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

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.

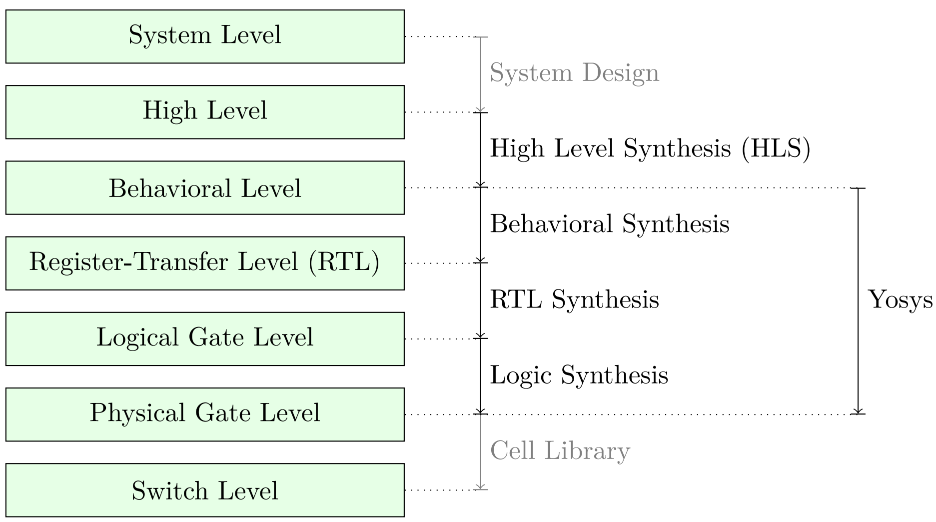


Figure 2.1: Different levels of abstraction and synthesis.

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:

```
always @(posedge clk)
        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:

```
assign tmp = a + b;       // combinatorial data path

always @(posedge clk)     // register
        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:

```
// asynchronous
always @* begin
	if (add_mode)
		y <= a + b;
	else
		y <= a - b;
end

// latched
always @* begin
	if (!hold)
		y <= a + b;
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:

```
// counter with synchronous reset
always @(posedge clk) begin
	if (reset)
		y <= 0;
	else
		y <= y + 1;
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:

```
// counter with asynchronous reset
always @(posedge clk, posedge reset) begin
	if (reset)
		y <= 0;
	else
		y <= y + 1;
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:

```
module (clk, in_data, out_data);

parameter BITS = 8;
parameter STAGES = 4;

input clk;
input [BITS-1:0] in_data;
output [BITS-1:0] out_data;
reg [BITS-1:0] ffs [STAGES-1:0];

integer i;
always @(posedge clk) begin
	ffs[0] <= in_data;
	for (i = 1; i < STAGES; i = i+1)
		ffs[i] <= ffs[i-1];
end

assign out_data = ffs[STAGES-1];

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.

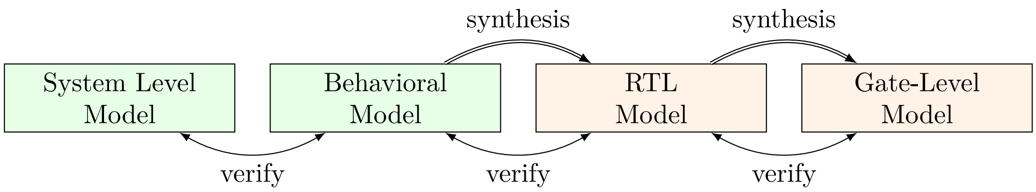


Figure 2.2: Typical design flow. Green boxes represent manually created models. Orange boxes represent models generated by synthesis tools.

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):

```
assign_stmt: TOK_ASSIGN TOK_IDENTIFIER TOK_EQ expr TOK_SEMICOLON;
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]

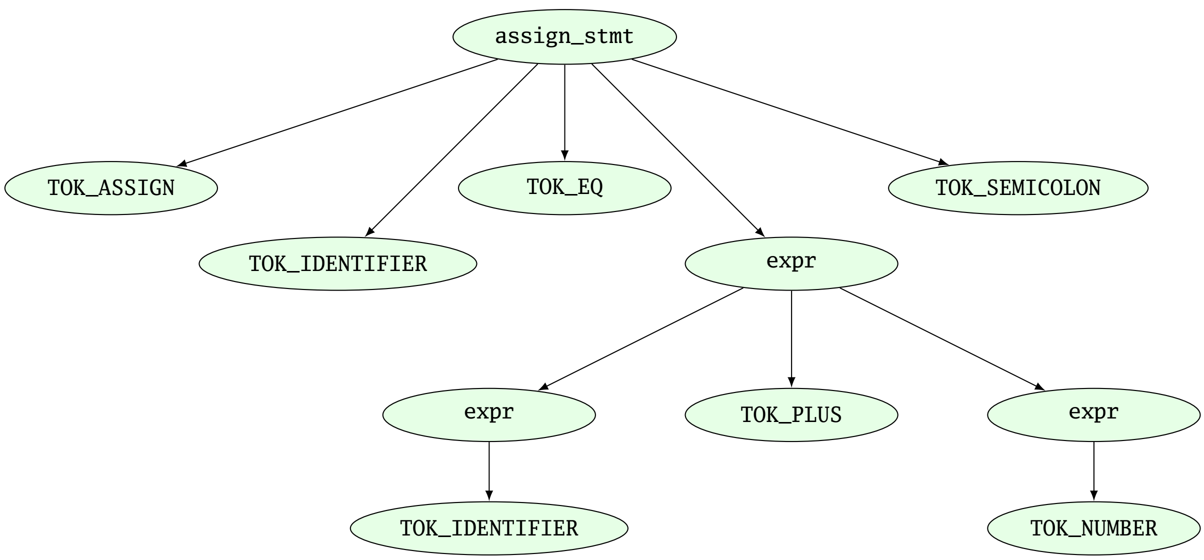


Figure 2.3: Example parse tree for the Verilog expression “**assign** foo = bar + 42;”.

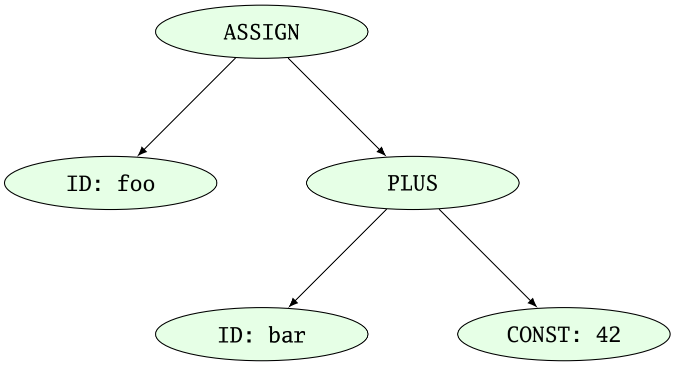


Figure 2.4: Example abstract syntax tree for the Verilog expression "`assign foo = bar + 42;`".

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.

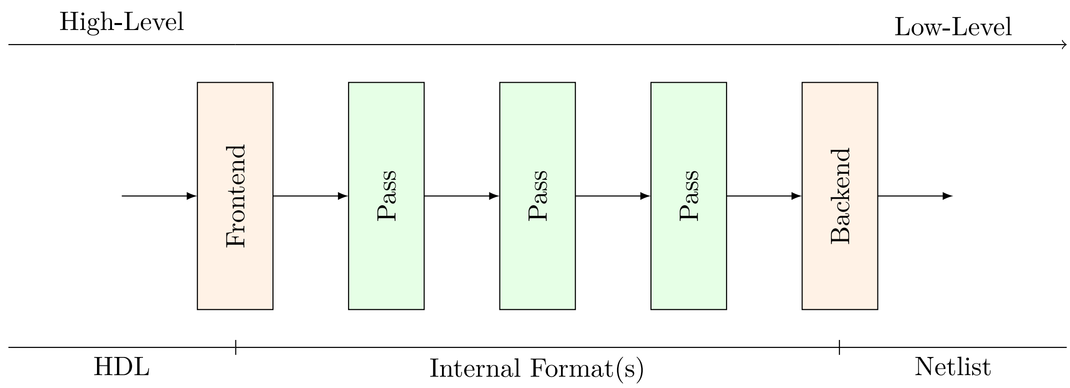


Figure 3.1: General data- and control-flow of a synthesis tool

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`:

```
# read input file to internal representation
read_verilog design.v

# convert high-level behavioral parts ("processes") to d-type flip-flops and muxes
proc
```

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

```

# perform some simple optimizations
opt

# convert high-level memory constructs to d-type flip-flops and multiplexers
memory

# perform some simple optimizations
opt

# convert design to (logical) gate-level netlists
techmap

# perform some simple optimizations
opt

# map internal register types to the ones from the cell library
dfflibmap -liberty cells.lib

# use ABC to map remaining logic to cells from the cell library
abc -liberty cells.lib

# cleanup
opt

# write results to output file
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^2C 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).

