to be used.
9            without this parameter a builtin library is used that
10           transforms the internal RTL cells to the internal gate
11           library.
```

```
12
13       -map %<design-name>
14           like -map above, but with an in-memory design instead of a file.
15
16       -extern
17           load the cell implementations as separate modules into the design
18           instead of inlining them.
19
20       -max_iter <number>
21           only run the specified number of iterations on each module.
22           default: unlimited
23
24       -recursive
25           instead of the iterative breadth-first algorithm use a recursive
26           depth-first algorithm. both methods should yield equivalent results,
27           but may differ in performance.
28
29       -autoproc
30           Automatically call "proc" on implementations that contain processes.
31
32       -wb
33           Ignore the 'whitebox' attribute on cell implementations.
34
35       -assert
36           this option will cause techmap to exit with an error if it can't map
37           a selected cell. only cell types that end on an underscore are accepted
38           as final cell types by this mode.
39
40       -D <define>, -I <incdir>
41           this options are passed as-is to the Verilog frontend for loading the
42           map file. Note that the Verilog frontend is also called with the
43           '-nooverwrite' option set.
44
45 When a module in the map file has the 'techmap_celltype' attribute set, it will
46 match cells with a type that match the text value of this attribute. Otherwise
47 the module name will be used to match the cell.  Multiple space-separated cell
48 types can be listed, and wildcards using [] will be expanded (ie.
49 "$_DFF_[PN]_" is the same as "$_DFF_P_ $_DFF_N_").
50
51 When a module in the map file has the 'techmap_simplemap' attribute set, techmap
52 will use 'simplemap' (see 'help simplemap') to map cells matching the module.
53
54 When a module in the map file has the 'techmap_maccmap' attribute set, techmap
55 will use 'maccmap' (see 'help maccmap') to map cells matching the module.
56
57 When a module in the map file has the 'techmap_wrap' attribute set, techmap
58 will create a wrapper for the cell and then run the command string that the
59 attribute is set to on the wrapper module.
60
61 When a port on a module in the map file has the 'techmap_autopurge' attribute
62 set, and that port is not connected in the instantiation that is mapped, then
63 then a cell port connected only to such wires will be omitted in the mapped
64 version of the circuit.
65
```

```
66  All wires in the modules from the map file matching the pattern _TECHMAP_*
67  or *._TECHMAP_* are special wires that are used to pass instructions from
68  the mapping module to the techmap command. At the moment the following special
69  wires are supported:
70
71      _TECHMAP_FAIL_
72          When this wire is set to a non-zero constant value, techmap will not
73          use this module and instead try the next module with a matching
74          'techmap_celltype' attribute.
75
76          When such a wire exists but does not have a constant value after all
77          _TECHMAP_DO_* commands have been executed, an error is generated.
78
79      _TECHMAP_DO_*
80          This wires are evaluated in alphabetical order. The constant text value
81          of this wire is a yosys command (or sequence of commands) that is run
82          by techmap on the module. A common use case is to run 'proc' on modules
83          that are written using always-statements.
84
85          When such a wire has a non-constant value at the time it is to be
86          evaluated, an error is produced. That means it is possible for such a
87          wire to start out as non-constant and evaluate to a constant value
88          during processing of other _TECHMAP_DO_* commands.
89
90          A _TECHMAP_DO_* command may start with the special token 'CONSTMAP; '.
91          in this case techmap will create a copy for each distinct configuration
92          of constant inputs and shorted inputs at this point and import the
93          constant and connected bits into the map module. All further commands
94          are executed in this copy. This is a very convenient way of creating
95          optimized specializations of techmap modules without using the special
96          parameters described below.
97
98          A _TECHMAP_DO_* command may start with the special token 'RECURSION; '.
99          then techmap will recursively replace the cells in the module with their
100         implementation. This is not affected by the -max_iter option.
101
102         It is possible to combine both prefixes to 'RECURSION; CONSTMAP; '.
103
104     _TECHMAP_REMOVEINIT_<port-name>_
105         When this wire is set to a constant value, the init attribute of the
106         wire(s) connected to this port will be consumed.  This wire must have
107         the same width as the given port, and for every bit that is set to 1 in
108         the value, the corresponding init attribute bit will be changed to 1'bx.
109         If all bits of an init attribute are left as x, it will be removed.
110
111 In addition to this special wires, techmap also supports special parameters in
112 modules in the map file:
113
114     _TECHMAP_CELLTYPE_
115         When a parameter with this name exists, it will be set to the type name
116         of the cell that matches the module.
117
118     _TECHMAP_CELLNAME_
119         When a parameter with this name exists, it will be set to the name
```

```
120          of the cell that matches the module.
121
122      _TECHMAP_CONSTMSK_<port-name>_
123      _TECHMAP_CONSTVAL_<port-name>_
124          When this pair of parameters is available in a module for a port, then
125          former has a 1-bit for each constant input bit and the latter has the
126          value for this bit. The unused bits of the latter are set to undef (x).
127
128      _TECHMAP_WIREINIT_<port-name>_
129          When a parameter with this name exists, it will be set to the initial
130          value of the wire(s) connected to the given port, as specified by the
131          init attribute. If the attribute doesn't exist, x will be filled for the
132          missing bits.  To remove the init attribute bits used, use the
133          _TECHMAP_REMOVEINIT_*_ wires.
134
135      _TECHMAP_BITS_CONNMAP_
136      _TECHMAP_CONNMAP_<port-name>_
137          For an N-bit port, the _TECHMAP_CONNMAP_<port-name>_ parameter, if it
138          exists, will be set to an N*_TECHMAP_BITS_CONNMAP_ bit vector containing
139          N words (of _TECHMAP_BITS_CONNMAP_ bits each) that assign each single
140          bit driver a unique id. The values 0-3 are reserved for 0, 1, x, and z.
141          This can be used to detect shorted inputs.
142
143  When a module in the map file has a parameter where the according cell in the
144  design has a port, the module from the map file is only used if the port in
145  the design is connected to a constant value. The parameter is then set to the
146  constant value.
147
148  A cell with the name _TECHMAP_REPLACE_ in the map file will inherit the name
149  and attributes of the cell that is being replaced.
150  A cell with a name of the form `_TECHMAP_REPLACE_.<suffix>` in the map file will
151  be named thus but with the `_TECHMAP_REPLACE_' prefix substituted with the name
152  of the cell being replaced.
153  Similarly, a wire named in the form `_TECHMAP_REPLACE_.<suffix>` will cause a
154  new wire alias to be created and named as above but with the `_TECHMAP_REPLACE_'
155  prefix also substituted.
156
157  See 'help extract' for a pass that does the opposite thing.
158
159  See 'help flatten' for a pass that does flatten the design (which is
160  essentially techmap but using the design itself as map library).
```

C.211   tee – redirect command output to file

```
1       tee [-q] [-o logfile|-a logfile] cmd
2
3   Execute the specified command, optionally writing the commands output to the
4   specified logfile(s).
5
6       -q
7           Do not print output to the normal destination (console and/or log file).
8
```

```
 9      -o logfile
10          Write output to this file, truncate if exists.
11
12      -a logfile
13          Write output to this file, append if exists.
14
15      +INT , -INT
16          Add/subtract INT from the -v setting for this command.
```

C.212   test_abcloop – automatically test handling of loops in abc command

```
1      test_abcloop [options]
2
3  Test handling of logic loops in ABC.
4
5      -n {integer}
6          create this number of circuits and test them (default = 100).
7
8      -s {positive_integer}
9          use this value as rng seed value (default = unix time).
```

C.213   test_autotb – generate simple test benches

```
 1      test_autotb [options] [filename]
 2
 3  Automatically create primitive Verilog test benches for all modules in the
 4  design. The generated testbenches toggle the input pins of the module in
 5  a semi-random manner and dumps the resulting output signals.
 6
 7  This can be used to check the synthesis results for simple circuits by
 8  comparing the testbench output for the input files and the synthesis results.
 9
10  The backend automatically detects clock signals. Additionally a signal can
11  be forced to be interpreted as clock signal by setting the attribute
12  'gentb_clock' on the signal.
13
14  The attribute 'gentb_constant' can be used to force a signal to a constant
15  value after initialization. This can e.g. be used to force a reset signal
16  low in order to explore more inner states in a state machine.
17
18  The attribute 'gentb_skip' can be attached to modules to suppress testbench
19  generation.
20
21      -n <int>
22          number of iterations the test bench should run (default = 1000)
23
24      -seed <int>
25          seed used for pseudo-random number generation (default = 0).
```

```
26        a value of 0 will cause an arbitrary seed to be chosen, based on
27        the current system time.
```

C.214   test_cell – automatically test the implementation of a cell type

```
1      test_cell [options] {cell-types}
2
3  Tests the internal implementation of the given cell type (for example '$add')
4  by comparing SAT solver, EVAL and TECHMAP implementations of the cell types..
5
6  Run with 'all' instead of a cell type to run the test on all supported
7  cell types. Use for example 'all /$add' for all cell types except $add.
8
9      -n {integer}
10        create this number of cell instances and test them (default = 100).
11
12      -s {positive_integer}
13        use this value as rng seed value (default = unix time).
14
15      -f {rtlil_file}
16        don't generate circuits. instead load the specified RTLIL file.
17
18      -w {filename_prefix}
19        don't test anything. just generate the circuits and write them
20        to RTLIL files with the specified prefix
21
22      -map {filename}
23        pass this option to techmap.
24
25      -simlib
26        use "techmap -D SIMLIB_NOCHECKS -map +/simlib.v -max_iter 2 -autoproc"
27
28      -aigmap
29        instead of calling "techmap", call "aigmap"
30
31      -muxdiv
32        when creating test benches with dividers, create an additional mux
33        to mask out the division-by-zero case
34
35      -script {script_file}
36        instead of calling "techmap", call "script {script_file}".
37
38      -const
39        set some input bits to random constant values
40
41      -nosat
42        do not check SAT model or run SAT equivalence checking
43
44      -noeval
45        do not check const-eval models
46
```

```
47    -edges
48        test cell edges db creator against sat-based implementation
49
50    -v
51        print additional debug information to the console
52
53    -vlog {filename}
54        create a Verilog test bench to test simlib and write_verilog
```

C.215   test_pmgen – test pass for pmgen

```
 1    test_pmgen -reduce_chain [options] [selection]
 2
 3 Demo for recursive pmgen patterns. Map chains of AND/OR/XOR to $reduce_*.
 4
 5
 6    test_pmgen -reduce_tree [options] [selection]
 7
 8 Demo for recursive pmgen patterns. Map trees of AND/OR/XOR to $reduce_*.
 9
10
11    test_pmgen -eqpmux [options] [selection]
12
13 Demo for recursive pmgen patterns. Optimize EQ/NE/PMUX circuits.
14
15
16    test_pmgen -generate [options] <pattern_name>
17
18 Create modules that match the specified pattern.
```

C.216   torder – print cells in topological order

```
 1    torder [options] [selection]
 2
 3 This command prints the selected cells in topological order.
 4
 5    -stop <cell_type> <cell_port>
 6        do not use the specified cell port in topological sorting
 7
 8    -noautostop
 9        by default Q outputs of internal FF cells and memory read port outputs
10        are not used in topological sorting. this option deactivates that.
```

C.217   trace – redirect command output to file

```
 1    trace cmd
 2
```

```
 3   Execute the specified command, logging all changes the command performs on
 4   the design in real time.
```

C.218   tribuf – infer tri-state buffers

```
 1       tribuf [options] [selection]
 2
 3   This pass transforms $mux cells with 'z' inputs to tristate buffers.
 4
 5       -merge
 6           merge multiple tri-state buffers driving the same net
 7           into a single buffer.
 8
 9       -logic
10           convert tri-state buffers that do not drive output ports
11           to non-tristate logic. this option implies -merge.
12
13       -formal
14           convert all tri-state buffers to non-tristate logic and
15           add a formal assertion that no two buffers are driving the
16           same net simultaneously. this option implies -merge.
```

C.219   uniquify – create unique copies of modules

```
 1       uniquify [selection]
 2
 3   By default, a module that is instantiated by several other modules is only
 4   kept once in the design. This preserves the original modularity of the design
 5   and reduces the overall size of the design in memory. But it prevents certain
 6   optimizations and other operations on the design. This pass creates unique
 7   modules for all selected cells. The created modules are marked with the
 8   'unique' attribute.
 9
10   This commands only operates on modules that by themself have the 'unique'
11   attribute set (the 'top' module is unique implicitly).
```

C.220   verific – load Verilog and VHDL designs using Verific

```
 1       verific {-vlog95|-vlog2k|-sv2005|-sv2009|-sv2012|-sv} <verilog-file>..
 2
 3   Load the specified Verilog/SystemVerilog files into Verific.
 4
 5   All files specified in one call to this command are one compilation unit.
 6   Files passed to different calls to this command are treated as belonging to
 7   different compilation units.
 8
 9   Additional -D<macro>[=<value>] options may be added after the option indicating
```

```
10  the language version (and before file names) to set additional verilog defines.
11  The macros YOSYS, SYNTHESIS, and VERIFIC are defined implicitly.
12
13
14      verific -formal <verilog-file>..
15
16  Like -sv, but define FORMAL instead of SYNTHESIS.
17
18
19      verific {-vhdl87|-vhdl93|-vhdl2k|-vhdl2008|-vhdl} <vhdl-file>..
20
21  Load the specified VHDL files into Verific.
22
23
24      verific {-edif} <edif-file>..
25
26  Load the specified EDIF files into Verific.
27
28
29      verific {-liberty} <liberty-file>..
30
31  Load the specified Liberty files into Verific.
32  Default library when -work is not present is one specified in liberty file.
33  To use from SystemVerilog or VHDL use -L to specify liberty library.
34      -lib
35          only create empty blackbox modules
36
37
38      verific {-f|-F} [-vlog95|-vlog2k|-sv2005|-sv2009|
39                      -sv2012|-sv|-formal] <command-file>
40
41  Load and execute the specified command file.
42  Override verilog parsing mode can be set.
43  The macros YOSYS, SYNTHESIS/FORMAL, and VERIFIC are defined implicitly.
44
45  Command file parser supports following commands in file:
46      +define+<MACRO>=<VALUE> - defines macro
47      -u                      - upper case all identifier (makes Verilog parser
48                                 case insensitive)
49      -v <filepath>           - register library name (file)
50      -y <filepath>           - register library name (directory)
51      +incdir+<filepath>      - specify include dir
52      +libext+<filepath>      - specify library extension
53      +liborder+<id>          - add library in ordered list
54      +librescan              - unresolved modules will be always searched
55                                 starting with the first library specified
56                                 by -y/-v options.
57      -f/-file <filepath>     - nested -f option
58      -F <filepath>           - nested -F option (relative path)
59      parse files:
60          <filepath>
61          +systemverilogext+<filepath>
62          +verilog1995ext+<filepath>
63          +verilog2001ext+<filepath>
```

```
 64
 65       analysis mode:
 66           -ams
 67           +v2k
 68           -sverilog
 69
 70
 71       verific [-work <libname>] {-sv|-vhdl|...} <hdl-file>
 72
 73   Load the specified Verilog/SystemVerilog/VHDL file into the specified library.
 74   (default library when -work is not present: "work")
 75
 76
 77       verific [-L <libname>] {-sv|-vhdl|...} <hdl-file>
 78
 79   Look up external definitions in the specified library.
 80   (-L may be used more than once)
 81
 82
 83       verific -vlog-incdir <directory>..
 84
 85   Add Verilog include directories.
 86
 87
 88       verific -vlog-libdir <directory>..
 89
 90   Add Verilog library directories. Verific will search in this directories to
 91   find undefined modules.
 92
 93
 94       verific -vlog-libext <extension>..
 95
 96   Add Verilog library extensions, used when searching in library directories.
 97
 98
 99       verific -vlog-define <macro>[=<value>]..
100
101   Add Verilog defines.
102
103
104       verific -vlog-undef <macro>..
105
106   Remove Verilog defines previously set with -vlog-define.
107
108
109       verific -set-error <msg_id>..
110       verific -set-warning <msg_id>..
111       verific -set-info <msg_id>..
112       verific -set-ignore <msg_id>..
113
114   Set message severity. <msg_id> is the string in square brackets when a message
115   is printed, such as VERI-1209.
116
117
```