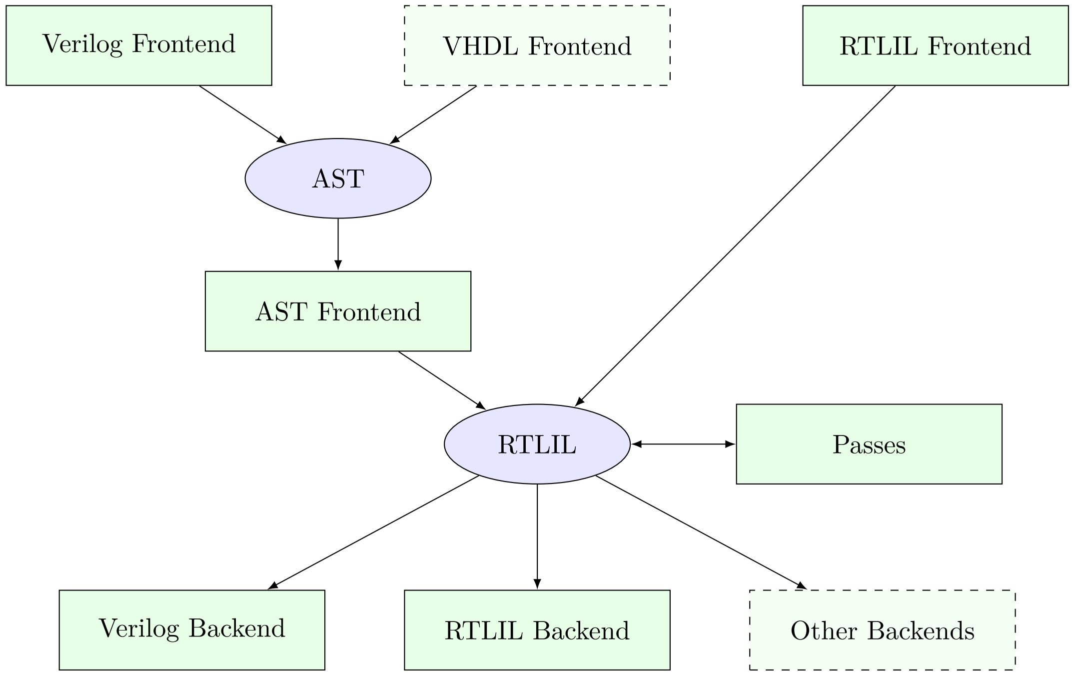


Figure 4.1: Yosys simplified data flow (ellipses: data structures, rectangles: program modules)

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.

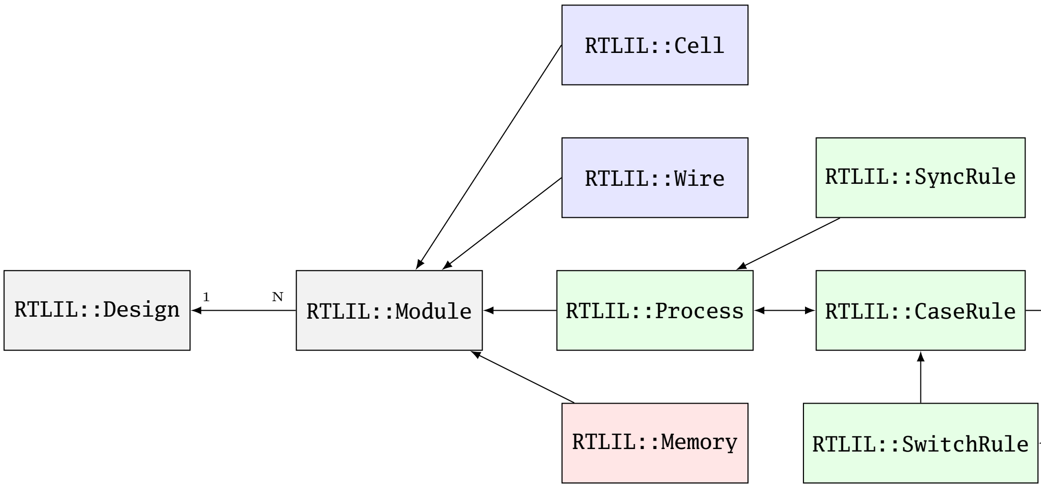


Figure 4.2: Simplified RTLIL Entity-Relationship Diagram

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

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:

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

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

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

An RTLIL::SyncRule object contains an (optional) synchronization condition (signal and edge-type), 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:

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

This pass has transformed the outer RTLIL::SwitchRule into a modified RTLIL::SyncRule object for the `\reset` signal. Further processing converts the RTLIL::Process 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").

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

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

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

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)

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.

```
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;
```

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)

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.

```
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)
```

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)

```
        if (R == RstLvl)
                Q <= 0;
        else if (S == SetLvl)
                Q <= 1;
        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:

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)

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

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)

```
    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 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

```
Getting Started
===============



Outline of a Yosys command
--------------------------

Here is a the C++ code for a "hello_world" Yosys command (hello.cc):

	#include "kernel/yosys.h"

	USING_YOSYS_NAMESPACE
	PRIVATE_NAMESPACE_BEGIN

	struct HelloWorldPass : public Pass {
		HelloWorldPass() : Pass("hello_world") { }
		void execute(vector<string>, Design*) override {
			log("Hello World!\n");
		}
	} HelloWorldPass;

	PRIVATE_NAMESPACE_END

This can be built into a Yosys module using the following command:

	yosys-config --exec --cxx --cxxflags --ldflags -o hello.so -shared hello.cc --ldlibs
```

```

Or short:

        yosys-config --build hello.so hello.cc

And then executed using the following command:

        yosys -m hello.so -p hello_world


Yosys Data Structures
---------------------

Here is a short list of data structures that you should make yourself familiar
with before you write C++ code for Yosys. The following data structures are all
defined when "kernel/yosys.h" is included and USING_YOSYS_NAMESPACE is used.

  1. Yosys Container Classes

Yosys uses dict<K, T> and pool<T> as main container classes. dict<K, T> is
essentially a replacement for std::unordered_map<K, T> and pool<T> is a
replacement for std::unordered_set<T>. The main characteristics are:

        - dict<K, T> and pool<T> are about 2x faster than the std containers

        - references to elements in a dict<K, T> or pool<T> are invalidated by
          insert and remove operations (similar to std::vector<T> on push_back()).

        - some iterators are invalidated by erase(). specifically, iterators
          that have not passed the erased element yet are invalidated. (erase()
          itself returns valid iterator to the next element.)

        - no iterators are invalidated by insert(). elements are inserted at
          begin(). i.e. only a new iterator that starts at begin() will see the
          inserted elements.

        - the method .count(key, iterator) is like .count(key) but only
          considers elements that can be reached via the iterator.

        - iterators can be compared. it1 < it2 means that the position of t2
          can be reached via t1 but not vice versa.

        - the method .sort() can be used to sort the elements in the container
          the container stays sorted until elements are added or removed.

        - dict<K, T> and pool<T> will have the same order of iteration across
          all compilers, standard libraries and architectures.

In addition to dict<K, T> and pool<T> there is also an idict<K> that
creates a bijective map from K to the integers. For example:

        idict<string, 42> si;
        log("%d\n", si("hello"));      // will print 42
        log("%d\n", si("world"));      // will print 43
```

```
        log("%d\n", si.at("world"));   // will print 43
        log("%d\n", si.at("dummy"));   // will throw exception
        log("%s\n", si[42].c_str()));  // will print hello
        log("%s\n", si[43].c_str()));  // will print world
        log("%s\n", si[44].c_str()));  // will throw exception

It is not possible to remove elements from an idict.

Finally mfp<K> implements a merge-find set data structure (aka. disjoint-set or
union-find) over the type K ("mfp" = merge-find-promote).

  2. Standard STL data types

In Yosys we use std::vector<T> and std::string whenever applicable. When
dict<K, T> and pool<T> are not suitable then std::map<K, T> and std::set<T>
are used instead.

The types std::vector<T> and std::string are also available as vector<T>
and string in the Yosys namespace.

  3. RTLIL objects

The current design (essentially a collection of modules, each defined by a
netlist) is stored in memory using RTLIL object (declared in kernel/rtlil.h,
automatically included by kernel/yosys.h). You should glance over at least
the declarations for the following types in kernel/rtlil.h:

        RTLIL::IdString
                This is a handle for an identifier (e.g. cell or wire name).
                It feels a lot like a std::string, but is only a single int
                in size. (The actual string is stored in a global lookup
                table.)

        RTLIL::SigBit
                A single signal bit. I.e. either a constant state (0, 1,
                x, z) or a single bit from a wire.

        RTLIL::SigSpec
                Essentially a vector of SigBits.

        RTLIL::Wire
        RTLIL::Cell
                The building blocks of the netlist in a module.

        RTLIL::Module
        RTLIL::Design
                The module is a container with connected cells and wires
                in it. The design is a container with modules in it.

All this types are also available without the RTLIL:: prefix in the Yosys
namespace.

  4. SigMap and other Helper Classes

```

```
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
```

```
------------

The following yosys commands are a good starting point if you are looking for examples
of how to use the Yosys API:

	manual/CHAPTER_Prog/stubnets.cc
	manual/PRESENTATION_Prog/my_cmd.cc


Script Passes
-------------

The ScriptPass base class can be used to implement passes that just call other passes,
like a script. Examples for such passes are:

	techlibs/common/prep.cc
	techlibs/common/synth.cc

In some cases it is easier to implement such a pass as regular pass, for example when
ScriptPass doesn't provide the type of flow control desired. (But many of the
script passes in Yosys that don't use ScriptPass simply predate the ScriptPass base
class.) Examples for such passes are:

	passes/opt/opt.cc
	passes/proc/proc.cc

Whether they use the ScriptPass base-class or not, a pass should always either
call other passes without doing any non-trivial work itself, or should implement
a non-trivial algorithm but not call any other passes. The reason for this is that
this helps containing complexity in individual passes and simplifies debugging the
entire system.

Exceptions to this rule should be rare and limited to cases where calling other
passes is optional and only happens when requested by the user (such as for
example `techmap -autoproc`), or where it is about commands that are "top-level
commands" in their own right, not components to be used in regular synthesis
flows (such as the `bugpoint` command).

A pass that would "naturally" call other passes and also do some work itself
should be re-written in one of two ways:

1) It could be re-written as script pass with the parts that are not calls
to other passes factored out into individual new passes. Usually in those
cases the new sub passes share the same prefix as the top-level script pass.

2) It could be re-written so that it already expects the design in a certain
state, expecting the calling script to set up this state before calling the
pass in questions.

Many back-ends are examples for the 2nd approach. For example, `write_aiger`
does not convert the design into AIG representation, but expects the design
to be already in this form, and prints an `Unsupported cell type` error
message otherwise.

```

```

Notes on the existing codebase
------------------------------

For historical reasons not all parts of Yosys adhere to the current coding
style. When adding code to existing parts of the system, adhere to this guide
for the new code instead of trying to mimic the style of the surrounding code.
```

CodingStyle

```
Coding Style
============


Formatting of code
------------------

- Yosys code is using tabs for indentation. Tabs are 8 characters.

- A continuation of a statement in the following line is indented by
  two additional tabs.

- Lines are as long as you want them to be. A good rule of thumb is
  to break lines at about column 150.

- Opening braces can be put on the same or next line as the statement
  opening the block (if, switch, for, while, do). Put the opening brace
  on its own line for larger blocks, especially blocks that contains
  blank lines.

- Otherwise stick to the Linux Kernel Coding Style:
    https://www.kernel.org/doc/Documentation/CodingStyle


C++ Language
-------------

Yosys is written in C++11. At the moment only constructs supported by
gcc 4.8 are allowed in Yosys code. This will change in future releases.

In general Yosys uses "int" instead of "size_t". To avoid compiler
warnings for implicit type casts, always use "GetSize(foobar)" instead
of "foobar.size()". (GetSize() is defined in kernel/yosys.h)

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

```
// This is free and unencumbered software released into the public domain.
//
// Anyone is free to copy, modify, publish, use, compile, sell, or
// distribute this software, either in source code form or as a compiled
// binary, for any purpose, commercial or non-commercial, and by any
// means.

#include "kernel/yosys.h"
#include "kernel/sigtools.h"

#include <string>
#include <map>
#include <set>

USING_YOSYS_NAMESPACE
PRIVATE_NAMESPACE_BEGIN

// this function is called for each module in the design
static void find_stub_nets(RTLIL::Design *design, RTLIL::Module *module, bool report_bits)
{
	// use a SigMap to convert nets to a unique representation
	SigMap sigmap(module);

	// count how many times a single-bit signal is used
	std::map<RTLIL::SigBit, int> bit_usage_count;

	// count output lines for this module (needed only for summary output at the end)
	int line_count = 0;

	log("Looking␣for␣stub␣wires␣in␣module␣%s:\n", RTLIL::id2cstr(module->name));

	// For all ports on all cells
	for (auto &cell_iter : module->cells_)
	for (auto &conn : cell_iter.second->connections())
	{
		// Get the signals on the port
		// (use sigmap to get a uniqe signal name)
		RTLIL::SigSpec sig = sigmap(conn.second);

		// add each bit to bit_usage_count, unless it is a constant
		for (auto &bit : sig)
			if (bit.wire != NULL)
				bit_usage_count[bit]++;
	}

	// for each wire in the module
	for (auto &wire_iter : module->wires_)
	{
		RTLIL::Wire *wire = wire_iter.second;

		// .. but only selected wires
		if (!design->selected(module, wire))
			continue;

```

```
			// add +1 usage if this wire actually is a port
			int usage_offset = wire->port_id > 0 ? 1 : 0;

			// we will record which bits of the (possibly multi-bit) wire are stub signals
			std::set<int> stub_bits;

			// get a signal description for this wire and split it into separate bits
			RTLIL::SigSpec sig = sigmap(wire);

			// for each bit (unless it is a constant):
			// check if it is used at least two times and add to stub_bits otherwise
			for (int i = 0; i < GetSize(sig); i++)
				if (sig[i].wire != NULL && (bit_usage_count[sig[i]] + usage_offset) < 2)
					stub_bits.insert(i);

			// continue if no stub bits found
			if (stub_bits.size() == 0)
				continue;

			// report stub bits and/or stub wires, don't report single bits
			// if called with report_bits set to false.
			if (GetSize(stub_bits) == GetSize(sig)) {
				log("  found stub wire: %s\n", RTLIL::id2cstr(wire->name));
			} else {
				if (!report_bits)
					continue;
				log("  found wire with stub bits: %s [", RTLIL::id2cstr(wire->name));
				for (int bit : stub_bits)
					log("%s%d", bit == *stub_bits.begin() ? "" : ", ", bit);
				log("]\n");
			}