```
118        verific -import [options] <top-module>..
119
120   Elaborate the design for the specified top modules, import to Yosys and
121   reset the internal state of Verific.
122
123   Import options:
124
125     -all
126       Elaborate all modules, not just the hierarchy below the given top
127       modules. With this option the list of modules to import is optional.
128
129     -gates
130       Create a gate-level netlist.
131
132     -flatten
133       Flatten the design in Verific before importing.
134
135     -extnets
136       Resolve references to external nets by adding module ports as needed.
137
138     -autocover
139       Generate automatic cover statements for all asserts
140
141     -fullinit
142       Keep all register initializations, even those for non-FF registers.
143
144     -cells
145       Import all cell definitions from Verific loaded libraries even if they are
146       unused in design. Useful with "-edif" and "-liberty" option.
147
148     -chparam name value
149       Elaborate the specified top modules (all modules when -all given) using
150       this parameter value. Modules on which this parameter does not exist will
151       cause Verific to produce a VERI-1928 or VHDL-1676 message. This option
152       can be specified multiple times to override multiple parameters.
153       String values must be passed in double quotes (").
154
155     -v, -vv
156       Verbose log messages. (-vv is even more verbose than -v.)
157
158     -pp <filename>
159       Pretty print design after elaboration to specified file.
160
161   The following additional import options are useful for debugging the Verific
162   bindings (for Yosys and/or Verific developers):
163
164     -k
165       Keep going after an unsupported verific primitive is found. The
166       unsupported primitive is added as blockbox module to the design.
167       This will also add all SVA related cells to the design parallel to
168       the checker logic inferred by it.
169
170     -V
171       Import Verific netlist as-is without translating to Yosys cell types.
```

```
172
173    -nosva
174      Ignore SVA properties, do not infer checker logic.
175
176    -L <int>
177      Maximum number of ctrl bits for SVA checker FSMs (default=16).
178
179    -n
180      Keep all Verific names on instances and nets. By default only
181      user-declared names are preserved.
182
183    -d <dump_file>
184      Dump the Verific netlist as a verilog file.
185
186
187      verific [-work <libname>] -pp [options] <filename> [<module>]..
188
189  Pretty print design (or just module) to the specified file from the
190  specified library. (default library when -work is not present: "work")
191
192  Pretty print options:
193
194    -verilog
195      Save output for Verilog/SystemVerilog design modules (default).
196
197    -vhdl
198      Save output for VHDL design units.
199
200
201      verific -cfg [<name> [<value>]]
202
203  Get/set Verific runtime flags.
204
205
206  Use YosysHQ Tabby CAD Suite if you need Yosys+Verific.
207  https://www.yosyshq.com/
208
209  Contact office@yosyshq.com for free evaluation
210  binaries of YosysHQ Tabby CAD Suite.
```

C.221   verilog_defaults – set default options for read_verilog

```
1      verilog_defaults -add [options]
2
3  Add the specified options to the list of default options to read_verilog.
4
5
6      verilog_defaults -clear
7
8  Clear the list of Verilog default options.
9
10
```

```
11        verilog_defaults -push
12        verilog_defaults -pop
13
14   Push or pop the list of default options to a stack. Note that -push does
15   not imply -clear.
```

C.222   verilog_defines – define and undefine verilog defines

```
 1        verilog_defines [options]
 2
 3   Define and undefine verilog preprocessor macros.
 4
 5       -Dname[=definition]
 6           define the preprocessor symbol 'name' and set its optional value
 7           'definition'
 8
 9       -Uname[=definition]
10           undefine the preprocessor symbol 'name'
11
12       -reset
13           clear list of defined preprocessor symbols
14
15       -list
16           list currently defined preprocessor symbols
```

C.223   wbflip – flip the whitebox attribute

```
 1        wbflip [selection]
 2
 3   Flip the whitebox attribute on selected cells. I.e. if it's set, unset it, and
 4   vice-versa. Blackbox cells are not effected by this command.
```

C.224   wreduce – reduce the word size of operations if possible

```
 1        wreduce [options] [selection]
 2
 3   This command reduces the word size of operations. For example it will replace
 4   the 32 bit adders in the following code with adders of more appropriate widths:
 5
 6       module test(input [3:0] a, b, c, output [7:0] y);
 7           assign y = a + b + c + 1;
 8       endmodule
 9
10   Options:
11
12       -memx
13           Do not change the width of memory address ports. Use this options in
```

```
14          flows that use the 'memory_memx' pass.
15
16      -mux_undef
17          remove 'undef' inputs from $mux, $pmux and $_MUX_ cells
18
19      -keepdc
20          Do not optimize explicit don't-care values.
```

C.225 write_aiger – write design to AIGER file

```
 1      write_aiger [options] [filename]
 2
 3  Write the current design to an AIGER file. The design must be flattened and
 4  must not contain any cell types except $_AND_, $_NOT_, simple FF types,
 5  $assert and $assume cells, and $initstate cells.
 6
 7  $assert and $assume cells are converted to AIGER bad state properties and
 8  invariant constraints.
 9
10      -ascii
11          write ASCII version of AIGER format
12
13      -zinit
14          convert FFs to zero-initialized FFs, adding additional inputs for
15          uninitialized FFs.
16
17      -miter
18          design outputs are AIGER bad state properties
19
20      -symbols
21          include a symbol table in the generated AIGER file
22
23      -map <filename>
24          write an extra file with port and latch symbols
25
26      -vmap <filename>
27          like -map, but more verbose
28
29      -no-startoffset
30          make indexes zero based, enable using map files with smt solvers.
31
32      -ywmap <filename>
33          write a map file for conversion to and from yosys witness traces.
34
35      -I, -O, -B, -L
36          If the design contains no input/output/assert/flip-flop then create one
37          dummy input/output/bad_state-pin or latch to make the tools reading the
38          AIGER file happy.
```

C.226 write_blif – write design to BLIF file

```
     write_blif [options] [filename]

Write the current design to an BLIF file.

    -top top_module
        set the specified module as design top module

    -buf <cell-type> <in-port> <out-port>
        use cells of type <cell-type> with the specified port names for buffers

    -unbuf <cell-type> <in-port> <out-port>
        replace buffer cells with the specified name and port names with
        a .names statement that models a buffer

    -true <cell-type> <out-port>
    -false <cell-type> <out-port>
    -undef <cell-type> <out-port>
        use the specified cell types to drive nets that are constant 1, 0, or
        undefined. when '-' is used as <cell-type>, then <out-port> specifies
        the wire name to be used for the constant signal and no cell driving
        that wire is generated. when '+' is used as <cell-type>, then <out-port>
        specifies the wire name to be used for the constant signal and a .names
        statement is generated to drive the wire.

    -noalias
        if a net name is aliasing another net name, then by default a net
        without fanout is created that is driven by the other net. This option
        suppresses the generation of this nets without fanout.

The following options can be useful when the generated file is not going to be
read by a BLIF parser but a custom tool. It is recommended to not name the
output file *.blif when any of this options is used.

    -icells
        do not translate Yosys's internal gates to generic BLIF logic
        functions. Instead create .subckt or .gate lines for all cells.

    -gates
        print .gate instead of .subckt lines for all cells that are not
        instantiations of other modules from this design.

    -conn
        do not generate buffers for connected wires. instead use the
        non-standard .conn statement.

    -attr
        use the non-standard .attr statement to write cell attributes

    -param
        use the non-standard .param statement to write cell parameters

    -cname
        use the non-standard .cname statement to write cell names
```

```
55      -iname, -iattr
56          enable -cname and -attr functionality for .names statements
57          (the .cname and .attr statements will be included in the BLIF
58          output after the truth table for the .names statement)
59
60      -blackbox
61          write blackbox cells with .blackbox statement.
62
63      -impltf
64          do not write definitions for the $true, $false and $undef wires.
```

C.227   write_btor – write design to BTOR file

```
1       write_btor [options] [filename]
2
3   Write a BTOR description of the current design.
4
5     -v
6       Add comments and indentation to BTOR output file
7
8     -s
9       Output only a single bad property for all asserts
10
11    -c
12      Output cover properties using 'bad' statements instead of asserts
13
14    -i <filename>
15      Create additional info file with auxiliary information
16
17    -x
18      Output symbols for internal netnames (starting with '$')
```

C.228   write_cxxrtl – convert design to C++ RTL simulation

```
1       write_cxxrtl [options] [filename]
2
3   Write C++ code that simulates the design. The generated code requires a driver
4   that instantiates the design, toggles its clock, and interacts with its ports.
5
6   The following driver may be used as an example for a design with a single clock
7   driving rising edge triggered flip-flops:
8
9       #include "top.cc"
10
11      int main() {
12        cxxrtl_design::p_top top;
13        top.step();
14        while (1) {
15          /* user logic */
16          top.p_clk.set(false);
```

```
17        top.step();
18        top.p_clk.set(true);
19        top.step();
20      }
21    }
```

Note that CXXRTL simulations, just like the hardware they are simulating, are
subject to race conditions. If, in the example above, the user logic would run
simultaneously with the rising edge of the clock, the design would malfunction.

This backend supports replacing parts of the design with black boxes implemented
in C++. If a module marked as a CXXRTL black box, its implementation is ignored,
and the generated code consists only of an interface and a factory function.
The driver must implement the factory function that creates an implementation of
the black box, taking into account the parameters it is instantiated with.

For example, the following Verilog code defines a CXXRTL black box interface for
a synchronous debug sink:

```
    (* cxxrtl_blackbox *)
    module debug(...);
      (* cxxrtl_edge = "p" *) input clk;
      input en;
      input [7:0] i_data;
      (* cxxrtl_sync *) output [7:0] o_data;
    endmodule
```

For this HDL interface, this backend will generate the following C++ interface:

```
    struct bb_p_debug : public module {
      value<1> p_clk;
      bool posedge_p_clk() const { /* ... */ }
      value<1> p_en;
      value<8> p_i_data;
      wire<8> p_o_data;

      bool eval() override;
      bool commit() override;

      static std::unique_ptr<bb_p_debug>
      create(std::string name, metadata_map parameters, metadata_map attributes);
    };
```

The `create' function must be implemented by the driver. For example, it could
always provide an implementation logging the values to standard error stream:

```
    namespace cxxrtl_design {

    struct stderr_debug : public bb_p_debug {
      bool eval() override {
        if (posedge_p_clk() && p_en)
          fprintf(stderr, "debug: %02x\n", p_i_data.data[0]);
        p_o_data.next = p_i_data;
        return bb_p_debug::eval();
```

```
71        }
72      };
73
74      std::unique_ptr<bb_p_debug>
75      bb_p_debug::create(std::string name, cxxrtl::metadata_map parameters,
76                         cxxrtl::metadata_map attributes) {
77        return std::make_unique<stderr_debug>();
78      }
79
80      }
```

For complex applications of black boxes, it is possible to parameterize their
port widths. For example, the following Verilog code defines a CXXRTL black box
interface for a configurable width debug sink:

```
    (* cxxrtl_blackbox, cxxrtl_template = "WIDTH" *)
    module debug(...);
      parameter WIDTH = 8;
      (* cxxrtl_edge = "p" *) input clk;
      input en;
      (* cxxrtl_width = "WIDTH" *) input [WIDTH - 1:0] i_data;
      (* cxxrtl_width = "WIDTH" *) output [WIDTH - 1:0] o_data;
    endmodule
```

For this parametric HDL interface, this backend will generate the following C++
interface (only the differences are shown):

```
    template<size_t WIDTH>
    struct bb_p_debug : public module {
      // ...
      value<WIDTH> p_i_data;
      wire<WIDTH> p_o_data;
      // ...
      static std::unique_ptr<bb_p_debug<WIDTH>>
      create(std::string name, metadata_map parameters, metadata_map attributes);
    };
```

The `create' function must be implemented by the driver, specialized for every
possible combination of template parameters. (Specialization is necessary to
enable separate compilation of generated code and black box implementations.)

```
    template<size_t SIZE>
    struct stderr_debug : public bb_p_debug<SIZE> {
      // ...
    };

    template<>
    std::unique_ptr<bb_p_debug<8>>
    bb_p_debug<8>::create(std::string name, cxxrtl::metadata_map parameters,
                          cxxrtl::metadata_map attributes) {
      return std::make_unique<stderr_debug<8>>();
    }
```

The following attributes are recognized by this backend:

```
125
126     cxxrtl_blackbox
127         only valid on modules. if specified, the module contents are ignored,
128         and the generated code includes only the module interface and a factory
129         function, which will be called to instantiate the module.
130
131     cxxrtl_edge
132         only valid on inputs of black boxes. must be one of "p", "n", "a".
133         if specified on signal `clk`, the generated code includes edge detectors
134         `posedge_p_clk()` (if "p"), `negedge_p_clk()` (if "n"), or both (if
135         "a"), simplifying implementation of clocked black boxes.
136
137     cxxrtl_template
138         only valid on black boxes. must contain a space separated sequence of
139         identifiers that have a corresponding black box parameters. for each
140         of them, the generated code includes a `size_t` template parameter.
141
142     cxxrtl_width
143         only valid on ports of black boxes. must be a constant expression, which
144         is directly inserted into generated code.
145
146     cxxrtl_comb, cxxrtl_sync
147         only valid on outputs of black boxes. if specified, indicates that every
148         bit of the output port is driven, correspondingly, by combinatorial or
149         synchronous logic. this knowledge is used for scheduling optimizations.
150         if neither is specified, the output will be pessimistically treated as
151         driven by both combinatorial and synchronous logic.
152
153 The following options are supported by this backend:
154
155     -print-wire-types, -print-debug-wire-types
156         enable additional debug logging, for pass developers.
157
158     -header
159         generate separate interface (.h) and implementation (.cc) files.
160         if specified, the backend must be called with a filename, and filename
161         of the interface is derived from filename of the implementation.
162         otherwise, interface and implementation are generated together.
163
164     -namespace <ns-name>
165         place the generated code into namespace <ns-name>. if not specified,
166         "cxxrtl_design" is used.
167
168     -nohierarchy
169         use design hierarchy as-is. in most designs, a top module should be
170         present as it is exposed through the C API and has unbuffered outputs
171         for improved performance; it will be determined automatically if absent.
172
173     -noflatten
174         don't flatten the design. fully flattened designs can evaluate within
175         one delta cycle if they have no combinatorial feedback.
176         note that the debug interface and waveform dumps use full hierarchical
177         names for all wires even in flattened designs.
178
```

```
179    -noproc
180        don't convert processes to netlists. in most designs, converting
181        processes significantly improves evaluation performance at the cost of
182        slight increase in compilation time.
183
184    -O <level>
185        set the optimization level. the default is -O6. higher optimization
186        levels dramatically decrease compile and run time, and highest level
187        possible for a design should be used.
188
189    -O0
190        no optimization.
191
192    -O1
193        unbuffer internal wires if possible.
194
195    -O2
196        like -O1, and localize internal wires if possible.
197
198    -O3
199        like -O2, and inline internal wires if possible.
200
201    -O4
202        like -O3, and unbuffer public wires not marked (*keep*) if possible.
203
204    -O5
205        like -O4, and localize public wires not marked (*keep*) if possible.
206
207    -O6
208        like -O5, and inline public wires not marked (*keep*) if possible.
209
210    -g <level>
211        set the debug level. the default is -g4. higher debug levels provide
212        more visibility and generate more code, but do not pessimize evaluation.
213
214    -g0
215        no debug information. the C API is disabled.
216
217    -g1
218        include bare minimum of debug information necessary to access all design
219        state. the C API is enabled.
220
221    -g2
222        like -g1, but include debug information for all public wires that are
223        directly accessible through the C++ interface.
224
225    -g3
226        like -g2, and include debug information for public wires that are tied
227        to a constant or another public wire.
228
229    -g4
230        like -g3, and compute debug information on demand for all public wires
231        that were optimized out.
```

C.229   write_edif – write design to EDIF netlist file

```
 1      write_edif [options] [filename]
 2
 3  Write the current design to an EDIF netlist file.
 4
 5      -top top_module
 6          set the specified module as design top module
 7
 8      -nogndvcc
 9          do not create "GND" and "VCC" cells. (this will produce an error
10          if the design contains constant nets. use "hilomap" to map to custom
11          constant drivers first)
12
13      -gndvccy
14          create "GND" and "VCC" cells with "Y" outputs. (the default is
15          "G" for "GND" and "P" for "VCC".)
16
17      -attrprop
18          create EDIF properties for cell attributes
19
20      -keep
21          create extra KEEP nets by allowing a cell to drive multiple nets.
22
23      -pvector {par|bra|ang}
24          sets the delimiting character for module port rename clauses to
25          parentheses, square brackets, or angle brackets.
26
27  Unfortunately there are different "flavors" of the EDIF file format. This
28  command generates EDIF files for the Xilinx place&route tools. It might be
29  necessary to make small modifications to this command when a different tool
30  is targeted.
```

C.230   write_file – write a text to a file

```
 1      write_file [options] output_file [input_file]
 2
 3  Write the text from the input file to the output file.
 4
 5      -a
 6          Append to output file (instead of overwriting)
 7
 8
 9  Inside a script the input file can also can a here-document:
10
11      write_file hello.txt <<EOT
12      Hello World!
13      EOT
```

C.231 write_firrtl – write design to a FIRRTL file

```
1      write_firrtl [options] [filename]
2
3  Write a FIRRTL netlist of the current design.
4  The following commands are executed by this command:
5          pmuxtree
6          bmuxmap
7          demuxmap
```

C.232 write_ilang – (deprecated) alias of write_rtlil