			// we have outputted a line, increment summary counter
			line_count++;
		}

		// report summary
		if (report_bits)
			log("  found %d stub wires or wires with stub bits.\n", line_count);
		else
			log("  found %d stub wires.\n", line_count);
}

// each pass contains a singleton object that is derived from Pass
struct StubnetsPass : public Pass {
	StubnetsPass() : Pass("stubnets") { }
	void execute(std::vector<std::string> args, RTLIL::Design *design) override
	{
		// variables to mirror information from passed options
		bool report_bits = 0;

		log_header(design, "Executing STUBNETS pass (find stub nets).\n");

		// parse options
```

```
			size_t argidx;
			for (argidx = 1; argidx < args.size(); argidx++) {
				std::string arg = args[argidx];
				if (arg == "-report_bits") {
					report_bits = true;
					continue;
				}
				break;
			}

			// handle extra options (e.g. selection)
			extra_args(args, argidx, design);

			// call find_stub_nets() for each module that is either
			// selected as a whole or contains selected objects.
			for (auto &it : design->modules_)
				if (design->selected_module(it.first))
					find_stub_nets(design, it.second, report_bits);
	}
} StubnetsPass;

PRIVATE_NAMESPACE_END
```

Makefile

```
test: stubnets.so
	yosys -ql test1.log -m ./stubnets.so test.v -p "stubnets"
	yosys -ql test2.log -m ./stubnets.so test.v -p "opt;␣stubnets"
	yosys -ql test3.log -m ./stubnets.so test.v -p "techmap;␣opt;␣stubnets␣-report_bits"
	tail test1.log test2.log test3.log

stubnets.so: stubnets.cc
	yosys-config --exec --cxx --cxxflags --ldflags -o $@ -shared $^ --ldlibs

clean:
	rm -f test1.log test2.log test3.log
	rm -f stubnets.so stubnets.d
```

test.v

```
module uut(in1, in2, in3, out1, out2);

input [8:0] in1, in2, in3;
output [8:0] out1, out2;

assign out1 = in1 + in2 + (in3 >> 4);

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.

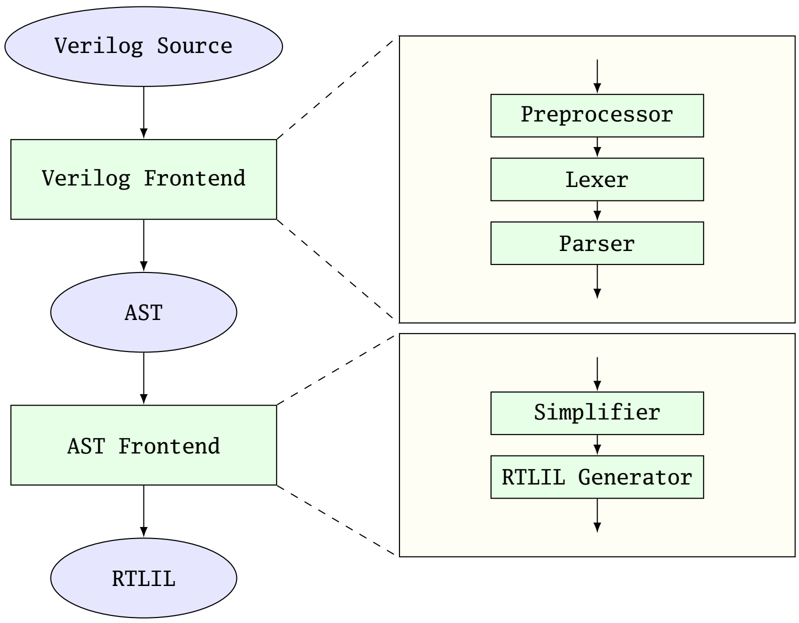


Figure 7.1: Simplified Verilog to RTLIL data flow

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:

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

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)

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:

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

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

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

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

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

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

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

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

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

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

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

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

7.3.1 The ProcessGenerator Algorithm

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

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

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

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

As the algorithm runs these variables are 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::SyncRule`s that describe the output registers.
- `proc_dff`
  This pass replaces the `RTLIL::SyncRule`s to d-type flip-flops (with asynchronous resets if necessary).
- `proc_dff`
  This pass replaces the `RTLIL::MemWriteAction`s 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:

```
module uut(a, y);
input a;
output [1:0] y = a ? (a ? 1 : 2) : 3;
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.202). 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.162).

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

```
    abc [options] [selection]

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

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

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

        if <file> starts with a plus sign (+), then the rest of the filename
        string is interpreted as the command string to be passed to ABC. The
        leading plus sign is removed and all commas (,) in the string are
        replaced with blanks before the string is passed to ABC.

        if no -script parameter is given, the following scripts are used:

        for -liberty/-genlib without -constr:
          strash; ifraig; scorr; dc2; dretime; strash; &get -n; &dch -f;
               &nf {D}; &put

        for -liberty/-genlib with -constr:
          strash; ifraig; scorr; dc2; dretime; strash; &get -n; &dch -f;
               &nf {D}; &put; buffer; upsize {D}; dnsize {D}; stime -p

        for -lut/-luts (only one LUT size):
          strash; ifraig; scorr; dc2; dretime; strash; dch -f; if; mfs2;
               lutpack {S}

        for -lut/-luts (different LUT sizes):
          strash; ifraig; scorr; dc2; dretime; strash; dch -f; if; mfs2

        for -sop:
```

```
          strash; ifraig; scorr; dc2; dretime; strash; dch -f;
               cover {I} {P}

        otherwise:
          strash; ifraig; scorr; dc2; dretime; strash; &get -n; &dch -f;
               &nf {D}; &put

    -fast
        use different default scripts that are slightly faster (at the cost
        of output quality):

        for -liberty/-genlib without -constr:
          strash; dretime; map {D}

        for -liberty/-genlib with -constr:
          strash; dretime; map {D}; buffer; upsize {D}; dnsize {D};
               stime -p

        for -lut/-luts:
          strash; dretime; if

        for -sop:
          strash; dretime; cover {I} {P}

        otherwise:
          strash; dretime; map

    -liberty <file>
        generate netlists for the specified cell library (using the liberty
        file format).

    -genlib <file>
        generate netlists for the specified cell library (using the SIS Genlib
        file format).

    -constr <file>
        pass this file with timing constraints to ABC.
        use with -liberty/-genlib.

        a constr file contains two lines:
            set_driving_cell <cell_name>
            set_load <floating_point_number>

        the set_driving_cell statement defines which cell type is assumed to
        drive the primary inputs and the set_load statement sets the load in
        femtofarads for each primary output.

    -D <picoseconds>
        set delay target. the string {D} in the default scripts above is
        replaced by this option when used, and an empty string otherwise.
        this also replaces 'dretime' with 'dretime; retime -o {D}' in the
        default scripts above.

    -I <num>
```

```
        maximum number of SOP inputs.
        (replaces {I} in the default scripts above)

    -P <num>
        maximum number of SOP products.
        (replaces {P} in the default scripts above)

    -S <num>
        maximum number of LUT inputs shared.
        (replaces {S} in the default scripts above, default: -S 1)

    -lut <width>
        generate netlist using luts of (max) the specified width.

    -lut <w1>:<w2>
        generate netlist using luts of (max) the specified width <w2>. All
        luts with width <= <w1> have constant cost. for luts larger than <w1>
        the area cost doubles with each additional input bit. the delay cost
        is still constant for all lut widths.

    -luts <cost1>,<cost2>,<cost3>,<sizeN>:<cost4-N>,..
        generate netlist using luts. Use the specified costs for luts with 1,
        2, 3, .. inputs.