```
1  See `help write_rtlil`.
```

C.233 write_intersynth – write design to InterSynth netlist file

```
1      write_intersynth [options] [filename]
2
3  Write the current design to an 'intersynth' netlist file. InterSynth is
4  a tool for Coarse-Grain Example-Driven Interconnect Synthesis.
5
6      -notypes
7          do not generate celltypes and conntypes commands. i.e. just output
8          the netlists. this is used for postsilicon synthesis.
9
10     -lib <verilog_or_rtlil_file>
11         Use the specified library file for determining whether cell ports are
12         inputs or outputs. This option can be used multiple times to specify
13         more than one library.
14
15     -selected
16         only write selected modules. modules must be selected entirely or
17         not at all.
18
19 http://bygone.clairexen.net/intersynth/
```

C.234 write_jny – generate design metadata

```
1      jny [options] [selection]
2
3  Write JSON netlist metadata for the current design
4
5      -no-connections
6          Don't include connection information in the netlist output.
7
8      -no-attributes
9          Don't include attributed information in the netlist output.
```

```
10
11     -no-properties
12         Don't include property information in the netlist output.
13
14 The JSON schema for JNY output files is located in the "jny.schema.json" file
15 which is located at "https://raw.githubusercontent.com/YosysHQ/yosys/master/misc/jny.schema.json"
```

C.235   write_json – write design to a JSON file

```
1      write_json [options] [filename]
2
3  Write a JSON netlist of the current design.
4
5      -aig
6          include AIG models for the different gate types
7
8      -compat-int
9          emit 32-bit or smaller fully-defined parameter values directly
10         as JSON numbers (for compatibility with old parsers)
11
12
13 The general syntax of the JSON output created by this command is as follows:
14
15     {
16       "creator": "Yosys <version info>",
17       "modules": {
18         <module_name>: {
19           "attributes": {
20             <attribute_name>: <attribute_value>,
21             ...
22           },
23           "parameter_default_values": {
24             <parameter_name>: <parameter_value>,
25             ...
26           },
27           "ports": {
28             <port_name>: <port_details>,
29             ...
30           },
31           "cells": {
32             <cell_name>: <cell_details>,
33             ...
34           },
35           "memories": {
36             <memory_name>: <memory_details>,
37             ...
38           },
39           "netnames": {
40             <net_name>: <net_details>,
41             ...
42           }
43         }
```

```
44          },
45          "models": {
46              ...
47          },
48      }
49
50  Where <port_details> is:
51
52      {
53          "direction": <"input" | "output" | "inout">,
54          "bits": <bit_vector>
55          "offset": <the lowest bit index in use, if non-0>
56          "upto": <1 if the port bit indexing is MSB-first>
57          "signed": <1 if the port is signed>
58      }
59
60  The "offset" and "upto" fields are skipped if their value would be 0.
61  They don't affect connection semantics, and are only used to preserve original
62  HDL bit indexing.And <cell_details> is:
63
64      {
65          "hide_name": <1 | 0>,
66          "type": <cell_type>,
67          "model": <AIG model name, if -aig option used>,
68          "parameters": {
69              <parameter_name>: <parameter_value>,
70              ...
71          },
72          "attributes": {
73              <attribute_name>: <attribute_value>,
74              ...
75          },
76          "port_directions": {
77              <port_name>: <"input" | "output" | "inout">,
78              ...
79          },
80          "connections": {
81              <port_name>: <bit_vector>,
82              ...
83          },
84      }
85
86  And <memory_details> is:
87
88      {
89          "hide_name": <1 | 0>,
90          "attributes": {
91              <attribute_name>: <attribute_value>,
92              ...
93          },
94          "width": <memory width>
95          "start_offset": <the lowest valid memory address>
96          "size": <memory size>
97      }
```

```
 98
 99   And <net_details> is:
100
101       {
102           "hide_name": <1 | 0>,
103           "bits": <bit_vector>
104           "offset": <the lowest bit index in use, if non-0>
105           "upto": <1 if the port bit indexing is MSB-first>
106           "signed": <1 if the port is signed>
107       }
108
109   The "hide_name" fields are set to 1 when the name of this cell or net is
110   automatically created and is likely not of interest for a regular user.
111
112   The "port_directions" section is only included for cells for which the
113   interface is known.
114
115   Module and cell ports and nets can be single bit wide or vectors of multiple
116   bits. Each individual signal bit is assigned a unique integer. The <bit_vector>
117   values referenced above are vectors of this integers. Signal bits that are
118   connected to a constant driver are denoted as string "0", "1", "x", or
119   "z" instead of a number.
120
121   Bit vectors (including integers) are written as string holding the binary
122   representation of the value. Strings are written as strings, with an appended
123   blank in cases of strings of the form /[01xz]* */.
124
125   For example the following Verilog code:
126
127       module test(input x, y);
128         (* keep *) foo #(.P(42), .Q(1337))
129             foo_inst (.A({x, y}), .B({y, x}), .C({4'd10, {4{x}}}));
130       endmodule
131
132   Translates to the following JSON output:
133
134       {
135         "creator": "Yosys 0.9+2406 (git sha1 fb1168d8, clang 9.0.1 -fPIC -Os)",
136         "modules": {
137           "test": {
138             "attributes": {
139               "cells_not_processed": "00000000000000000000000000000001",
140               "src": "test.v:1.1-4.10"
141             },
142             "ports": {
143               "x": {
144                 "direction": "input",
145                 "bits": [ 2 ]
146               },
147               "y": {
148                 "direction": "input",
149                 "bits": [ 3 ]
150               }
151             },
```

```
152              "cells": {
153                "foo_inst": {
154                  "hide_name": 0,
155                  "type": "foo",
156                  "parameters": {
157                    "P": "00000000000000000000000000101010",
158                    "Q": "000000000000000000000010100111001"
159                  },
160                  "attributes": {
161                    "keep": "00000000000000000000000000000001",
162                    "module_not_derived": "000000000000000000000000000000001",
163                    "src": "test.v:3.1-3.55"
164                  },
165                  "connections": {
166                    "A": [ 3, 2 ],
167                    "B": [ 2, 3 ],
168                    "C": [ 2, 2, 2, 2, "0", "1", "0", "1" ]
169                  }
170                }
171              },
172              "netnames": {
173                "x": {
174                  "hide_name": 0,
175                  "bits": [ 2 ],
176                  "attributes": {
177                    "src": "test.v:1.19-1.20"
178                  }
179                },
180                "y": {
181                  "hide_name": 0,
182                  "bits": [ 3 ],
183                  "attributes": {
184                    "src": "test.v:1.22-1.23"
185                  }
186                }
187              }
188            }
189          }
190        }
191
192 The models are given as And-Inverter-Graphs (AIGs) in the following form:
193
194      "models": {
195        <model_name>: [
196          /*   0 */ [ <node-spec> ],
197          /*   1 */ [ <node-spec> ],
198          /*   2 */ [ <node-spec> ],
199          ...
200        ],
201        ...
202      },
203
204 The following node-types may be used:
205
```

```
206        [ "port", <portname>, <bitindex>, <out-list> ]
207          - the value of the specified input port bit
208
209        [ "nport", <portname>, <bitindex>, <out-list> ]
210          - the inverted value of the specified input port bit
211
212        [ "and", <node-index>, <node-index>, <out-list> ]
213          - the ANDed value of the specified nodes
214
215        [ "nand", <node-index>, <node-index>, <out-list> ]
216          - the inverted ANDed value of the specified nodes
217
218        [ "true", <out-list> ]
219          - the constant value 1
220
221        [ "false", <out-list> ]
222          - the constant value 0
223
224    All nodes appear in topological order. I.e. only nodes with smaller indices
225    are referenced by "and" and "nand" nodes.
226
227    The optional <out-list> at the end of a node specification is a list of
228    output portname and bitindex pairs, specifying the outputs driven by this node.
229
230    For example, the following is the model for a 3-input 3-output $reduce_and cell
231    inferred by the following code:
232
233        module test(input [2:0] in, output [2:0] out);
234          assign in = &out;
235        endmodule
236
237        "$reduce_and:3U:3": [
238          /*   0 */ [ "port", "A", 0 ],
239          /*   1 */ [ "port", "A", 1 ],
240          /*   2 */ [ "and", 0, 1 ],
241          /*   3 */ [ "port", "A", 2 ],
242          /*   4 */ [ "and", 2, 3, "Y", 0 ],
243          /*   5 */ [ "false", "Y", 1, "Y", 2 ]
244        ]
245
246    Future version of Yosys might add support for additional fields in the JSON
247    format. A program processing this format must ignore all unknown fields.
```

C.236   write_rtlil – write design to RTLIL file

```
1        write_rtlil [filename]
2
3    Write the current design to an RTLIL file. (RTLIL is a text representation
4    of a design in yosys's internal format.)
5
6        -selected
7            only write selected parts of the design.
```

C.237   write_simplec – convert design to simple C code

```
1      write_simplec [options] [filename]
2
3  Write simple C code for simulating the design. The C code written can be used to
4  simulate the design in a C environment, but the purpose of this command is to
5  generate code that works well with C-based formal verification.
6
7      -verbose
8          this will print the recursive walk used to export the modules.
9
10     -i8, -i16, -i32, -i64
11         set the maximum integer bit width to use in the generated code.
12
13 THIS COMMAND IS UNDER CONSTRUCTION
```

C.238   write_smt2 – write design to SMT-LIBv2 file

```
1      write_smt2 [options] [filename]
2
3  Write a SMT-LIBv2 [1] description of the current design. For a module with name
4  '<mod>' this will declare the sort '<mod>_s' (state of the module) and will
5  define and declare functions operating on that state.
6
7  The following SMT2 functions are generated for a module with name '<mod>'.
8  Some declarations/definitions are printed with a special comment. A prover
9  using the SMT2 files can use those comments to collect all relevant metadata
10 about the design.
11
12     ; yosys-smt2-module <mod>
13     (declare-sort |<mod>_s| 0)
14         The sort representing a state of module <mod>.
15
16     (define-fun |<mod>_h| ((state |<mod>_s|)) Bool (...))
17         This function must be asserted for each state to establish the
18         design hierarchy.
19
20     ; yosys-smt2-input <wirename> <width>
21     ; yosys-smt2-output <wirename> <width>
22     ; yosys-smt2-register <wirename> <width>
23     ; yosys-smt2-wire <wirename> <width>
24     (define-fun |<mod>_n <wirename>| (|<mod>_s|) (_ BitVec <width>))
25     (define-fun |<mod>_n <wirename>| (|<mod>_s|) Bool)
26         For each port, register, and wire with the 'keep' attribute set an
27         accessor function is generated. Single-bit wires are returned as Bool,
28         multi-bit wires as BitVec.
29
30     ; yosys-smt2-cell <submod> <instancename>
31     (declare-fun |<mod>_h <instancename>| (|<mod>_s|) |<submod>_s|)
32         There is a function like that for each hierarchical instance. It
33         returns the sort that represents the state of the sub-module that
34         implements the instance.
```

```
35
36      (declare-fun |<mod>_is| (|<mod>_s|) Bool)
37          This function must be asserted 'true' for initial states, and 'false'
38          otherwise.
39
40      (define-fun |<mod>_i| ((state |<mod>_s|)) Bool (...))
41          This function must be asserted 'true' for initial states. For
42          non-initial states it must be left unconstrained.
43
44      (define-fun |<mod>_t| ((state |<mod>_s|) (next_state |<mod>_s|)) Bool (...))
45          This function evaluates to 'true' if the states 'state' and
46          'next_state' form a valid state transition.
47
48      (define-fun |<mod>_a| ((state |<mod>_s|)) Bool (...))
49          This function evaluates to 'true' if all assertions hold in the state.
50
51      (define-fun |<mod>_u| ((state |<mod>_s|)) Bool (...))
52          This function evaluates to 'true' if all assumptions hold in the state.
53
54      ; yosys-smt2-assert <id> <filename:linenum>
55      (define-fun |<mod>_a <id>| ((state |<mod>_s|)) Bool (...))
56          Each $assert cell is converted into one of this functions. The function
57          evaluates to 'true' if the assert statement holds in the state.
58
59      ; yosys-smt2-assume <id> <filename:linenum>
60      (define-fun |<mod>_u <id>| ((state |<mod>_s|)) Bool (...))
61          Each $assume cell is converted into one of this functions. The function
62          evaluates to 'true' if the assume statement holds in the state.
63
64      ; yosys-smt2-cover <id> <filename:linenum>
65      (define-fun |<mod>_c <id>| ((state |<mod>_s|)) Bool (...))
66          Each $cover cell is converted into one of this functions. The function
67          evaluates to 'true' if the cover statement is activated in the state.
68
69  Options:
70
71      -verbose
72          this will print the recursive walk used to export the modules.
73
74      -stbv
75          Use a BitVec sort to represent a state instead of an uninterpreted
76          sort. As a side-effect this will prevent use of arrays to model
77          memories.
78
79      -stdt
80          Use SMT-LIB 2.6 style datatypes to represent a state instead of an
81          uninterpreted sort.
82
83      -nobv
84          disable support for BitVec (FixedSizeBitVectors theory). without this
85          option multi-bit wires are represented using the BitVec sort and
86          support for coarse grain cells (incl. arithmetic) is enabled.
87
88      -nomem
```

```
 89          disable support for memories (via ArraysEx theory). this option is
 90          implied by -nobv. only $mem cells without merged registers in
 91          read ports are supported. call "memory" with -nordff to make sure
 92          that no registers are merged into $mem read ports. '<mod>_m' functions
 93          will be generated for accessing the arrays that are used to represent
 94          memories.
 95
 96      -wires
 97          create '<mod>_n' functions for all public wires. by default only ports,
 98          registers, and wires with the 'keep' attribute are exported.
 99
100      -tpl <template_file>
101          use the given template file. the line containing only the token '%%'
102          is replaced with the regular output of this command.
103
104      -solver-option <option> <value>
105          emit a `; yosys-smt2-solver-option` directive for yosys-smtbmc to write
106          the given option as a `(set-option ...)` command in the SMT-LIBv2.
107
108  [1] For more information on SMT-LIBv2 visit http://smt-lib.org/ or read David
109  R. Cok's tutorial: https://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf
110
111  -------------------------------------------------------------------------
112
113  Example:
114
115  Consider the following module (test.v). We want to prove that the output can
116  never transition from a non-zero value to a zero value.
117
118          module test(input clk, output reg [3:0] y);
119            always @(posedge clk)
120              y <= (y << 1) | ^y;
121          endmodule
122
123  For this proof we create the following template (test.tpl).
124
125          ; we need QF_UFBV for this proof
126          (set-logic QF_UFBV)
127
128          ; insert the auto-generated code here
129          %%
130
131          ; declare two state variables s1 and s2
132          (declare-fun s1 () test_s)
133          (declare-fun s2 () test_s)
134
135          ; state s2 is the successor of state s1
136          (assert (test_t s1 s2))
137
138          ; we are looking for a model with y non-zero in s1
139          (assert (distinct (|test_n y| s1) #b0000))
140
141          ; we are looking for a model with y zero in s2
142          (assert (= (|test_n y| s2) #b0000))
```

```
143
144          ; is there such a model?
145          (check-sat)
146
147 The following yosys script will create a 'test.smt2' file for our proof:
148
149          read_verilog test.v
150          hierarchy -check; proc; opt; check -assert
151          write_smt2 -bv -tpl test.tpl test.smt2
152
153 Running 'cvc4 test.smt2' will print 'unsat' because y can never transition
154 from non-zero to zero in the test design.
```

C.239   write_smv − write design to SMV file

```
1       write_smv [options] [filename]
2
3  Write an SMV description of the current design.
4
5      -verbose
6          this will print the recursive walk used to export the modules.
7
8      -tpl <template_file>
9          use the given template file. the line containing only the token '%%'
10         is replaced with the regular output of this command.
11
12 THIS COMMAND IS UNDER CONSTRUCTION
```

C.240   write_spice − write design to SPICE netlist file

```
1       write_spice [options] [filename]
2
3  Write the current design to an SPICE netlist file.
4
5      -big_endian
6          generate multi-bit ports in MSB first order
7          (default is LSB first)
8
9      -neg net_name
10         set the net name for constant 0 (default: Vss)
11
12     -pos net_name
13         set the net name for constant 1 (default: Vdd)
14
15     -buf DC|subckt_name
16         set the name for jumper element (default: DC)
17         (used to connect different nets)
18
19     -nc_prefix
20         prefix for not-connected nets (default: _NC)
```

```
21
22      -inames
23          include names of internal ($-prefixed) nets in outputs
24          (default is to use net numbers instead)
25
26      -top top_module
27          set the specified module as design top module
```

C.241   write_table – write design as connectivity table

```
 1      write_table [options] [filename]
 2
 3  Write the current design as connectivity table. The output is a tab-separated
 4  ASCII table with the following columns:
 5
 6    module name
 7    cell name
 8    cell type
 9    cell port
10    direction
11    signal
12
13  module inputs and outputs are output using cell type and port '-' and with
14  'pi' (primary input) or 'po' (primary output) or 'pio' as direction.
```

C.242   write_verilog – write design to Verilog file