    -sop
        map to sum-of-product cells and inverters

    -g type1,type2,...
        Map to the specified list of gate types. Supported gates types are:
           AND, NAND, OR, NOR, XOR, XNOR, ANDNOT, ORNOT, MUX,
           NMUX, AOI3, OAI3, AOI4, OAI4.
        (The NOT gate is always added to this list automatically.)

        The following aliases can be used to reference common sets of gate types:
          simple: AND OR XOR MUX
          cmos2:  NAND NOR
          cmos3:  NAND NOR AOI3 OAI3
          cmos4:  NAND NOR AOI3 OAI3 AOI4 OAI4
          cmos:   NAND NOR AOI3 OAI3 AOI4 OAI4 NMUX MUX XOR XNOR
          gates:  AND NAND OR NOR XOR XNOR ANDNOT ORNOT
          aig:    AND NAND OR NOR ANDNOT ORNOT

        The alias 'all' represent the full set of all gate types.

        Prefix a gate type with a '-' to remove it from the list. For example
        the arguments 'AND,OR,XOR' and 'simple,-MUX' are equivalent.

        The default is 'all,-NMUX,-AOI3,-OAI3,-AOI4,-OAI4'.

    -dff
        also pass $_DFF_?_ and $_DFFE_??_ cells through ABC. modules with many
        clock domains are automatically partitioned in clock domains and each
        domain is passed through ABC independently.

```

```
    -clk [!]<clock-signal-name>[,[!]<enable-signal-name>]
        use only the specified clock domain. this is like -dff, but only FF
        cells that belong to the specified clock domain are used.

    -keepff
        set the "keep" attribute on flip-flop output wires. (and thus preserve
        them, for example for equivalence checking.)

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

    -showtmp
        print the temp dir name in log. usually this is suppressed so that the
        command output is identical across runs.

    -markgroups
        set a 'abcgroup' attribute on all objects created by ABC. The value of
        this attribute is a unique integer for each ABC process started. This
        is useful for debugging the partitioning of clock domains.

    -dress
        run the 'dress' command after all other ABC commands. This aims to
        preserve naming by an equivalence check between the original and post-ABC
        netlists (experimental).

When no target cell library is specified the Yosys standard cell library is
loaded into ABC before the ABC script is executed.

Note that this is a logic optimization pass within Yosys that is calling ABC
internally. This is not going to "run ABC on your design". It will instead run
ABC on logic snippets extracted from your design. You will not get any useful
output when passing an ABC script that writes a file. Instead write your full
design as BLIF file with write_blif and then load that into ABC externally if
you want to use ABC to convert your design into another format.

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

C.2 abc9 – use ABC9 for technology mapping

```
    abc9 [options] [selection]

This script pass performs a sequence of commands to facilitate the use of the ABC
tool [1] for technology mapping of the current design to a target FPGA
architecture. Only fully-selected modules are supported.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -exe <command>
```

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

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

        if <file> starts with a plus sign (+), then the rest of the filename
        string is interpreted as the command string to be passed to ABC. The
        leading plus sign is removed and all commas (,) in the string are
        replaced with blanks before the string is passed to ABC.

        if no -script parameter is given, the following scripts are used:
          &scorr; &sweep; &dc2; &dch -f; &ps; &if {C} {W} {D} {R} -v; &mfs

    -fast
        use different default scripts that are slightly faster (at the cost
        of output quality):
          &if {C} {W} {D} {R} -v

    -D <picoseconds>
        set delay target. the string {D} in the default scripts above is
        replaced by this option when used, and an empty string otherwise
        (indicating best possible delay).

    -lut <width>
        generate netlist using luts of (max) the specified width.

    -lut <w1>:<w2>
        generate netlist using luts of (max) the specified width <w2>. All
        luts with width <= <w1> have constant cost. for luts larger than <w1>
        the area cost doubles with each additional input bit. the delay cost
        is still constant for all lut widths.

    -lut <file>
        pass this file with lut library to ABC.

    -luts <cost1>,<cost2>,<cost3>,<sizeN>:<cost4-N>,..
        generate netlist using luts. Use the specified costs for luts with 1,
        2, 3, .. inputs.

    -maxlut <width>
        when auto-generating the lut library, discard all luts equal to or
        greater than this size (applicable when neither -lut nor -luts is
        specified).

    -dff
        also pass $_DFF_[NP]_ cells through to ABC. modules with many clock
        domains are supported and automatically partitioned by ABC.

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

    -showtmp
```

```
        print the temp dir name in log. usually this is suppressed so that the
        command output is identical across runs.

    -box <file>
        pass this file with box library to ABC.

Note that this is a logic optimization pass within Yosys that is calling ABC
internally. This is not going to "run ABC on your design". It will instead run
ABC on logic snippets extracted from your design. You will not get any useful
output when passing an ABC script that writes a file. Instead write your full
design as an XAIGER file with ‘write_xaiger’ and then load that into ABC
externally if you want to use ABC to convert your design into another format.

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


    check:
        abc9_ops -check [-dff]    (option if -dff)

    map:
        abc9_ops -prep_hier [-dff]    (option if -dff)
        scc -specify -set_attr abc9_scc_id {}
        abc9_ops -prep_bypass [-prep_dff]    (option if -dff)
        design -stash $abc9
        design -load $abc9_map
        proc
        wbflip
        techmap -wb -map %$abc9 -map +/techmap.v A:abc9_flop
        opt -nodffe -nosdff
        abc9_ops -prep_dff_submod                                               (only if -dff)
        setattr -set submod "$abc9_flop" t:$_DFF_?_ %ci* %co* t:$_DFF_?_ %d           (only if -dff)
        submod                                                                  (only if -dff)
        setattr -mod -set whitebox 1 -set abc9_flop 1 -set abc9_box 1 *_$abc9_flop    (only if -dff)
        foreach module in design
            rename <module-name>_$abc9_flop _TECHMAP_REPLACE_                   (only if -dff)
        abc9_ops -prep_dff_unmap                                                (only if -dff)
        design -copy-to $abc9 =*_$abc9_flop                                     (only if -dff)
        delete =*_$abc9_flop                                                    (only if -dff)
        design -stash $abc9_map
        design -load $abc9
        design -delete $abc9
        techmap -wb -max_iter 1 -map %$abc9_map -map +/abc9_map.v [-D DFF]    (option if -dff)
        design -delete $abc9_map

    pre:
        read_verilog -icells -lib -specify +/abc9_model.v
        abc9_ops -break_scc -prep_delays -prep_xaiger [-dff]    (option for -dff)
        abc9_ops -prep_lut <maxlut>    (skip if -lut or -luts)
        abc9_ops -prep_box    (skip if -box)
        design -stash $abc9
        design -load $abc9_holes
        techmap -wb -map %$abc9 -map +/techmap.v
        opt -purge
        aigmap
```

```
        design -stash $abc9_holes
        design -load $abc9
        design -delete $abc9

    exe:
        aigmap
        foreach module in selection
            abc9_ops -write_lut <abc-temp-dir>/input.lut    (skip if '-lut' or '-luts')
            abc9_ops -write_box <abc-temp-dir>/input.box    (skip if '-box')
            write_xaiger -map <abc-temp-dir>/input.sym [-dff] <abc-temp-dir>/input.xaig
            abc9_exe [options] -cwd <abc-temp-dir> -lut [<abc-temp-dir>/input.lut] -box [<abc-temp-dir>/inp
            read_aiger -xaiger -wideports -module_name <module-name>$abc9 -map <abc-temp-dir>/input.sym <ab
            abc9_ops -reintegrate [-dff]

    unmap:
        techmap -wb -map %$abc9_unmap -map +/abc9_unmap.v
        design -delete $abc9_unmap
        design -delete $abc9_holes
        delete =*_$abc9_byp
        setattr -mod -unset abc9_box_id
```

C.3 abc9_exe – use ABC9 for technology mapping

```
    abc9_exe [options]


This pass uses the ABC tool [1] for technology mapping of the top module
(according to the (* top *) attribute or if only one module is currently selected)
to a target FPGA architecture.