```
 1      write_verilog [options] [filename]
 2
 3  Write the current design to a Verilog file.
 4
 5      -sv
 6          with this option, SystemVerilog constructs like always_comb are used
 7
 8      -norename
 9          without this option all internal object names (the ones with a dollar
10          instead of a backslash prefix) are changed to short names in the
11          format '_<number>_'.
12
13      -renameprefix <prefix>
14          insert this prefix in front of auto-generated instance names
15
16      -noattr
17          with this option no attributes are included in the output
18
19      -attr2comment
20          with this option attributes are included as comments in the output
21
22      -noexpr
23          without this option all internal cells are converted to Verilog
```

```
24          expressions.
25
26      -siminit
27          add initial statements with hierarchical refs to initialize FFs when
28          in -noexpr mode.
29
30      -nodec
31          32-bit constant values are by default dumped as decimal numbers,
32          not bit pattern. This option deactivates this feature and instead
33          will write out all constants in binary.
34
35      -decimal
36          dump 32-bit constants in decimal and without size and radix
37
38      -nohex
39          constant values that are compatible with hex output are usually
40          dumped as hex values. This option deactivates this feature and
41          instead will write out all constants in binary.
42
43      -nostr
44          Parameters and attributes that are specified as strings in the
45          original input will be output as strings by this back-end. This
46          deactivates this feature and instead will write string constants
47          as binary numbers.
48
49      -simple-lhs
50          Connection assignments with simple left hand side without
51          concatenations.
52
53      -extmem
54          instead of initializing memories using assignments to individual
55          elements, use the '$readmemh' function to read initialization data
56          from a file. This data is written to a file named by appending
57          a sequential index to the Verilog filename and replacing the extension
58          with '.mem', e.g. 'write_verilog -extmem foo.v' writes 'foo-1.mem',
59          'foo-2.mem' and so on.
60
61      -defparam
62          use 'defparam' statements instead of the Verilog-2001 syntax for
63          cell parameters.
64
65      -blackboxes
66          usually modules with the 'blackbox' attribute are ignored. with
67          this option set only the modules with the 'blackbox' attribute
68          are written to the output file.
69
70      -selected
71          only write selected modules. modules must be selected entirely or
72          not at all.
73
74      -v
75          verbose output (print new names of all renamed wires and cells)
76
77  Note that RTLIL processes can't always be mapped directly to Verilog
```

```
78  always blocks. This frontend should only be used to export an RTLIL
79  netlist, i.e. after the "proc" pass has been used to convert all
80  processes to logic networks and registers. A warning is generated when
81  this command is called on a design with RTLIL processes.
```

C.243   write_xaiger – write design to XAIGER file

```
1      write_xaiger [options] [filename]
2
3  Write the top module (according to the (* top *) attribute or if only one module
4  is currently selected) to an XAIGER file. Any non $_NOT_, $_AND_, (optionally
5  $_DFF_N_, $_DFF_P_), or non (* abc9_box *) cells will be converted into psuedo-
6  inputs and pseudo-outputs. Whitebox contents will be taken from the equivalent
7  module in the '$abc9_holes' design, if it exists.
8
9      -ascii
10         write ASCII version of AIGER format
11
12     -map <filename>
13         write an extra file with port and box symbols
14
15     -dff
16         write $_DFF_[NP]_ cells
```

C.244   xilinx_dffopt – Xilinx: optimize FF control signal usage

```
1      xilinx_dffopt [options] [selection]
2
3  Converts hardware clock enable and set/reset signals on FFs to emulation
4  using LUTs, if doing so would improve area.  Operates on post-techmap Xilinx
5  cells (LUT*, FD*).
6
7      -lut4
8          Assume a LUT4-based device (instead of a LUT6-based device).
```

C.245   xilinx_dsp – Xilinx: pack resources into DSPs

```
1      xilinx_dsp [options] [selection]
2
3  Pack input registers (A2, A1, B2, B1, C, D, AD; with optional enable/reset),
4  pipeline registers (M; with optional enable/reset), output registers (P; with
5  optional enable/reset), pre-adder and/or post-adder into Xilinx DSP resources.
6
7  Multiply-accumulate operations using the post-adder with feedback on the 'C'
8  input will be folded into the DSP. In this scenario only, the 'C' input can be
9  used to override the current accumulation result with a new value, which will
10  be added to the multiplier result to form the next accumulation result.
```

```
11
12   Use of the dedicated 'PCOUT' -> 'PCIN' cascade path is detected for 'P' -> 'C'
13   connections (optionally, where 'P' is right-shifted by 17-bits and used as an
14   input to the post-adder -- a pattern common for summing partial products to
15   implement wide multipliers). Limited support also exists for similar cascading
16   for A and B using '[AB]COUT' -> '[AB]CIN'. Currently, cascade chains are limited
17   to a maximum length of 20 cells, corresponding to the smallest Xilinx 7 Series
18   device.
19
20   This pass is a no-op if the scratchpad variable 'xilinx_dsp.multonly' is set
21   to 1.
22
23
24   Experimental feature: addition/subtractions less than 12 or 24 bits with the
25   '(* use_dsp="simd" *)' attribute attached to the output wire or attached to
26   the add/subtract operator will cause those operations to be implemented using
27   the 'SIMD' feature of DSPs.
28
29   Experimental feature: the presence of a `$ge' cell attached to the registered
30   P output implementing the operation "(P >= <power-of-2>)" will be transformed
31   into using the DSP48E1's pattern detector feature for overflow detection.
32
33       -family {xcup|xcu|xc7|xc6v|xc5v|xc4v|xc6s|xc3sda}
34           select the family to target
35           default: xc7
```

C.246   xilinx_srl – Xilinx shift register extraction

```
1        xilinx_srl [options] [selection]
2
3    This pass converts chains of built-in flops (bit-level: $_DFF_[NP]_, $_DFFE_*
4    and word-level: $dff, $dffe) as well as Xilinx flops (FDRE, FDRE_1) into a
5    $__XILINX_SHREG cell. Chains must be of the same cell type, clock, clock
6    polarity, enable, and enable polarity (where relevant).
7    Flops with resets cannot be mapped to Xilinx devices and will not be inferred.
8
9        -minlen N
10           min length of shift register (default = 3)
11
12       -fixed
13           infer fixed-length shift registers.
14
15       -variable
16           infer variable-length shift registers (i.e. fixed-length shifts where
17           each element also fans-out to a $shiftx cell).
```

C.247   zinit – add inverters so all FF are zero-initialized

```
1        zinit [options] [selection]
2
```

```
3   Add inverters as needed to make all FFs zero-initialized.
4
5       -all
6           also add zero initialization to uninitialized FFs
```

Appendix D

RTLIL Text Representation

This appendix documents the text representation of RTLIL in extended Backus-Naur form (EBNF).

The grammar is not meant to represent semantic limitations. That is, the grammar is "permissive", and later stages of processing perform more rigorous checks.

The grammar is also not meant to represent the exact grammar used in the RTLIL frontend, since that grammar is specific to processing by lex and yacc, is even more permissive, and is somewhat less understandable than simple EBNF notation.

Finally, note that all statements (rules ending in `-stmt`) terminate in an end-of-line. Because of this, a statement cannot be broken into multiple lines.

D.1   Lexical elements

D.1.1   Characters

An RTLIL file is a stream of bytes. Strictly speaking, a "character" in an RTLIL file is a single byte. The lexer treats multi-byte encoded characters as consecutive single-byte characters. While other encodings *may* work, UTF-8 is known to be safe to use. Byte order marks at the beginning of the file will cause an error.

ASCII spaces (32) and tabs (9) separate lexer tokens.

A `nonws` character, used in identifiers, is any character whose encoding consists solely of bytes above ASCII space (32).

An `eol` is one or more consecutive ASCII newlines (10) and carriage returns (13).

D.1.2   Identifiers

There are two types of identifiers in RTLIL:

- Publically visible identifiers
- Auto-generated identifiers

$\langle id \rangle$ ::= $\langle public\text{-}id \rangle$ | $\langle autogen\text{-}id \rangle$

$\langle public\text{-}id \rangle$ ::= \ $\langle nonws \rangle$+

$\langle autogen\text{-}id \rangle$ ::= $ $\langle nonws \rangle$+

D.1.3 Values

A *value* consists of a width in bits and a bit representation, most significant bit first. Bits may be any of:

- `0`: A logic zero value

- `1`: A logic one value

- `x`: An unknown logic value (or don't care in case patterns)

- `z`: A high-impedance value (or don't care in case patterns)

- `m`: A marked bit (internal use only)

- `-`: A don't care value

An *integer* is simply a signed integer value in decimal format. **Warning:** Integer constants are limited to 32 bits. That is, they may only be in the range $[-2147483648, 2147483648)$. Integers outside this range will result in an error.

$\langle value \rangle$ ::= $\langle decimal\text{-}digit \rangle$+ ' $\langle binary\text{-}digit \rangle$*

$\langle decimal\text{-}digit \rangle$ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

$\langle binary\text{-}digit \rangle$ ::= `0` | `1` | `x` | `z` | `m` | `-`

$\langle integer \rangle$ ::= `-?` $\langle decimal\text{-}digit \rangle$+

D.1.4 Strings

A string is a series of characters delimited by double-quote characters. Within a string, any character except ASCII NUL (0) may be used. In addition, certain escapes can be used:

- `\n`: A newline

- `\t`: A tab

- `\ooo`: A character specified as a one, two, or three digit octal value

All other characters may be escaped by a backslash, and become the following character. Thus:

- `\\`: A backslash

- `\"`: A double-quote

- `\r`: An 'r' character

D.1.5 Comments

A comment starts with a `#` character and proceeds to the end of the line. All comments are ignored.

D.2 File

A file consists of an optional autoindex statement followed by zero or more modules.

⟨*file*⟩   ::= ⟨*autoidx-stmt*⟩? ⟨*module*⟩*

D.2.1 Autoindex statements

The autoindex statement sets the global autoindex value used by Yosys when it needs to generate a unique name, e.g. `$flatten$N`. The N part is filled with the value of the global autoindex value, which is subsequently incremented. This global has to be dumped into RTLIL, otherwise e.g. dumping and running a pass would have different properties than just running a pass on a warm design.

⟨*autoidx-stmt*⟩   ::= autoidx ⟨*integer*⟩ ⟨*eol*⟩

D.2.2 Modules

Declares a module, with zero or more attributes, consisting of zero or more wires, memories, cells, processes, and connections.

⟨*module*⟩	::=	⟨*attr-stmt*⟩∗ ⟨*module-stmt*⟩ ⟨*module-body*⟩ ⟨*module-end-stmt*⟩
⟨*module-stmt*⟩	::=	module ⟨*id*⟩ ⟨*eol*⟩
⟨*module-body*⟩	::=	(⟨*param-stmt*⟩
		| ⟨*wire*⟩
		| ⟨*memory*⟩
		| ⟨*cell*⟩
		| ⟨*process*⟩
		| ⟨*conn-stmt*⟩)∗
⟨*param-stmt*⟩	::=	parameter ⟨*id*⟩ ⟨*constant*⟩? ⟨*eol*⟩
⟨*constant*⟩	::=	⟨*value*⟩ | ⟨*integer*⟩ | ⟨*string*⟩
⟨*module-end-stmt*⟩	::=	end ⟨*eol*⟩

D.2.3 Attribute statements

Declares an attribute with the given identifier and value.

⟨*attr-stmt*⟩   ::= attribute ⟨*id*⟩ ⟨*constant*⟩ ⟨*eol*⟩

D.2.4 Signal specifications

A signal is anything that can be applied to a cell port, i.e. a constant value, all bits or a selection of bits from a wire, or concatenations of those.

Warning: When an integer constant is a sigspec, it is always 32 bits wide, 2's complement. For example, a constant of -1 is the same as 32'11111111111111111111111111111111, while a constant of 1 is the same as 32'1.

See Sec. 4.2.4 for an overview of signal specifications.

⟨*sigspec*⟩ ::= ⟨*constant*⟩
        | ⟨*wire-id*⟩
        | ⟨*sigspec*⟩ [⟨*integer*⟩ (: ⟨*integer*⟩)?]
        | { ⟨*sigspec*⟩* }

D.2.5 Connections

Declares a connection between the given signals.

⟨*conn-stmt*⟩ ::= connect ⟨*sigspec*⟩ ⟨*sigspec*⟩ ⟨*eol*⟩

D.2.6 Wires

Declares a wire, with zero or more attributes, with the given identifier and options in the enclosing module.

See Sec. 4.2.3 for an overview of wires.

⟨*wire*⟩   ::= ⟨*attr-stmt*⟩* ⟨*wire-stmt*⟩

⟨*wire-stmt*⟩   ::= wire ⟨*wire-option*⟩* ⟨*wire-id*⟩ ⟨*eol*⟩

⟨*wire-id*⟩   ::= ⟨*id*⟩

⟨*wire-option*⟩ ::= width ⟨*integer*⟩
            | offset ⟨*integer*⟩
            | input ⟨*integer*⟩
            | output ⟨*integer*⟩
            | inout ⟨*integer*⟩
            | upto
            | signed

D.2.7   Memories

Declares a memory, with zero or more attributes, with the given identifier and options in the enclosing module.

See Sec. 4.2.6 for an overview of memory cells, and Sec. 5.1.5 for details about memory cell types.

⟨*memory*⟩   ::= ⟨*attr-stmt*⟩∗ ⟨*memory-stmt*⟩

⟨*memory-stmt*⟩   ::= memory ⟨*memory-option*⟩∗ ⟨*id*⟩ ⟨*eol*⟩

⟨*memory-option*⟩ ::= width ⟨*integer*⟩
                   | size ⟨*integer*⟩
                   | offset ⟨*integer*⟩

D.2.8   Cells

Declares a cell, with zero or more attributes, with the given identifier and type in the enclosing module.

Cells perform functions on input signals. See Chap. 5 for a detailed list of cell types.

⟨*cell*⟩   ::= ⟨*attr-stmt*⟩∗ ⟨*cell-stmt*⟩ ⟨*cell-body-stmt*⟩∗ ⟨*cell-end-stmt*⟩

⟨*cell-stmt*⟩   ::= cell ⟨*cell-type*⟩ ⟨*cell-id*⟩ ⟨*eol*⟩

⟨*cell-id*⟩   ::= ⟨*id*⟩

⟨*cell-type*⟩   ::= ⟨*id*⟩

⟨*cell-body-stmt*⟩ ::= parameter (signed | real)? ⟨*id*⟩ ⟨*constant*⟩ ⟨*eol*⟩
                   | connect ⟨*id*⟩ ⟨*sigspec*⟩ ⟨*eol*⟩

⟨*cell-end-stmt*⟩ ::= end ⟨*eol*⟩

D.2.9   Processes

Declares a process, with zero or more attributes, with the given identifier in the enclosing module. The body of a process consists of zero or more assignments, exactly one switch, and zero or more syncs.

See Sec. 4.2.5 for an overview of processes.

⟨*process*⟩   ::= ⟨*attr-stmt*⟩∗ ⟨*proc-stmt*⟩ ⟨*process-body*⟩ ⟨*proc-end-stmt*⟩

⟨*proc-stmt*⟩   ::= process ⟨*id*⟩ ⟨*eol*⟩

⟨*process-body*⟩   ::= ⟨*assign-stmt*⟩∗ ⟨*switch*⟩? ⟨*assign-stmt*⟩∗ ⟨*sync*⟩∗

⟨*assign-stmt*⟩   ::= assign ⟨*dest-sigspec*⟩ ⟨*src-sigspec*⟩ ⟨*eol*⟩

⟨*dest-sigspec*⟩   ::= ⟨*sigspec*⟩

⟨*src-sigspec*⟩   ::= ⟨*sigspec*⟩

⟨*proc-end-stmt*⟩ ::= end ⟨*eol*⟩

D.2.10   Switches

Switches test a signal for equality against a list of cases. Each case specifies a comma-separated list of signals to check against. If there are no signals in the list, then the case is the default case. The body of a case consists of zero or more switches and assignments. Both switches and cases may have zero or more attributes.

⟨*switch*⟩   ::= ⟨*switch-stmt*⟩ ⟨*case*⟩∗ ⟨*switch-end-stmt*⟩

⟨*switch-stmt*⟩   := ⟨*attr-stmt*⟩∗ `switch` ⟨*sigspec*⟩ ⟨*eol*⟩

⟨*case*⟩   ::= ⟨*attr-stmt*⟩∗ ⟨*case-stmt*⟩ ⟨*case-body*⟩

⟨*case-stmt*⟩   ::= `case` ⟨*compare*⟩? ⟨*eol*⟩

⟨*compare*⟩   ::= ⟨*sigspec*⟩ (, ⟨*sigspec*⟩)∗

⟨*case-body*⟩   ::= (⟨*switch*⟩ | ⟨*assign-stmt*⟩)∗

⟨*switch-end-stmt*⟩ ::= `end` ⟨*eol*⟩

D.2.11   Syncs

Syncs update signals with other signals when an event happens. Such an event may be:

- An edge or level on a signal
- Global clock ticks
- Initialization
- Always

⟨*sync*⟩   ::= ⟨*sync-stmt*⟩ ⟨*update-stmt*⟩∗

⟨*sync-stmt*⟩   ::= `sync` ⟨*sync-type*⟩ ⟨*sigspec*⟩ ⟨*eol*⟩
                 | `sync global` ⟨*eol*⟩
                 | `sync init` ⟨*eol*⟩
                 | `sync always` ⟨*eol*⟩

⟨*sync-type*⟩   ::= `low` | `high` | `posedge` | `negedge` | `edge`

⟨*update-stmt*⟩   ::= `update` ⟨*dest-sigspec*⟩ ⟨*src-sigspec*⟩ ⟨*eol*⟩

Appendix E

Application Notes

This appendix contains copies of the Yosys application notes.

Yosys Application Note 010:
Converting Verilog to BLIF

Claire Xenia Wolf
November 2013

Abstract—**Verilog-2005 is a powerful Hardware Description Language (HDL) that can be used to easily create complex designs from small HDL code. It is the preferred method of design entry for many designers[1].**

The Berkeley Logic Interchange Format (BLIF) [6] is a simple file format for exchanging sequential logic between programs. It is easy to generate and easy to parse and is therefore the preferred method of design entry for many authors of logic synthesis tools.

Yosys [1] is a feature-rich Open-Source Verilog synthesis tool that can be used to bridge the gap between the two file formats. It implements most of Verilog-2005 and thus can be used to import modern behavioral Verilog designs into BLIF-based design flows without dependencies on proprietary synthesis tools.

The scope of Yosys goes of course far beyond Verilog logic synthesis. But it is a useful and important feature and this Application Note will focus on this aspect of Yosys.

I. INSTALLATION

Yosys written in C++ (using features from C++11) and is tested on modern Linux. It should compile fine on most UNIX systems with a C++11 compiler. The README file contains useful information on building Yosys and its prerequisites.

Yosys is a large and feature-rich program with a couple of dependencies. It is, however, possible to deactivate some of the dependencies in the Makefile, resulting in features in Yosys becoming unavailable. When problems with building Yosys are encountered, a user who is only interested in the features of Yosys that are discussed in this Application Note may deactivate `TCL`, `Qt` and `MiniSAT` support in the `Makefile` and may opt against building `yosys-abc`.

This Application Note is based on GIT Rev. `e216e0e` from 2013-11-23 of Yosys [1]. The Verilog sources used for the examples are taken from yosys-bigsim [2], a collection of real-world designs used for regression testing Yosys.

II. GETTING STARTED

We start our tour with the Navré processor from yosys-bigsim. The Navré processor [3] is an Open Source AVR clone. It is a single module (`softusb_navre`) in a single design file (`softusb_navre.v`). It also is using only features that map nicely to the BLIF format, for example it only uses synchronous resets.

Converting `softusb_navre.v` to `softusb_navre.blif` could not be easier:

```
1  yosys -o softusb_navre.blif -S softusb_navre.v
```

Listing 1. Calling Yosys without script file

Behind the scenes Yosys is controlled by synthesis scripts that execute commands that operate on Yosys' internal state. For example, the `-o softusb_navre.blif` option just adds the command `write_blif softusb_navre.blif` to the end of the script. Likewise a file on the command line – `softusb_navre.v` in this case – adds the command `read_verilog softusb_navre.v` to the beginning of the synthesis script. In both cases the file type is detected from the file extension.

[1]The other half prefers VHDL, a very different but – of course – equally powerful language.

Finally the option `-s` instantiates a built-in default synthesis script. Instead of using `-s` one could also specify the synthesis commands for the script on the command line using the `-p` option, either using individual options for each command or by passing one big command string with a semicolon-separated list of commands. But in most cases it is more convenient to use an actual script file.

III. USING A SYNTHESIS SCRIPT

With a script file we have better control over Yosys. The following script file replicates what the command from the last section did:

```
1  read_verilog softusb_navre.v
2  hierarchy
3  proc; opt; memory; opt; techmap; opt
4  write_blif softusb_navre.blif
```

Listing 2. `softusb_navre.ys`

The first and last line obviously read the Verilog file and write the BLIF file.

The 2nd line checks the design hierarchy and instantiates parametrized versions of the modules in the design, if necessary. In the case of this simple design this is a no-op. However, as a general rule a synthesis script should always contain this command as first command after reading the input files.

The 3rd line does most of the actual work:

- The command `opt` is the Yosys' built-in optimizer. It can perform some simple optimizations such as const-folding and removing unconnected parts of the design. It is common practice to call opt after each major step in the synthesis procedure. In cases where too much optimization is not appreciated (for example when analyzing a design), it is recommended to call `clean` instead of `opt`.
- The command `proc` converts *processes* (Yosys' internal representation of Verilog `always`- and `initial`-blocks) to circuits of multiplexers and storage elements (various types of flip-flops).
- The command `memory` converts Yosys' internal representations of arrays and array accesses to multi-port block memories, and then maps this block memories to address decoders and flip-flops, unless the option `-nomap` is used, in which case the multi-port block memories stay in the design and can then be mapped to architecture-specific memory primitives using other commands.
- The command `techmap` turns a high-level circuit with coarse grain cells such as wide adders and multipliers to a fine-grain circuit of simple logic primitives and single-bit storage elements. The command does that by substituting the complex cells by circuits of simpler cells. It is possible to provide a custom set of rules for this process in the form of a Verilog source file, as we will see in the next section.

Now Yosys can be run with the filename of the synthesis script as argument:

```
1  yosys softusb_navre.ys
```

Listing 3. Calling Yosys with script file

Now that we are using a synthesis script we can easily modify how Yosys synthesizes the design. The first thing we should customize is the call to the `hierarchy` command:

Whenever it is known that there are no implicit blackboxes in the design, i.e. modules that are referenced but are not defined, the `hierarchy` command should be called with the `-check` option. This will then cause synthesis to fail when implicit blackboxes are found in the design.

The 2nd thing we can improve regarding the `hierarchy` command is that we can tell it the name of the top level module of the design hierarchy. It will then automatically remove all modules that are not referenced from this top level module.

For many designs it is also desired to optimize the encodings for the finite state machines (FSMs) in the design. The `fsm` command finds FSMs, extracts them, performs some basic optimizations and then generate a circuit from the extracted and optimized description. It would also be possible to tell the `fsm` command to leave the FSMs in their extracted form, so they can be further processed using custom commands. But in this case we don't want that.

So now we have the final synthesis script for generating a BLIF file for the Navré CPU:

```
1  read_verilog softusb_navre.v
2  hierarchy -check -top softusb_navre
3  proc; opt; memory; opt; fsm; opt; techmap; opt
4  write_blif softusb_navre.blif
```

Listing 4. `softusb_navre.ys` (improved)

IV. ADVANCED EXAMPLE: THE AMBER23 ARMV2A CPU

Our 2nd example is the Amber23 [4] ARMv2a CPU. Once again we base our example on the Verilog code that is included in yosys-bigsim [2].

The problem with this core is that it contains no dedicated reset logic. Instead the coding techniques shown in Listing 6 are used to define reset values for the global asynchronous reset in an FPGA implementation. This design can not be expressed in BLIF as it is. Instead we need to use a synthesis script that transforms this form to synchronous resets that can be expressed in BLIF.

```
1  read_verilog a23_alu.v
2  read_verilog a23_barrel_shift_fpga.v
3  read_verilog a23_barrel_shift.v
4  read_verilog a23_cache.v
5  read_verilog a23_coprocessor.v
6  read_verilog a23_core.v
7  read_verilog a23_decode.v
8  read_verilog a23_execute.v
9  read_verilog a23_fetch.v
10 read_verilog a23_multiply.v
11 read_verilog a23_ram_register_bank.v
12 read_verilog a23_register_bank.v
13 read_verilog a23_wishbone.v
14 read_verilog generic_sram_byte_en.v
15 read_verilog generic_sram_line_en.v
16 hierarchy -check -top a23_core
17 add -global_input globrst 1
18 proc -global_arst globrst
19 techmap -map adff2dff.v
20 opt; memory; opt; fsm; opt; techmap
21 write_blif amber23.blif
```

Listing 5. `amber23.ys`

```
1  reg [7:0] a = 13, b;
2  initial b = 37;
```

Listing 6. Implicit coding of global asynchronous resets

(Note that there is no problem if this coding techniques are used to model ROM, where the register is initialized using this syntax but is never updated otherwise.)

Listing 5 shows the synthesis script for the Amber23 core. In line 17 the `add` command is used to add a 1-bit wide global input signal with the name `globrst`. That means that an input with that name is added to each module in the design hierarchy and then all module instantiations are altered so that this new signal is connected throughout the whole design hierarchy.

In line 18 the `proc` command is called. But in this script the signal name `globrst` is passed to the command as a global reset signal for resetting the registers to their assigned initial values.

Finally in line 19 the `techmap` command is used to replace all instances of flip-flops with asynchronous resets with flip-flops with synchronous resets. The map file used for this is shown in Listing 7. Note how the `techmap_celltype` attribute is used in line 1 to tell the techmap command which cells to replace in the design, how the `_TECHMAP_FAIL_` wire in lines 15 and 16 (which evaluates to a constant value) determines if the parameter set is compatible with this replacement circuit, and how the `_TECHMAP_DO_` wire in line 13 provides a mini synthesis-script to be used to process this cell.

V. VERIFICATION OF THE AMBER23 CPU

The BLIF file for the Amber23 core, generated using Listings 5 and 7 and the version of the Amber23 RTL source that is bundled with yosys-bigsim, was verified using the test-bench from yosys-bigsim. It successfully executed the program shown in Listing 8 in the test-bench.

```
1  (* techmap_celltype = "$adff" *)
2  module adff2dff (CLK, ARST, D, Q);
3
4  parameter WIDTH = 1;
5  parameter CLK_POLARITY = 1;
6  parameter ARST_POLARITY = 1;
7  parameter ARST_VALUE = 0;
8
9  input CLK, ARST;
10 input [WIDTH-1:0] D;
11 output reg [WIDTH-1:0] Q;
12
13 wire [1023:0] _TECHMAP_DO_ = "proc";
14
15 wire _TECHMAP_FAIL_ =
16     !CLK_POLARITY || !ARST_POLARITY;
17
18 always @(posedge CLK)
19     if (ARST)
20         Q <= ARST_VALUE;
21     else
22         Q <= D;
23
24 endmodule
```

Listing 7. `adff2dff.v`

```
1   #include <stdint.h>
2   #include <stdbool.h>
3
4   #define BITMAP_SIZE 64
5   #define OUTPORT 0x10000000
6
7   static uint32_t bitmap[BITMAP_SIZE/32];
8
9   static void bitmap_set(uint32_t idx) { bitmap[idx/32] |= 1 << (idx % 32); }
10  static bool bitmap_get(uint32_t idx) { return (bitmap[idx/32] & (1 << (idx % 32))) != 0; }
11  static void output(uint32_t val) { *((volatile uint32_t*)OUTPORT) = val; }
12
13  int main() {
14          uint32_t i, j, k;
15          output(2);
16          for (i = 0; i < BITMAP_SIZE; i++) {
17                  if (bitmap_get(i)) continue;
18                  output(3+2*i);
19                  for (j = 2*(3+2*i);; j += 3+2*i) {
20                          if (j%2 == 0) continue;
21                          k = (j-3)/2;
22                          if (k >= BITMAP_SIZE) break;
23                          bitmap_set(k);
24                  }
25          }
26          output(0);
27          return 0;
28  }
```

Listing 8. Test program for the Amber23 CPU (Sieve of Eratosthenes). Compiled using GCC 4.6.3 for ARM with `-Os -marm -march=armv2a -mno-thumb-interwork -ffreestanding`, linked with `--fix-v4bx` set and booted with a custom setup routine written in ARM assembler.

For simulation the BLIF file was converted back to Verilog using ABC [5]. So this test includes the successful transformation of the BLIF file into ABC's internal format as well.

The only thing left to write about the simulation itself is that it probably was one of the most energy inefficient and time consuming ways of successfully calculating the first 31 primes the author has ever conducted.

VI. LIMITATIONS

At the time of this writing Yosys does not support multi-dimensional memories, does not support writing to individual bits of array elements, does not support initialization of arrays with `$readmemb` and `$readmemh`, and has only limited support for tristate logic, to name just a few limitations.

That being said, Yosys can synthesize an overwhelming majority of real-world Verilog RTL code. The remaining cases can usually be modified to be compatible with Yosys quite easily.

The various designs in yosys-bigsim are a good place to look for examples of what is within the capabilities of Yosys.

VII. CONCLUSION

Yosys is a feature-rich Verilog-2005 synthesis tool. It has many uses, but one is to provide an easy gateway from high-level Verilog code to low-level logic circuits.

The command line option `-s` can be used to quickly synthesize Verilog code to BLIF files without a hassle.

With custom synthesis scripts it becomes possible to easily perform high-level optimizations, such as re-encoding FSMs. In some extreme cases, such as the Amber23 ARMv2 CPU, the more advanced Yosys features can be used to change a design to fit a certain need without actually touching the RTL code.

REFERENCES

[1] Claire Xenia Wolf. The Yosys Open SYnthesis Suite.
    https://yosyshq.net/yosys/
[2] yosys-bigsim, a collection of real-world Verilog designs for regression
    testing purposes.
    https://github.com/YosysHQ/yosys-bigsim
[3] Sebastien Bourdeauducq. Navré AVR clone (8-bit RISC).
    http://opencores.org/project,navre
[4] Conor Santifort. Amber ARM-compatible core.
    http://opencores.org/project,amber
[5] Berkeley Logic Synthesis and Verification Group. ABC: A System for
    Sequential Synthesis and Verification.
    http://www.eecs.berkeley.edu/~alanmi/abc/
[6] Berkeley Logic Interchange Format (BLIF)
    http://vlsi.colorado.edu/~vis/blif.ps

Yosys Application Note 011: Interactive Design Investigation

Claire Xenia Wolf

Original Version December 2013

Abstract—**Yosys [1] can be a great environment for building custom synthesis flows. It can also be an excellent tool for teaching and learning Verilog based RTL synthesis. In both applications it is of great importance to be able to analyze the designs it produces easily.**

This Yosys application note covers the generation of circuit diagrams with the Yosys show command, the selection of interesting parts of the circuit using the select command, and briefly discusses advanced investigation commands for evaluating circuits and solving SAT problems.

I. INSTALLATION AND PREREQUISITES

This Application Note is based on the Yosys [1] GIT Rev. 2b90ba1 from 2013-12-08. The README file covers how to install Yosys. The show command requires a working installation of GraphViz [2] and [3] for generating the actual circuit diagrams.

II. OVERVIEW

This application note is structured as follows:

Sec. III introduces the show command and explains the symbols used in the circuit diagrams generated by it.

Sec. IV introduces additional commands used to navigate in the design, select portions of the design, and print additional information on the elements in the design that are not contained in the circuit diagrams.

Sec. V introduces commands to evaluate the design and solve SAT problems within the design.

Sec. VI concludes the document and summarizes the key points.

III. INTRODUCTION TO THE show COMMAND

The show command generates a circuit diagram for the design in its current state. Various options can be used to change the appearance of the circuit diagram, set the name and format for the output file, and so forth. When called without any special options, it saves the circuit diagram in a temporary file and launches xdot to display the diagram. Subsequent calls to show re-use the xdot instance (if still running).