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

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

        if <file> starts with a plus sign (+), then the rest of the filename
        string is interpreted as the command string to be passed to ABC. The
        leading plus sign is removed and all commas (,) in the string are
        replaced with blanks before the string is passed to ABC.

        if no -script parameter is given, the following scripts are used:
          &scorr; &sweep; &dc2; &dch -f; &ps; &if {C} {W} {D} {R} -v; &mfs

    -fast
        use different default scripts that are slightly faster (at the cost
        of output quality):
          &if {C} {W} {D} {R} -v

    -D <picoseconds>
        set delay target. the string {D} in the default scripts above is
```

```
        replaced by this option when used, and an empty string otherwise
        (indicating best possible delay).

    -lut <width>
        generate netlist using luts of (max) the specified width.

    -lut <w1>:<w2>
        generate netlist using luts of (max) the specified width <w2>. All
        luts with width <= <w1> have constant cost. for luts larger than <w1>
        the area cost doubles with each additional input bit. the delay cost
        is still constant for all lut widths.

    -lut <file>
        pass this file with lut library to ABC.

    -luts <cost1>,<cost2>,<cost3>,<sizeN>:<cost4-N>,..
        generate netlist using luts. Use the specified costs for luts with 1,
        2, 3, .. inputs.

    -showtmp
        print the temp dir name in log. usually this is suppressed so that the
        command output is identical across runs.

    -box <file>
        pass this file with box library to ABC.

    -cwd <dir>
        use this as the current working directory, inside which the 'input.xaig'
        file is expected. temporary files will be created in this directory, and
        the mapped result will be written to 'output.aig'.

Note that this is a logic optimization pass within Yosys that is calling ABC
internally. This is not going to "run ABC on your design". It will instead run
ABC on logic snippets extracted from your design. You will not get any useful
output when passing an ABC script that writes a file. Instead write your full
design as BLIF file with write_blif and then load that into ABC externally if
you want to use ABC to convert your design into another format.

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

C.4 abc9_ops – helper functions for ABC9

```
    abc9_ops [options] [selection]

This pass contains a set of supporting operations for use during ABC technology
mapping, and is expected to be called in conjunction with other operations from
the 'abc9' script pass. Only fully-selected modules are supported.

    -check
        check that the design is valid, e.g. (* abc9_box_id *) values are unique,
        (* abc9_carry *) is only given for one input/output port, etc.

```

```
    -prep_hier
        derive all used (* abc9_box *) or (* abc9_flop *) (if -dff option)
        whitebox modules. with (* abc9_flop *) modules, only those containing
        $dff/$_DFF_[NP]_ cells with zero initial state -- due to an ABC limitation
        -- will be derived.

    -prep_bypass
        create techmap rules in the '$abc9_map' and '$abc9_unmap' designs for
        bypassing sequential (* abc9_box *) modules using a combinatorial box
        (named *_$abc9_byp). bypassing is necessary if sequential elements (e.g.
        $dff, $mem, etc.) are discovered inside so that any combinatorial paths
        will be correctly captured. this bypass box will only contain ports that
        are referenced by a simple path declaration ($specify2 cell) inside a
        specify block.

    -prep_dff
        select all (* abc9_flop *) modules instantiated in the design and store
        in the named selection '$abc9_flops'.

    -prep_dff_submod
        within (* abc9_flop *) modules, rewrite all edge-sensitive path
        declarations and $setup() timing checks ($specify3 and $specrule cells)
        that share a 'DST' port with the $_DFF_[NP]_.Q port from this 'Q' port to
        the DFF's 'D' port. this is to prepare such specify cells to be moved
        into the flop box.

    -prep_dff_unmap
        populate the '$abc9_unmap' design with techmap rules for mapping *_$abc9_flop
        cells back into their derived cell types (where the rules created by
        -prep_hier will then map back to the original cell with parameters).

    -prep_delays
        insert `$__ABC9_DELAY' blackbox cells into the design to account for
        certain required times.

    -break_scc
        for an arbitrarily chosen cell in each unique SCC of each selected module
        (tagged with an (* abc9_scc_id = <int> *) attribute) interrupt all wires
        driven by this cell's outputs with a temporary $__ABC9_SCC_BREAKER cell
        to break the SCC.

    -prep_xaiger
        prepare the design for XAIGER output. this includes computing the
        topological ordering of ABC9 boxes, as well as preparing the '$abc9_holes'
        design that contains the logic behaviour of ABC9 whiteboxes.

    -dff
        consider flop cells (those instantiating modules marked with (* abc9_flop *))
        during -prep_{delays,xaiger,box}.

    -prep_lut <maxlut>
        pre-compute the lut library by analysing all modules marked with
        (* abc9_lut=<area> *).

```

```
    -write_lut <dst>
        write the pre-computed lut library to <dst>.

    -prep_box
        pre-compute the box library by analysing all modules marked with
        (* abc9_box *).

    -write_box <dst>
        write the pre-computed box library to <dst>.

    -reintegrate
        for each selected module, re-intergrate the module '<module-name>$abc9'
        by first recovering ABC9 boxes, and then stitching in the remaining primary
        inputs and outputs.
```

C.5 add – add objects to the design

```
    add <command> [selection]

This command adds objects to the design. It operates on all fully selected
modules. So e.g. 'add -wire foo' will add a wire foo to all selected modules.


    add {-wire|-input|-inout|-output} <name> <width> [selection]

Add a wire (input, inout, output port) with the given name and width. The
command will fail if the object exists already and has different properties
than the object to be created.


    add -global_input <name> <width> [selection]

Like 'add -input', but also connect the signal between instances of the
selected modules.


    add {-assert|-assume|-live|-fair|-cover} <name1> [-if <name2>]

Add an $assert, $assume, etc. cell connected to a wire named name1, with its
enable signal optionally connected to a wire named name2 (default: 1'b1).


    add -mod <name[s]>

Add module[s] with the specified name[s].
```

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

```
    aigmap [options] [selection]

```

```
Replace all logic cells with circuits made of only $_AND_ and
$_NOT_ cells.

    -nand
        Enable creation of $_NAND_ cells

    -select
        Overwrite replaced cells in the current selection with new $_AND_,
        $_NOT_, and $_NAND_, cells
```

C.7 alumacc – extract ALU and MACC cells

```
    alumacc [selection]

This pass translates arithmetic operations like $add, $mul, $lt, etc. to $alu
and $macc cells.
```

C.8 anlogic_eqn – Anlogic: Calculate equations for luts

```
    anlogic_eqn [selection]

Calculate equations for luts since bitstream generator depends on it.
```

C.9 anlogic_fixcarry – Anlogic: fix carry chain

```
    anlogic_fixcarry [options] [selection]

Add Anlogic adders to fix carry chain if needed.
```

C.10 assertpmux – adds asserts for parallel muxes

```
    assertpmux [options] [selection]

This command adds asserts to the design that assert that all parallel muxes
($pmux cells) have a maximum of one of their inputs enable at any time.

    -noinit
        do not enforce the pmux condition during the init state

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

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

```
    async2sync [options] [selection]

This command replaces async FF inputs with sync circuits emulating the same
behavior for when the async signals are actually synchronized to the clock.

This pass assumes negative hold time for the async FF inputs. For example when
a reset deasserts with the clock edge, then the FF output will still drive the
reset value in the next cycle regardless of the data-in value at the time of
the clock edge.
```

C.12 attrmap – renaming attributes

```
    attrmap [options] [selection]

This command renames attributes and/or maps key/value pairs to
other key/value pairs.

    -tocase <name>
        Match attribute names case-insensitively and set it to the specified
        name.

    -rename <old_name> <new_name>
        Rename attributes as specified

    -map <old_name>=<old_value> <new_name>=<new_value>
        Map key/value pairs as indicated.