```
1   $ cat example.ys
2   read_verilog example.v
3   show -pause
4   proc
5   show -pause
6   opt
7   show -pause
8
9   $ cat example.v
10  module example(input clk, a, b, c,
11                 output reg [1:0] y);
12      always @(posedge clk)
13          if (c)
14              y <= c ? a + b : 2'd0;
15  endmodule
```

Figure 1. Yosys script with show commands and example design

A. A simple circuit

Fig. 1 shows a simple synthesis script and a Verilog file that demonstrate the usage of show in a simple setting. Note that show is called with the -pause option, that halts execution of the Yosys script until the user presses the Enter key. The show -pause command also allows the user to enter an interactive shell to further investigate the circuit before continuing synthesis.

So this script, when executed, will show the design after each of the three synthesis commands. The generated circuit diagrams are shown in Fig. 2.

The first diagram (from top to bottom) shows the design directly after being read by the Verilog front-end. Input and output ports are displayed as octagonal shapes. Cells are displayed as rectangles with inputs on the left and outputs on the right side. The cell labels are two lines long: The first line contains a unique identifier for the cell and the second line contains the cell type. Internal cell types are prefixed with a dollar sign. The Yosys manual contains a chapter on the internal cell library used in Yosys.

Constants are shown as ellipses with the constant value as label. The syntax `<bit_width>'<bits>` is used for for constants that are not 32-bit wide and/or contain bits that are not 0 or 1 (i.e. x or z). Ordinary 32-bit constants are written using decimal numbers.

Single-bit signals are shown as thin arrows pointing from the driver to the load. Signals that are multiple bits wide are shown as think arrows.

Finally *processes* are shown in boxes with round corners. Processes are Yosys' internal representation of the decision-trees and synchronization events modelled in a Verilog always-block. The label reads PROC followed by a unique identifier in the first line and contains the source code location of the original always-block in the 2nd line. Note how the multiplexer from the ?:-expression is represented as a $mux cell but the multiplexer from the if-statement is yet still hidden within the process.
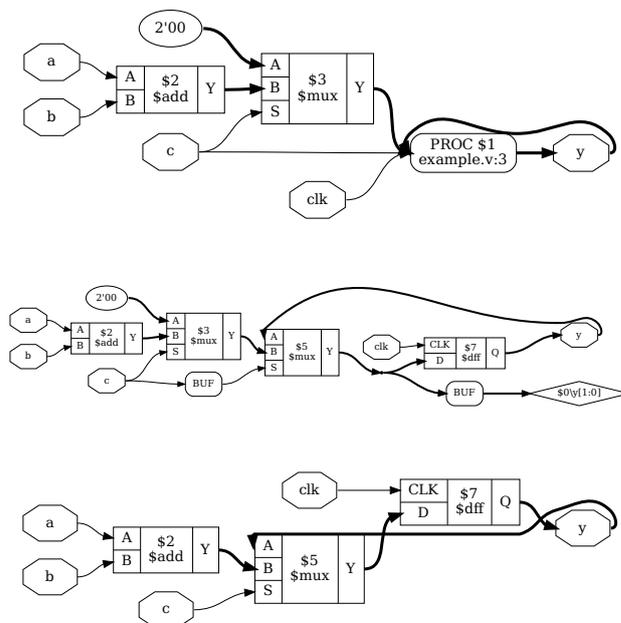

Figure 2. Output of the three show commands from Fig. 1

The `proc` command transforms the process from the first diagram into a multiplexer and a d-type flip-flip, which brings us to the 2nd diagram.

The Rhombus shape to the right is a dangling wire. (Wire nodes are only shown if they are dangling or have "public" names, for example names assigned from the Verilog input.) Also note that the design now contains two instances of a `BUF`-node. This are artefacts left behind by the `proc`-command. It is quite usual to see such artefacts after calling commands that perform changes in the design, as most commands only care about doing the transformation in the least complicated way, not about cleaning up after them. The next call to `clean` (or `opt`, which includes `clean` as one of its operations) will clean up this artefacts. This operation is so common in Yosys scripts that it can simply be abbreviated with the `;;` token, which doubles as separator for commands. Unless one wants to specifically analyze this artefacts left behind some operations, it is therefore recommended to always call `clean` before calling `show`.

In this script we directly call `opt` as next step, which finally leads us to the 3rd diagram in Fig. 2. Here we see that the `opt` command not only has removed the artifacts left behind by `proc`, but also determined correctly that it can remove the first `$mux` cell without changing the behavior of the circuit.

B. Break-out boxes for signal vectors

As has been indicated by the last example, Yosys is can manage signal vectors (aka. multi-bit wires or buses) as native objects. This provides great advantages when analyzing circuits that operate on wide integers. But it also introduces some additional complexity when the individual bits of of a signal vector are accessed. The example show in Fig. 3 and 4 demonstrates how such circuits are visualized by the `show` command.
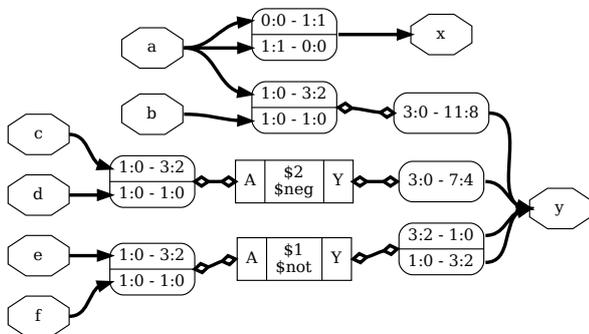
Figure 3. Output of `yosys -p 'proc; opt; show'` splice.v

```
 1   module splice_demo(a, b, c, d, e, f, x, y);
 2
 3   input [1:0] a, b, c, d, e, f;
 4   output [1:0] x = {a[0], a[1]};
 5
 6   output [11:0] y;
 7   assign {y[11:4], y[1:0], y[3:2]} =
 8                   {a, b, -{c, d}, ~{e, f}};
 9
10   endmodule
```

Figure 4. splice.v
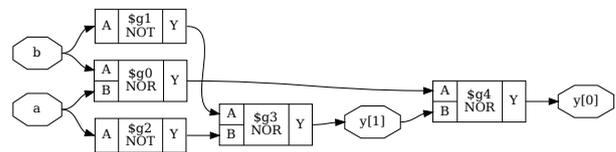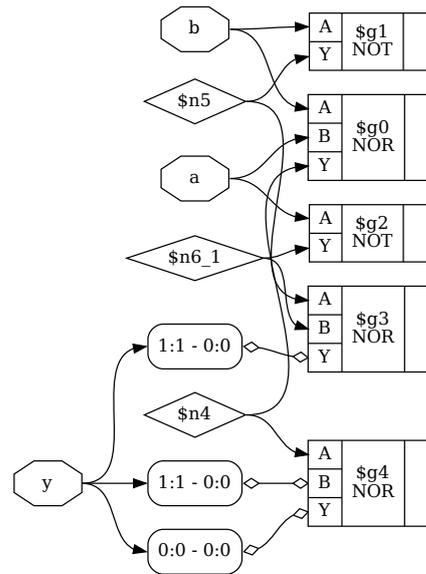
Figure 5. Effects of `splitnets` command and of providing a cell library. (The circuit is a half-adder built from simple CMOS gates.)

The key elements in understanding this circuit diagram are of course the boxes with round corners and rows labeled `<MSB_LEFT>:<LSB_LEFT> - <MSB_RIGHT>:<LSB_RIGHT>`. Each of this boxes has one signal per row on one side and a common signal for all rows on the other side. The `<MSB>:<LSB>` tuples specify which bits of the signals are broken out and connected. So the top row of the box connecting the signals `a` and `x` indicates that the bit 0 (i.e. the range 0:0) from signal `a` is connected to bit 1 (i.e. the range 1:1) of signal `x`.

Lines connecting such boxes together and lines connecting such boxes to cell ports have a slightly different look to emphasise that they are not actual signal wires but a necessity of the graphical representation. This distinction seems like a technicality, until one wants to debug a problem related to the way Yosys internally represents signal vectors, for example when writing custom Yosys commands.

C. Gate level netlists

Finally Fig. 5 shows two common pitfalls when working with designs mapped to a cell library. The top figure has two problems: First Yosys did not have access to the cell library when this diagram was generated, resulting in all cell ports defaulting to being inputs. This is why all ports are drawn on the left side the cells are awkwardly arranged in a large column. Secondly the two-bit vector `y` requires breakout-boxes for its individual bits, resulting in an unnecessary complex diagram.

273

For the 2nd diagram Yosys has been given a description of the cell library as Verilog file containing blackbox modules. There are two ways to load cell descriptions into Yosys: First the Verilog file for the cell library can be passed directly to the `show` command using the `-lib <filename>` option. Secondly it is possible to load cell libraries into the design with the `read_verilog -lib <filename>` command. The 2nd method has the great advantage that the library only needs to be loaded once and can then be used in all subsequent calls to the `show` command.

In addition to that, the 2nd diagram was generated after `splitnet -ports` was run on the design. This command splits all signal vectors into individual signal bits, which is often desirable when looking at gate-level circuits. The `-ports` option is required to also split module ports. Per default the command only operates on interior signals.

D. Miscellaneous notes

Per default the `show` command outputs a temporary `dot` file and launches `xdot` to display it. The options `-format`, `-viewer` and `-prefix` can be used to change format, viewer and filename prefix. Note that the `pdf` and `ps` format are the only formats that support plotting multiple modules in one run.

In densely connected circuits it is sometimes hard to keep track of the individual signal wires. For this cases it can be useful to call `show` with the `-colors <integer>` argument, which randomly assigns colors to the nets. The integer (> 0) is used as seed value for the random color assignments. Sometimes it is necessary it try some values to find an assignment of colors that looks good.

The command `help show` prints a complete listing of all options supported by the `show` command.

IV. NAVIGATING THE DESIGN

Plotting circuit diagrams for entire modules in the design brings us only helps in simple cases. For complex modules the generated circuit diagrams are just stupidly big and are no help at all. In such cases one first has to select the relevant portions of the circuit.

In addition to *what* to display one also needs to carefully decide *when* to display it, with respect to the synthesis flow. In general it is a good idea to troubleshoot a circuit in the earliest state in which a problem can be reproduced. So if, for example, the internal state before calling the `techmap` command already fails to verify, it is better to troubleshoot the coarse-grain version of the circuit before `techmap` than the gate-level circuit after `techmap`.

Note: It is generally recommended to verify the internal state of a design by writing it to a Verilog file using `write_verilog -noexpr` and using the simulation models from `simlib.v` and `simcells.v` from the Yosys data directory (as printed by `yosys-config --datdir`).

A. Interactive Navigation

Once the right state within the synthesis flow for debugging the circuit has been identified, it is recommended to simply add the `shell` command to the matching place in the synthesis script. This command will stop the synthesis at the specified moment and go to shell mode, where the user can interactively enter commands.

For most cases, the shell will start with the whole design selected (i.e. when the synthesis script does not already narrow the selection). The command `ls` can now be used to create a list of all modules. The command `cd` can be used to switch to one of the modules (type `cd ..` to switch back). Now the `ls` command lists the objects within that module. Fig. 6 demonstrates this using the design from Fig. 1.

There is a thing to note in Fig. 6: We can see that the cell names from Fig. 2 are just abbreviations of the actual cell names, namely

```
1   yosys> ls
2
3   1 modules:
4     example
5
6   yosys> cd example
7
8   yosys [example]> ls
9
10  7 wires:
11    $0\y[1:0]
12    $add$example.v:5$2_Y
13    a
14    b
15    c
16    clk
17    y
18
19  3 cells:
20    $add$example.v:5$2
21    $procdff$7
22    $procmux$5
```

Figure 6. Demonstration of `ls` and `cd` using `example.v` from Fig. 1

the part after the last dollar-sign. Most auto-generated names (the ones starting with a dollar sign) are rather long and contains some additional information on the origin of the named object. But in most cases those names can simply be abbreviated using the last part.

Usually all interactive work is done with one module selected using the `cd` command. But it is also possible to work from the design-context (`cd ..`). In this case all object names must be prefixed with `<module_name>/`. For example `a*/b*` would refer to all objects whose names start with `b` from all modules whose names start with `a`.

The `dump` command can be used to print all information about an object. For example `dump $2` will print Fig. 7. This can for example be useful to determine the names of nets connected to cells, as the net-names are usually suppressed in the circuit diagram if they are auto-generated.

For the remainder of this document we will assume that the commands are run from module-context and not design-context.

B. Working with selections

When a module is selected using the `cd` command, all commands (with a few exceptions, such as the `read_*` and `write_*` commands) operate only on the selected module. This can also be useful for

```
1    attribute \src "example.v:5"
2    cell $add $add$example.v:5$2
3      parameter \A_SIGNED 0
4      parameter \A_WIDTH 1
5      parameter \B_SIGNED 0
6      parameter \B_WIDTH 1
7      parameter \Y_WIDTH 2
8      connect \A \a
9      connect \B \b
10     connect \Y $add$example.v:5$2_Y
11   end
```

Figure 7. Output of `dump $2` using the design from Fig. 1 and Fig. 2
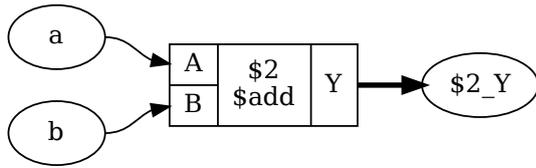
Figure 8. Output of `show` after `select $2` or `select t:$add` (see also Fig. 2)

```
1   module sumprod(a, b, c, sum, prod);
2
3      input [7:0] a, b, c;
4      output [7:0] sum, prod;
5
6      {* sumstuff *}
7      assign sum = a + b + c;
8      {* *}
9
10     assign prod = a * b * c;
11
12  endmodule
```

Figure 10. Another test module for operations on selections

synthesis scripts where different synthesis strategies should be applied to different modules in the design.

But for most interactive work we want to further narrow the set of selected objects. This can be done using the `select` command.

For example, if the command `select $2` is executed, a subsequent `show` command will yield the diagram shown in Fig. 8. Note that the nets are now displayed in ellipses. This indicates that they are not selected, but only shown because the diagram contains a cell that is connected to the net. This of course makes no difference for the circuit that is shown, but it can be a useful information when manipulating selections.

Objects can not only be selected by their name but also by other properties. For example `select t:$add` will select all cells of type `$add`. In this case this is also yields the diagram shown in Fig. 8.

The output of `help select` contains a complete syntax reference for matching different properties.

Many commands can operate on explicit selections. For example the command `dump t:$add` will print information on all `$add` cells in the active module. Whenever a command has `[selection]` as last argument in its usage help, this means that it will use the engine behind the `select` command to evaluate additional arguments and use the resulting selection instead of the selection created by the last `select` command.

Normally the `select` command overwrites a previous selection. The commands `select -add` and `select -del` can be used to add or remove objects from the current selection.

The command `select -clear` can be used to reset the selection to the default, which is a complete selection of everything in the current module.

C. Operations on selections

The `select` command is actually much more powerful than it might seem on the first glimpse. When it is called with multiple arguments, each argument is evaluated and pushed separately on a stack. After all arguments have been processed it simply creates the union of all elements on the stack. So the following command will select all `$add` cells and all objects with the `foo` attribute set:

```
select t:$add a:foo
```

(Try this with the design shown in Fig. 9. Use the `select -list` command to list the current selection.)

In many cases simply adding more and more stuff to the selection is an ineffective way of selecting the interesting part of the design. Special arguments can be used to combine the elements on the stack. For example the `%i` arguments pops the last two elements from the stack, intersects them, and pushes the result back on the stack. So the following command will select all `$add` cells that have the `foo` attribute set:

```
select t:$add a:foo %i
```

The listing in Fig. 10 uses the Yosys non-standard `{* ... *}` syntax to set the attribute `sumstuff` on all cells generated by the first assign statement. (This works on arbitrary large blocks of Verilog code an can be used to mark portions of code for analysis.)

Selecting `a:sumstuff` in this module will yield the circuit diagram shown in Fig. 11. As only the cells themselves are selected, but not the temporary wire `$1_Y`, the two adders are shown as two disjunct parts. This can be very useful for global signals like clock and reset signals: just unselect them using a command such as `select -del clk rst` and each cell using them will get its own net label.

```
1   module foobaraddsub(a, b, c, d, fa, fs, ba, bs);
2      input [7:0] a, b, c, d;
3      output [7:0] fa, fs, ba, bs;
4      assign fa = a + (* foo *) b;
5      assign fs = a - (* foo *) b;
6      assign ba = c + (* bar *) d;
7      assign bs = c - (* bar *) d;
8   endmodule
```

Figure 9. Test module for operations on selections
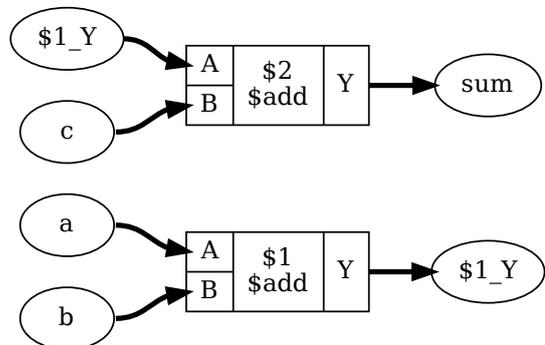
Figure 11. Output of `show a:sumstuff` on Fig. 10
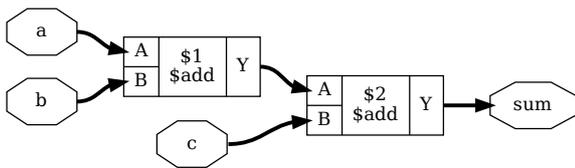
Figure 12. Output of `show a:sumstuff %x` on Fig. 10

In this case however we would like to see the cells connected properly. This can be achieved using the `%x` action, that broadens the selection, i.e. for each selected wire it selects all cells connected to the wire and vice versa. So `show a:sumstuff %x` yields the diagram shown in Fig. 12.

D. Selecting logic cones

Fig. 12 shows what is called the *input cone* of `sum`, i.e. all cells and signals that are used to generate the signal `sum`. The `%ci` action can be used to select the input cones of all object in the top selection in the stack maintained by the `select` command.

As the `%x` action, this commands broadens the selection by one "step". But this time the operation only works against the direction of data flow. That means, wires only select cells via output ports and cells only select wires via input ports.

Fig. 13 show the sequence of diagrams generated by the following commands:

```
show prod
show prod %ci
show prod %ci %ci
show prod %ci %ci %ci
```

When selecting many levels of logic, repeating `%ci` over and over again can be a bit dull. So there is a shortcut for that: the number of iterations can be appended to the action. So for example the action `%ci3` is identical to performing the `%ci` action three times.

The action `%ci*` performs the `%ci` action over and over again until it has no effect anymore.

In most cases there are certain cell types and/or ports that should not be considered for the `%ci` action, or we only want to follow certain cell types and/or ports. This can be achieved using additional patterns that can be appended to the `%ci` action.

Lets consider the design from Fig. 14. It serves no purpose other than being a non-trivial circuit for demonstrating some of the advanced Yosys features. We synthesize the circuit using `proc; opt; memory; opt` and change to the `memdemo` module with `cd memdemo`. If we type `show` now we see the diagram shown in Fig. 15.

But maybe we are only interested in the tree of multiplexers that select the output value. In order to get there, we would start by just showing the output signal and its immediate predecessors:

```
show y %ci2
```

From this we would learn that `y` is driven by a `$dff` cell, that `y` is connected to the output port `Q`, that the `clk` signal goes into the `CLK` input port of the cell, and that the data comes from a auto-generated wire into the input `D` of the flip-flop cell.

As we are not interested in the clock signal we add an additional pattern to the `%ci` action, that tells it to only follow ports `Q` and `D` of `$dff` cells:
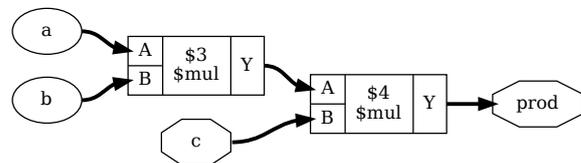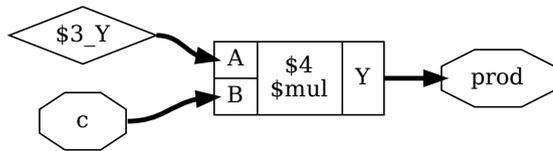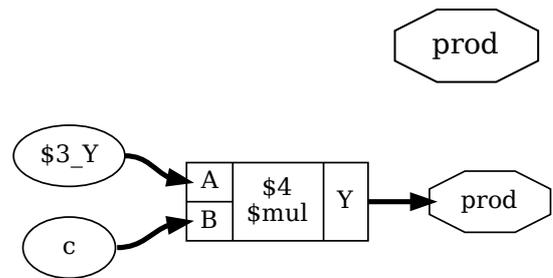
Figure 13. Objects selected by `select prod %ci...`

```
show y %ci2:+$dff[Q,D]
```

To add a pattern we add a colon followed by the pattern to the `%ci` action. The pattern it self starts with - or +, indicating if it is an include or exclude pattern, followed by an optional comma separated list of cell types, followed by an optional comma separated list of port names in square brackets.

Since we know that the only cell considered in this case is a `$dff` cell, we could as well only specify the port names:

```
show y %ci2:+[Q,D]
```

```verilog
 1  module memdemo(clk, d, y);
 2
 3  input clk;
 4  input [3:0] d;
 5  output reg [3:0] y;
 6
 7  integer i;
 8  reg [1:0] s1, s2;
 9  reg [3:0] mem [0:3];
10
11  always @(posedge clk) begin
12      for (i = 0; i < 4; i = i+1)
13          mem[i] <= mem[(i+1) % 4] + mem[(i+2) % 4];
14      { s2, s1 } = d ? { s1, s2 } ^ d : 4'b0;
15      mem[s1] <= d;
16      y <= mem[s2];
17  end
18
19  endmodule
```

Figure 14. Demo circuit for demonstrating some advanced Yosys features
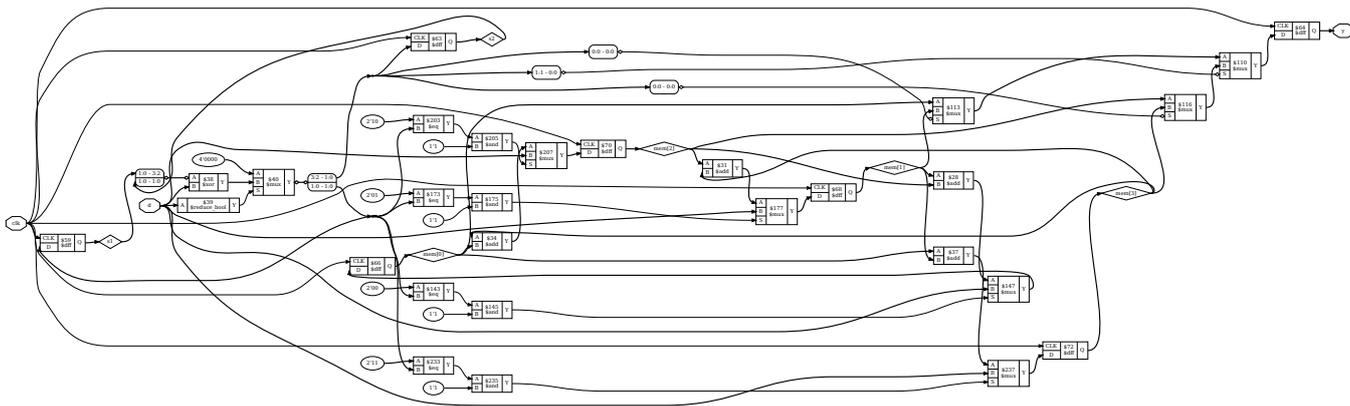
Figure 15. Complete circuit diagram for the design shown in Fig. 14

Or we could decide to tell the `%ci` action to not follow the `CLK` input:

```
show y %ci2:-[CLK]
```

Next we would investigate the next logic level by adding another `%ci2` to the command:

```
show y %ci2:-[CLK] %ci2
```

From this we would learn that the next cell is a `$mux` cell and we would add additional pattern to narrow the selection on the path we are interested. In the end we would end up with a command such as

```
show y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff
```

in which the first `%ci` jumps over the initial d-type flip-flop and the 2nd action selects the entire input cone without going over multiplexer select inputs and flip-flop cells. The diagram produces by this command is shown in Fig. 16.

Similar to `%ci` exists an action `%co` to select output cones that accepts the same syntax for pattern and repetition. The `%x` action mentioned previously also accepts this advanced syntax.

This actions for traversing the circuit graph, combined with the actions for boolean operations such as intersection (`%i`) and difference (`%d`) are powerful tools for extracting the relevant portions of the circuit under investigation.

See `help select` for a complete list of actions available in selections.
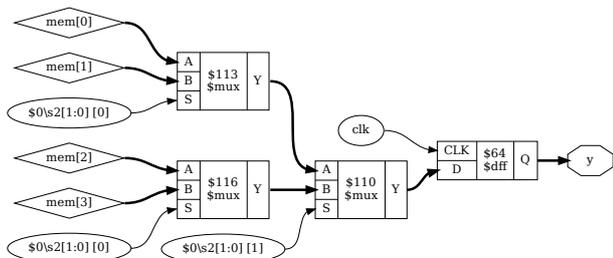
E. Storing and recalling selections

The current selection can be stored in memory with the command `select -set <name>`. It can later be recalled using `select @<name>`. In fact, the `@<name>` expression pushes the stored selection on the stack maintained by the `select` command. So for example

```
select @foo @bar %i
```

will select the intersection between the stored selections `foo` and `bar`.

In larger investigation efforts it is highly recommended to maintain a script that sets up relevant selections, so they can easily be recalled, for example when Yosys needs to be re-run after a design or source code change.

The `history` command can be used to list all recent interactive commands. This feature can be useful for creating such a script from the commands used in an interactive session.

V. ADVANCED INVESTIGATION TECHNIQUES

When working with very large modules, it is often not enough to just select the interesting part of the module. Instead it can be useful to extract the interesting part of the circuit into a separate module. This can for example be useful if one wants to run a series of synthesis commands on the critical part of the module and wants to carefully read all the debug output created by the commands in order to spot a problem. This kind of troubleshooting is much easier if the circuit under investigation is encapsulated in a separate module.

Fig. 17 shows how the `submod` command can be used to split the circuit from Fig. 14 and 15 into its components. The `-name` option is used to specify the name of the new module and also the name of the new cell in the current module.

A. Evaluation of combinatorial circuits

The `eval` command can be used to evaluate combinatorial circuits. For example (see Fig. 17 for the circuit diagram of `selstage`):

```
yosys [selstage]> eval -set s2,s1 4'b1001 -set d 4'hc -show n2 -show n1

9. Executing EVAL pass (evaluate the circuit given an input).
Full command line: eval -set s2,s1 4'b1001 -set d 4'hc -show n2 -show n1
Eval result: \n2 = 2'10.
Eval result: \n1 = 2'10.
```

So the `-set` option is used to set input values and the `-show` option is used to specify the nets to evaluate. If no `-show` option is specified, all selected output ports are used per default.

Figure 16. Output of `show y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff`
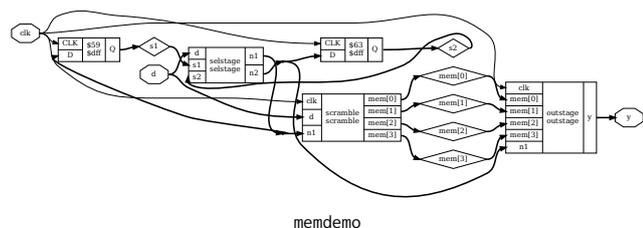
memdemo
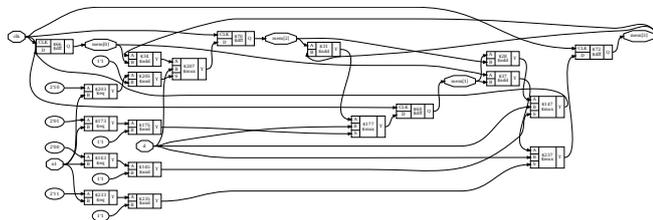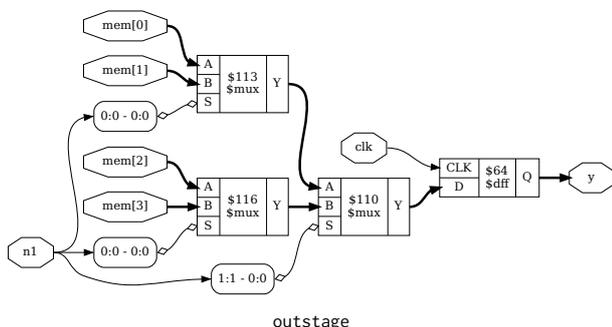
scramble

outstage
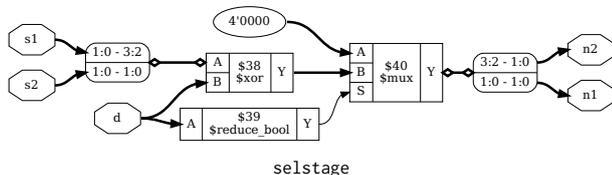
selstage

```
1   select -set outstage y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff
2   select -set selstage y %ci2:+$dff[Q,D] %ci*:-$dff @outstage %d
3   select -set scramble mem* %ci2 %ci*:-$dff mem* %d @selstage %d
4   submod -name scramble @scramble
5   submod -name outstage @outstage
6   submod -name selstage @selstage
```

Figure 17. The circuit from Fig. 14 and 15 broken up using submod

If a necessary input value is not given, an error is produced. The option -set-undef can be used to instead set all unspecified input nets to undef (x).

The -table option can be used to create a truth table. For example:

```
yosys [selstage]> eval -set-undef -set d[3:1] 0 -table s1,d[0]

10. Executing EVAL pass (evaluate the circuit given an input).
Full command line: eval -set-undef -set d[3:1] 0 -table s1,d[0]

 \s1 \d [0] |  \n1  \n2
 ---- ------ | ---- ----
 2'00   1'0 | 2'00 2'00
 2'00   1'1 | 2'xx 2'00
 2'01   1'0 | 2'00 2'00
 2'01   1'1 | 2'xx 2'01
 2'10   1'0 | 2'00 2'00
 2'10   1'1 | 2'xx 2'10
 2'11   1'0 | 2'00 2'00
```

```
2'11    1'1 | 2'xx 2'11

Assumed undef (x) value for the following signals: \s2
```

Note that the eval command (as well as the sat command discussed in the next sections) does only operate on flattened modules. It can not analyze signals that are passed through design hierarchy levels. So the flatten command must be used on modules that instantiate other modules before this commands can be applied.

B. Solving combinatorial SAT problems

Often the opposite of the eval command is needed, i.e. the circuits output is given and we want to find the matching input signals. For small circuits with only a few input bits this can be accomplished by trying all possible input combinations, as it is done by the eval -table command. For larger circuits however, Yosys provides the sat command that uses a SAT [4] solver [5] to solve this kind of problems.

The sat command works very similar to the eval command. The main difference is that it is now also possible to set output values and find the corresponding input values. For Example:

```
yosys [selstage]> sat -show s1,s2,d -set s1 s2 -set n2,n1 4'b1001

11. Executing SAT pass (solving SAT problems in the circuit).
Full command line: sat -show s1,s2,d -set s1 s2 -set n2,n1 4'b1001

Setting up SAT problem:
Import set-constraint: \s1 = \s2
Import set-constraint: { \n2 \n1 } = 4'1001
Final constraint equation: { \n2 \n1 \s1 } = { 4'1001 \s2 }
Imported 3 cells to SAT database.
Import show expression: { \s1 \s2 \d }

Solving problem with 81 variables and 207 clauses..
SAT solving finished - model found:

 Signal Name          Dec        Hex             Bin
 ------------------ ---------- ---------- ---------------
 \d                   9          9               1001
 \s1                  0          0               00
 \s2                  0          0               00
```

Note that the sat command supports signal names in both arguments to the -set option. In the above example we used -set s1 s2 to constraint s1 and s2 to be equal. When more complex constraints are needed, a wrapper circuit must be constructed that checks the constraints and signals if the constraint was met using an extra output port, which then can be forced to a value using the -set option. (Such a circuit that contains the circuit under test plus additional constraint checking circuitry is called a *miter* circuit.)

Fig. 18 shows a miter circuit that is supposed to be used as a prime number test. If ok is 1 for all input values a and b for a given p, then p is prime, or at least that is the idea.

The Yosys shell session shown in Fig. 19 demonstrates that SAT solvers can even find the unexpected solutions to a problem: Using integer overflow there actually is a way of "factorizing" 31. The clean solution would of course be to perform the test in 32 bits, for example by replacing p != a*b in the miter with p != {16'd0,a}*b, or by using a temporary variable for the 32 bit product a*b. But as 31 fits well into 8 bits (and as the purpose of this document is to show off Yosys

```
1   module primetest(p, a, b, ok);
2   input [15:0] p, a, b;
3   output ok = p != a*b || a == 1 || b == 1;
4   endmodule
```

Figure 18. A simple miter circuit for testing if a number is prime. But it has a problem (see main text and Fig. 19).