    -imap <old_name>=<old_value> <new_name>=<new_value>
        Like -map, but use case-insensitive match for <old_value> when
        it is a string value.

    -remove <name>=<value>
        Remove attributes matching this pattern.

    -modattr
        Operate on module attributes instead of attributes on wires and cells.

For example, mapping Xilinx-style "keep" attributes to Yosys-style:

    attrmap -tocase keep -imap keep="true" keep=1 \
            -imap keep="false" keep=0 -remove keep=0
```

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

```
    attrmvcp [options] [selection]

```

```
Move or copy attributes on wires to the cells driving them.

    -copy
        By default, attributes are moved. This will only add
        the attribute to the cell, without removing it from
        the wire.

    -purge
        If no selected cell consumes the attribute, then it is
        left on the wire by default. This option will cause the
        attribute to be removed from the wire, even if no selected
        cell takes it.

    -driven
        By default, attriburtes are moved to the cell driving the
        wire. With this option set it will be moved to the cell
        driven by the wire instead.

    -attr <attrname>
        Move or copy this attribute. This option can be used
        multiple times.
```

C.14 autoname – automatically assign names to objects

```
    autoname [selection]

Assign auto-generated public names to objects with private names (the ones
with $-prefix).
```

C.15 blackbox – convert modules into blackbox modules

```
    blackbox [options] [selection]

Convert modules into blackbox modules (remove contents and set the blackbox
module attribute).
```

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

```
    bmuxmap [selection]

This pass transforms $bmux cells to trees of $mux cells.
```

C.17 bugpoint – minimize testcases

```
    bugpoint [options] [-script <filename> | -command "<command>"]

This command minimizes the current design that is known to crash Yosys with the
given script into a smaller testcase. It does this by removing an arbitrary part
of the design and recursively invokes a new Yosys process with this modified design
and the same script, repeating these steps while it can find a smaller design that
still causes a crash. Once this command finishes, it replaces the current design
with the smallest testcase it was able to produce.
In order to save the reduced testcase you must write this out to a file with
another command after `bugpoint` like `write_rtlil` or `write_verilog`.

    -script <filename> | -command "<command>"
        use this script file or command to crash Yosys. required.

    -yosys <filename>
        use this Yosys binary. if not specified, `yosys` is used.

    -grep "<string>"
        only consider crashes that place this string in the log file.

    -fast
        run `proc_clean; clean -purge` after each minimization step. converges
        faster, but produces larger testcases, and may fail to produce any
        testcase at all if the crash is related to dangling wires.

    -clean
        run `proc_clean; clean -purge` before checking testcase and after
        finishing. produces smaller and more useful testcases, but may fail to
        produce any testcase at all if the crash is related to dangling wires.

It is possible to constrain which parts of the design will be considered for
removal. Unless one or more of the following options are specified, all parts
will be considered.

    -modules
        try to remove modules. modules with a (* bugpoint_keep *) attribute
        will be skipped.

    -ports
        try to remove module ports. ports with a (* bugpoint_keep *) attribute
        will be skipped (useful for clocks, resets, etc.)

    -cells
        try to remove cells. cells with a (* bugpoint_keep *) attribute will
        be skipped.

    -connections
        try to reconnect ports to 'x.

    -processes
        try to remove processes. processes with a (* bugpoint_keep *) attribute
        will be skipped.

    -assigns
```

```
        try to remove process assigns from cases.

    -updates
        try to remove process updates from syncs.

    -runner "<prefix>"
        child process wrapping command, e.g., "timeout 30", or valgrind.
```

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

```
    cd <modname>

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



    cd <cellname>

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

    cd ..

Remove trailing substrings that start with '.' in current module name until
the name of a module in the current design is generated, then switch to that
module. Otherwise clear the current selection.

    cd

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

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

```
    check [options] [selection]

This pass identifies the following problems in the current design:

  - combinatorial loops
  - two or more conflicting drivers for one wire
  - used wires that do not have a driver

Options:

    -noinit
        also check for wires which have the 'init' attribute set

    -initdrv
        also check for wires that have the 'init' attribute set and are not
        driven by an FF cell type

```

```
    -mapped
        also check for internal cells that have not been mapped to cells of the
        target architecture

    -allow-tbuf
        modify the -mapped behavior to still allow $_TBUF_ cells

    -assert
        produce a runtime error if any problems are found in the current design
```

C.20 chformal – change formal constraints of the design

```
    chformal [types] [mode] [options] [selection]

Make changes to the formal constraints of the design. The [types] options
the type of constraint to operate on. If none of the following options are given,
the command will operate on all constraint types:

    -assert       $assert cells, representing assert(...) constraints
    -assume       $assume cells, representing assume(...) constraints
    -live         $live cells, representing assert(s_eventually ...)
    -fair         $fair cells, representing assume(s_eventually ...)
    -cover        $cover cells, representing cover() statements

Exactly one of the following modes must be specified:

    -remove
        remove the cells and thus constraints from the design

    -early
        bypass FFs that only delay the activation of a constraint

    -delay <N>
        delay activation of the constraint by <N> clock cycles

    -skip <N>
        ignore activation of the constraint in the first <N> clock cycles

    -assert2assume
    -assume2assert
    -live2fair
    -fair2live
        change the roles of cells as indicated. these options can be combined
```

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

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

Re-evaluate the selected modules with new parameters. String values must be
passed in double quotes (").
```

```


    chparam -list [selection]

List the available parameters of the selected modules.
```

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

```
    chtype [options] [selection]

Change the types of cells in the design.

    -set <type>
        set the cell type to the given type

    -map <old_type> <new_type>
        change cells types that match <old_type> to <new_type>
```

C.23 clean – remove unused cells and wires

```
    clean [options] [selection]

This is identical to 'opt_clean', but less verbose.

When commands are separated using the ';;' token, this command will be executed
between the commands.

When commands are separated using the ';;;' token, this command will be executed
in -purge mode between the commands.
```

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

```
    clean_zerowidth [selection]

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

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

```
    clk2fflogic [options] [selection]

This command replaces clocked flip-flops with generic $ff cells that use the
```

```
implicit global clock. This is useful for formal verification of designs with
multiple clocks.
```

C.26 clkbufmap – insert clock buffers on clock networks

```
    clkbufmap [options] [selection]

Inserts clock buffers between nets connected to clock inputs and their drivers.

In the absence of any selection, all wires without the 'clkbuf_inhibit'
attribute will be considered for clock buffer insertion.
Alternatively, to consider all wires without the 'buffer_type' attribute set to
'none' or 'bufr' one would specify:
  'w:* a:buffer_type=none a:buffer_type=bufr %u %d'
as the selection.

    -buf <celltype> <portname_out>:<portname_in>
        Specifies the cell type to use for the clock buffers
        and its port names.  The first port will be connected to
        the clock network sinks, and the second will be connected
        to the actual clock source.

    -inpad <celltype> <portname_out>:<portname_in>
        If specified, a PAD cell of the given type is inserted on
        clock nets that are also top module's inputs (in addition
        to the clock buffer, if any).

At least one of -buf or -inpad should be specified.
```

C.27 connect – create or remove connections

```
    connect [-nomap] [-nounset] -set <lhs-expr> <rhs-expr>

Create a connection. This is equivalent to adding the statement 'assign
<lhs-expr> = <rhs-expr>;' to the Verilog input. Per default, all existing
drivers for <lhs-expr> are unconnected. This can be overwritten by using
the -nounset option.



    connect [-nomap] -unset <expr>

Unconnect all existing drivers for the specified expression.



    connect [-nomap] [-assert] -port <cell> <port> <expr>

Connect the specified cell port to the specified cell port.



Per default signal alias names are resolved and all signal names are mapped
```

```
the the signal name of the primary driver. Using the -nomap option deactivates
this behavior.

The connect command operates in one module only. Either only one module must
be selected or an active module must be set using the 'cd' command.

The -assert option verifies that the connection already exists, instead of
making it.

This command does not operate on module with processes.
```