```
1   yosys [primetest]> sat -prove ok 1 -set p 31
2
3   8. Executing SAT pass (solving SAT problems in the circuit).
4   Full command line: sat -prove ok 1 -set p 31
5
6   Setting up SAT problem:
7   Import set-constraint: \p = 16'0000000000011111
8   Final constraint equation: \p = 16'0000000000011111
9   Imported 6 cells to SAT database.
10  Import proof-constraint: \ok = 1'1
11  Final proof equation: \ok = 1'1
12
13  Solving problem with 2790 variables and 8241 clauses..
14  SAT proof finished - model found: FAIL!
15
16    _____                    ___      ___       _ _             _ _
17   (_____ \                  / __)    / __)     (_) |           | | |
18    _____) )___ ___    ___ _| |__    _| |__ _____ _| | _____   __| | |
19   |  ____/ ___) _ \ / _ (_   __)  (_   __|____ | | || ___ |/ _  |_|
20   | |   | |  | |_| | |_| || |       | | / ___ | | || ____( (_| |_
21   |_|   |_|   \___/ \___/ |_|       |_|  \_____|_|\_)_____)\____|_|
22
23
24     Signal Name              Dec        Hex                 Bin
25     -------------------- ---------- ---------- --------------------
26     \a                     15029       3ab5       0011101010110101
27     \b                      4099       1003       0001000000000011
28     \ok                        0          0                      0
29     \p                        31         1f       0000000000011111
30
31  yosys [primetest]> sat -prove ok 1 -set p 31 -set a[15:8],b[15:8] 0
32
33  9. Executing SAT pass (solving SAT problems in the circuit).
34  Full command line: sat -prove ok 1 -set p 31 -set a[15:8],b[15:8] 0
35
36  Setting up SAT problem:
37  Import set-constraint: \p = 16'0000000000011111
38  Import set-constraint: { \a [15:8] \b [15:8] } = 16'0000000000000000
39  Final constraint equation: { \a [15:8] \b [15:8] \p } = { 16'0000000000000000 16'0000000000011111 }
40  Imported 6 cells to SAT database.
41  Import proof-constraint: \ok = 1'1
42  Final proof equation: \ok = 1'1
43
44  Solving problem with 2790 variables and 8257 clauses..
45  SAT proof finished - no model found: SUCCESS!
46
47                  /$$$$$$     /$$$$$$$$     /$$$$$$$
48                 /$$__  $$   | $$_____/    | $$__  $$
49                | $$  \ $$   | $$          | $$  \ $$
50                | $$  | $$   | $$$$$       | $$  | $$
51                | $$  | $$   | $$__/       | $$  | $$
52                | $$/$$ $$   | $$          | $$  | $$
53                |  $$$$$$/ /$$| $$$$$$$$ /$$| $$$$$$$//$$
54                 \____ $$$|__/|_____/|__/|_____/|__/
55                      \__/
```

Figure 19. Experiments with the miter circuit from Fig. 18. The first attempt of proving that 31 is prime failed because the SAT solver found a creative way of factorizing 31 using integer overflow.

features) we can also simply force the upper 8 bits of a and b to zero for the sat call, as is done in the second command in Fig. 19 (line 31).

The -prove option used in this example works similar to -set, but tries to find a case in which the two arguments are not equal. If such a case is not found, the property is proven to hold for all inputs that satisfy the other constraints.

It might be worth noting, that SAT solvers are not particularly efficient at factorizing large numbers. But if a small factorization problem occurs as part of a larger circuit problem, the Yosys SAT solver is perfectly capable of solving it.

C. Solving sequential SAT problems

The SAT solver functionality in Yosys can not only be used to solve combinatorial problems, but can also solve sequential problems. Let's

```
 1  yosys [memdemo]> sat -seq 6 -show y -show d -set-init-undef \
 2          -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3
 3
 4  6. Executing SAT pass (solving SAT problems in the circuit).
 5  Full command line: sat -seq 6 -show y -show d -set-init-undef
 6          -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3
 7
 8  Setting up time step 1:
 9  Final constraint equation: { } = { }
10  Imported 29 cells to SAT database.
11
12  Setting up time step 2:
13  Final constraint equation: { } = { }
14  Imported 29 cells to SAT database.
15
16  Setting up time step 3:
17  Final constraint equation: { } = { }
18  Imported 29 cells to SAT database.
19
20  Setting up time step 4:
21  Import set-constraint for timestep: \y = 4'0001
22  Final constraint equation: \y = 4'0001
23  Imported 29 cells to SAT database.
24
25  Setting up time step 5:
26  Import set-constraint for timestep: \y = 4'0010
27  Final constraint equation: \y = 4'0010
28  Imported 29 cells to SAT database.
29
30  Setting up time step 6:
31  Import set-constraint for timestep: \y = 4'0011
32  Final constraint equation: \y = 4'0011
33  Imported 29 cells to SAT database.
34
35  Setting up initial state:
36  Final constraint equation: { \y \s2 \s1 \mem[3] \mem[2] \mem[1]
37                              \mem[0] } = 24'xxxxxxxxxxxxxxxxxxxxxxxx
38
39  Import show expression: \y
40  Import show expression: \d
41
42  Solving problem with 10322 variables and 27881 clauses..
43  SAT model found. maximizing number of undefs.
44  SAT solving finished - model found:
45
46     Time Signal Name              Dec        Hex        Bin
47     ---- -------------------- ---------- ---------- ----------------
48     init \mem[0]                  --         --                 xxxx
49     init \mem[1]                  --         --                 xxxx
50     init \mem[2]                  --         --                 xxxx
51     init \mem[3]                  --         --                 xxxx
52     init \s1                      --         --                   xx
53     init \s2                      --         --                   xx
54     init \y                       --         --                 xxxx
55     ---- -------------------- ---------- ---------- ----------------
56        1 \d                        0          0                 0000
57        1 \y                       --         --                 xxxx
58     ---- -------------------- ---------- ---------- ----------------
59        2 \d                        1          1                 0001
60        2 \y                       --         --                 xxxx
61     ---- -------------------- ---------- ---------- ----------------
62        3 \d                        2          2                 0010
63        3 \y                        0          0                 0000
64     ---- -------------------- ---------- ---------- ----------------
65        4 \d                        3          3                 0011
66        4 \y                        1          1                 0001
67     ---- -------------------- ---------- ---------- ----------------
68        5 \d                       --         --                 001x
69        5 \y                        2          2                 0010
70     ---- -------------------- ---------- ---------- ----------------
71        6 \d                       --         --                 xxxx
72        6 \y                        3          3                 0011
```

Figure 20. Solving a sequential SAT problem in the memdemo module from Fig. 14.

consider the entire memdemo module from Fig. 14 and suppose we want to know which sequence of input values for d will cause the output y to produce the sequence 1, 2, 3 from any initial state. Fig. 20 show the solution to this question, as produced by the following command:

```
sat -seq 6 -show y -show d -set-init-undef \
```

```
        -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3
```

The -seq 6 option instructs the sat command to solve a sequential problem in 6 time steps. (Experiments with lower number of steps have show that at least 3 cycles are necessary to bring the circuit in a state from which the sequence 1, 2, 3 can be produced.)

The -set-init-undef option tells the sat command to initialize all registers to the undef (x) state. The way the x state is treated in Verilog will ensure that the solution will work for any initial state.

The -max_undef option instructs the sat command to find a solution with a maximum number of undefs. This way we can see clearly which inputs bits are relevant to the solution.

Finally the three -set-at options add constraints for the y signal to play the 1, 2, 3 sequence, starting with time step 4.

It is not surprising that the solution sets d = 0 in the first step, as this is the only way of setting the s1 and s2 registers to a known value. The input values for the other steps are a bit harder to work out manually, but the SAT solver finds the correct solution in an instant.

There is much more to write about the sat command. For example, there is a set of options that can be used to performs sequential proofs using temporal induction [6]. The command help sat can be used to print a list of all options with short descriptions of their functions.

VI. CONCLUSION

Yosys provides a wide range of functions to analyze and investigate designs. For many cases it is sufficient to simply display circuit diagrams, maybe use some additional commands to narrow the scope of the circuit diagrams to the interesting parts of the circuit. But some cases require more than that. For this applications Yosys provides commands that can be used to further inspect the behavior of the circuit, either by evaluating which output values are generated from certain input values (eval) or by evaluation which input values and initial conditions can result in a certain behavior at the outputs (sat). The SAT command can even be used to prove (or disprove) theorems regarding the circuit, in more advanced cases with the additional help of a miter circuit.

This features can be powerful tools for the circuit designer using Yosys as a utility for building circuits and the software developer using Yosys as a framework for new algorithms alike.

REFERENCES

[1] Claire Xenia Wolf. The Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/
[2] Graphviz - Graph Visualization Software. http://www.graphviz.org/
[3] xdot.py - an interactive viewer for graphs written in Graphviz's dot language. https://github.com/jrfonseca/xdot.py
[4] *Circuit satisfiability problem* on Wikipedia http://en.wikipedia.org/wiki/Circuit_satisfiability
[5] MiniSat: a minimalistic open-source SAT solver. http://minisat.se/
[6] Niklas Een and Niklas Sörensson (2003). Temporal Induction by Incremental SAT Solving. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.4.8161

Yosys Application Note 012:
Converting Verilog to BTOR

Ahmed Irfan and Claire Xenia Wolf

April 2015

Abstract—**Verilog-2005 is a powerful Hardware Description Language (HDL) that can be used to easily create complex designs from small HDL code. BTOR [3] is a bit-precise word-level format for model checking. It is a simple format and easy to parse. It allows to model the model checking problem over the theory of bit-vectors with one-dimensional arrays, thus enabling to model Verilog designs with registers and memories. Yosys [1] is an Open-Source Verilog synthesis tool that can be used to convert Verilog designs with simple assertions to BTOR format.**

I. Installation

Yosys written in C++ (using features from C++11) and is tested on modern Linux. It should compile fine on most UNIX systems with a C++11 compiler. The README file contains useful information on building Yosys and its prerequisites.

Yosys is a large and feature-rich program with some dependencies. For this work, we may deactivate other extra features such as TCL and ABC support in the Makefile.

This Application Note is based on GIT Rev. 082550f from 2015-04-04 of Yosys [1].

II. Quick Start

We assume that the Verilog design is synthesizable and we also assume that the design does not have multi-dimensional memories. As BTOR implicitly initializes registers to zero value and memories stay uninitialized, we assume that the Verilog design does not contain initial blocks. For more details about the BTOR format, please refer to [3].

We provide a shell script verilog2btor.sh which can be used to convert a Verilog design to BTOR. The script can be found in the backends/btor directory. The following example shows its usage:

```
verilog2btor.sh fsm.v fsm.btor test
```

Listing 1. Using verilog2btor script

The script verilog2btor.sh takes three parameters. In the above example, the first parameter fsm.v is the input design, the second parameter fsm.btor is the file name of BTOR output, and the third parameter test is the name of top module in the design.

To specify the properties (that need to be checked), we have two options:

- We can use the Verilog assert statement in the procedural block or module body of the Verilog design, as shown in Listing 2. This is the preferred option.
- We can use a single-bit output wire, whose name starts with safety. The value of this output wire needs to be driven low when the property is met, i.e. the solver will try to find a model that makes the safety pin go high. This is demonstrated in Listing 3.

```
module test(input clk, input rst, output y);

  reg [2:0] state;

  always @(posedge clk) begin
    if (rst || state == 3) begin
      state <= 0;
    end else begin
      assert(state < 3);
      state <= state + 1;
    end
  end

  assign y = state[2];

  assert property (y !== 1'b1);

endmodule
```

Listing 2. Specifying property in Verilog design with assert

```
module test(input clk, input rst,
    output y, output safety1);

  reg [2:0] state;

  always @(posedge clk) begin
    if (rst || state == 3)
      state <= 0;
    else
      state <= state + 1;
  end

  assign y = state[2];

  assign safety1 = !(y !== 1'b1);

endmodule
```

Listing 3. Specifying property in Verilog design with output wire

We can run Boolector [2] 1.4.1[1] on the generated BTOR file:

```
$ boolector fsm.btor
unsat
```

Listing 4. Running boolector on BTOR file

We can also use nuXmv [4], but on BTOR designs it does not support memories yet. With the next release of nuXmv, we will be also able to verify designs with memories.

III. Detailed Flow

Yosys is able to synthesize Verilog designs up to the gate level. We are interested in keeping registers and memories when synthesizing the design. For this purpose, we describe a customized Yosys synthesis flow, that is also provided by the verilog2btor.sh script. Listing 5 shows the Yosys commands that are executed by verilog2btor.sh.

---

[1] Newer version of Boolector do not support sequential models. Boolector 1.4.1 can be built with picosat-951. Newer versions of picosat have an incompatible API.

```
1   read_verilog -sv $1;
2   hierarchy -top $3; hierarchy -libdir $DIR;
3   hierarchy -check;
4   proc; opt;
5   opt_expr -mux_undef; opt;
6   rename -hide;;;
7   splice; opt;
8   memory_dff -wr_only; memory_collect;;
9   flatten;;
10  memory_unpack;
11  splitnets -driver;
12  setundef -zero -undriven;
13  opt;;;
14  write_btor $2;
```

Listing 5. Synthesis Flow for BTOR with memories

Here is short description of what is happening in the script line by line:

1) Reading the input file.
2) Setting the top module in the hierarchy and trying to read automatically the files which are given as `include` in the file read in first line.
3) Checking the design hierarchy.
4) Converting processes to multiplexers (muxs) and flip-flops.
5) Removing undef signals from muxs.
6) Hiding all signal names that are not used as module ports.
7) Explicit type conversion, by introducing slice and concat cells in the circuit.
8) Converting write memories to synchronous memories, and collecting the memories to multi-port memories.
9) Flattening the design to get only one module.
10) Separating read and write memories.
11) Splitting the signals that are partially assigned
12) Setting undef to zero value.
13) Final optimization pass.
14) Writing BTOR file.

For detailed description of the commands mentioned above, please refer to the Yosys documentation, or run `yosys -h` *command_name*.

The script presented earlier can be easily modified to have a BTOR file that does not contain memories. This is done by removing the line number 8 and 10, and introduces a new command `memory` at line number 8. Listing 6 shows the modified Yosys script file:

```
read_verilog -sv $1;
hierarchy -top $3; hierarchy -libdir $DIR;
hierarchy -check;
proc; opt;
opt_expr -mux_undef; opt;
rename -hide;;;
splice; opt;
memory;;
flatten;;
splitnets -driver;
setundef -zero -undriven;
opt;;;
write_btor $2;
```

Listing 6. Synthesis Flow for BTOR without memories

IV. EXAMPLE

Here is an example Verilog design that we want to convert to BTOR:

```verilog
module array(input clk);

  reg [7:0] counter;
  reg [7:0] mem [7:0];

  always @(posedge clk) begin
    counter <= counter + 8'd1;
    mem[counter] <= counter;
  end

  assert property (!(counter > 8'd0) ||
    mem[counter - 8'd1] == counter - 8'd1);

endmodule
```

Listing 7. Example - Verilog Design

The generated BTOR file that contain memories, using the script shown in Listing 5:

```
1 var 1 clk
2 array 8 3
3 var 8 $auto$rename.cc:150:execute$20
4 const 8 00000001
5 sub 8 3 4
6 slice 3 5 2 0
7 read 8 2 6
8 slice 3 3 2 0
9 add 8 3 4
10 const 8 00000000
11 ugt 1 3 10
12 not 1 11
13 const 8 11111111
14 slice 1 13 0 0
15 one 1
16 eq 1 1 15
17 and 1 16 14
18 write 8 3 2 8 3
19 acond 8 3 17 18 2
20 anext 8 3 2 19
21 eq 1 7 5
22 or 1 12 21
23 const 1 1
24 one 1
25 eq 1 23 24
26 cond 1 25 22 24
27 root 1 -26
28 cond 8 1 9 3
29 next 8 3 28
```

Listing 8. Example - Converted BTOR with memory

And the BTOR file obtained by the script shown in Listing 6, which expands the memory into individual elements:

```
1 var 1 clk
2 var 8 mem[0]
3 var 8 $auto$rename.cc:150:execute$20
4 slice 3 3 2 0
5 slice 1 4 0 0
6 not 1 5
7 slice 1 4 1 1
8 not 1 7
9 slice 1 4 2 2
10 not 1 9
11 and 1 8 10
12 and 1 6 11
13 cond 8 12 3 2
14 cond 8 1 13 2
15 next 8 2 14
16 const 8 00000001
17 add 8 3 16
18 const 8 00000000
19 ugt 1 3 18
20 not 1 19
21 var 8 mem[2]
22 and 1 7 10
23 and 1 6 22
24 cond 8 23 3 21
25 cond 8 1 24 21
26 next 8 21 25
27 sub 8 3 16
   .
   .
   .
54 cond 1 53 50 52
55 root 1 -54
   .
   .
   .
77 cond 8 76 3 44
78 cond 8 1 77 44
79 next 8 44 78
```

Listing 9. Example - Converted BTOR without memory

V. LIMITATIONS

BTOR does not support initialization of memories and registers, i.e. they are implicitly initialized to value zero, so the initial block for memories need to be removed when converting to BTOR. It should also be kept in consideration that BTOR does not support the x or z values of Verilog.

Another thing to bear in mind is that Yosys will convert multi-dimensional memories to one-dimensional memories and address decoders. Therefore out-of-bounds memory accesses can yield unexpected results.

VI. CONCLUSION

Using the described flow, we can use Yosys to generate word-level verification benchmarks with or without memories from Verilog designs.

REFERENCES

[1] Claire Xenia Wolf. The Yosys Open SYnthesis Suite.
    https://yosyshq.net/yosys/
[2] Robert Brummayer and Armin Biere, Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays
    http://fmv.jku.at/boolector/
[3] Robert Brummayer and Armin Biere and Florian Lonsing, BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking
    http://fmv.jku.at/papers/BrummayerBiereLonsing-BPR08.pdf
[4] Roberto Cavada and Alessandro Cimatti and Michele Dorigatti and Alberto Griggio and Alessandro Mariotti and Andrea Micheli and Sergio Mover and Marco Roveri and Stefano Tonetta, The nuXmv Symbolic Model Checker
    https://es-static.fbk.eu/tools/nuxmv/index.php

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

[GW13] Glaser, Johann ; Wolf, C.: 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

[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

[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, C. ; 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

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

Internet References

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

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

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

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

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

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

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

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

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

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

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

[27] 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/.

[28] 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/.

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