C.28 connect_rpc – connect to RPC frontend

```
    connect_rpc -exec <command> [args...]
    connect_rpc -path <path>

Load modules using an out-of-process frontend.

    -exec <command> [args...]
        run <command> with arguments [args...]. send requests on stdin, read
        responses from stdout.

    -path <path>
        connect to Unix domain socket at <path>. (Unix)
        connect to bidirectional byte-type named pipe at <path>. (Windows)

A simple JSON-based, newline-delimited protocol is used for communicating with
the frontend. Yosys requests data from the frontend by sending exactly 1 line
of JSON. Frontend responds with data or error message by replying with exactly
1 line of JSON as well.

    -> {"method": "modules"}
    <- {"modules": ["<module-name>", ...]}
    <- {"error": "<error-message>"}
        request for the list of modules that can be derived by this frontend.
        the 'hierarchy' command will call back into this frontend if a cell
        with type <module-name> is instantiated in the design.

    -> {"method": "derive", "module": "<module-name>", "parameters": {
        "<param-name>": {"type": "[unsigned|signed|string|real]",
                         "value": "<param-value>"}, ...}}
    <- {"frontend": "[rtlil|verilog|...]","source": "<source>"}}
    <- {"error": "<error-message>"}
        request for the module <module-name> to be derived for a specific set of
        parameters. <param-name> starts with \ for named parameters, and with $
        for unnamed parameters, which are numbered starting at 1.<param-value>
        for integer parameters is always specified as a binary string of unlimited
        precision. the <source> returned by the frontend is hygienically parsed
        by a built-in Yosys <frontend>, allowing the RPC frontend to return any
        convenient representation of the module. the derived module is cached,
        so the response should be the same whenever the same set of parameters
        is provided.
```

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

```
    connwrappers [options] [selection]

Wrappers are used in coarse-grain synthesis to wrap cells with smaller ports
in wrapper cells with a (larger) constant port size. I.e. the upper bits
of the wrapper output are signed/unsigned bit extended. This command uses this
knowledge to rewire the inputs of the driven cells to match the output of
the driving cell.

    -signed <cell_type> <port_name> <width_param>
    -unsigned <cell_type> <port_name> <width_param>
        consider the specified signed/unsigned wrapper output

    -port <cell_type> <port_name> <width_param> <sign_param>
        use the specified parameter to decide if signed or unsigned

The options -signed, -unsigned, and -port can be specified multiple times.
```

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

```
    coolrunner2_fixup [options] [selection]

Insert necessary buffer cells for CoolRunner-II architecture.
```

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

```
    coolrunner2_sop [options] [selection]

Break $sop cells into ANDTERM/ORTERM cells.
```

C.32 copy – copy modules in the design

```
    copy old_name new_name

Copy the specified module. Note that selection patterns are not supported
by this command.
```

C.33 cover – print code coverage counters

```
    cover [options] [pattern]

Print the code coverage counters collected using the cover() macro in the Yosys
C++ code. This is useful to figure out what parts of Yosys are utilized by a
test bench.

    -q
        Do not print output to the normal destination (console and/or log file)

    -o file
        Write output to this file, truncate if exists.

    -a file
        Write output to this file, append if exists.

    -d dir
        Write output to a newly created file in the specified directory.

When one or more pattern (shell wildcards) are specified, then only counters
matching at least one pattern are printed.


It is also possible to instruct Yosys to print the coverage counters on program
exit to a file using environment variables:

    YOSYS_COVER_DIR="{dir-name}" yosys {args}

        This will create a file (with an auto-generated name) in this
        directory and write the coverage counters to it.

    YOSYS_COVER_FILE="{file-name}" yosys {args}

        This will append the coverage counters to the specified file.


Hint: Use the following AWK command to consolidate Yosys coverage files:

    gawk '{ p[$3] = $1; c[$3] += $2; } END { for (i in p)
      printf "%-60s %10d %s\n", p[i], c[i], i; }' {files} | sort -k3


Coverage counters are only available in Yosys for Linux.
```

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

```
    cutpoint [options] [selection]

This command adds formal cut points to the design.

    -undef
        set cupoint nets to undef (x). the default behavior is to create a
```

```
        $anyseq cell and drive the cutpoint net from that
```

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

```
    debug cmd

Execute the specified command with debug log messages enabled
```

C.36 delete – delete objects in the design

```
    delete [selection]

Deletes the selected objects. This will also remove entire modules, if the
whole module is selected.


    delete {-input|-output|-port} [selection]

Does not delete any object but removes the input and/or output flag on the
selected wires, thus 'deleting' module ports.
```

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

```
    deminout [options] [selection]

"Demote" inout ports to input or output ports, if possible.
```

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

```
    demuxmap [selection]

This pass transforms $demux cells to a bunch of equality comparisons.
```

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

```
    design -reset

Clear the current design.


    design -save <name>

```

```
Save the current design under the given name.


    design -stash <name>

Save the current design under the given name and then clear the current design.


    design -push

Push the current design to the stack and then clear the current design.


    design -push-copy

Push the current design to the stack without clearing the current design.


    design -pop

Reset the current design and pop the last design from the stack.


    design -load <name>

Reset the current design and load the design previously saved under the given
name.


    design -copy-from <name> [-as <new_mod_name>] <selection>

Copy modules from the specified design into the current one. The selection is
evaluated in the other design.


    design -copy-to <name> [-as <new_mod_name>] [selection]

Copy modules from the current design into the specified one.


    design -import <name> [-as <new_top_name>] [selection]

Import the specified design into the current design. The source design must
either have a selected top module or the selection must contain exactly one
module that is then used as top module for this command.


    design -reset-vlog

The Verilog front-end remembers defined macros and top-level declarations
between calls to 'read_verilog'. This command resets this memory.

    design -delete <name>

```

```
Delete the design previously saved under the given name.
```

C.40 dffinit – set INIT param on FF cells

```
    dffinit [options] [selection]

This pass sets an FF cell parameter to the the initial value of the net it
drives. (This is primarily used in FPGA flows.)

    -ff <cell_name> <output_port> <init_param>
        operate on the specified cell type. this option can be used
        multiple times.

    -highlow
        use the string values "high" and "low" to represent a single-bit
        initial value of 1 or 0. (multi-bit values are not supported in this
        mode.)

    -strinit <string for high> <string for low>
        use string values in the command line to represent a single-bit
        initial value of 1 or 0. (multi-bit values are not supported in this
        mode.)

    -noreinit
        fail if the FF cell has already a defined initial value set in other
        passes and the initial value of the net it drives is not equal to
        the already defined initial value.
```

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

```
    dfflegalize [options] [selection]

Converts FFs to types supported by the target.

    -cell <cell_type_pattern> <init_values>
        specifies a supported group of FF cells.  <cell_type_pattern>
        is a yosys internal fine cell name, where ? characters can be
        as a wildcard matching any character.  <init_values> specifies
        which initialization values these FF cells can support, and can
        be one of:

        - 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
- 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 dffis available), an error is
```

C.42 dfflibmap – technology mapping of flip-flops

```
    dfflibmap [-prepare] [-map-only] [-info] -liberty <file> [selection]

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

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

```

```
When called with -prepare, this command will convert the internal FF cells
to the internal cell types that best match the cells found in the given
liberty file, but won't actually map them to the target cells.

When called with -map-only, this command will only map internal cell
types that are already of exactly the right type to match the target
cells, leaving remaining internal cells untouched.

When called with -info, this command will only print the target cell
list, along with their associated internal cell types, and the argumentsthat would be passed to the dfflega
The design will not be
changed.
```

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

```
    dffunmap [options] [selection]

This pass transforms FF types with clock enable and/or synchronous reset into
their base type (with neither clock enable nor sync reset) by emulating the clock
enable and synchronous reset with multiplexers on the cell input.

    -ce-only
        unmap only clock enables, leave synchronous resets alone.

    -srst-only
        unmap only synchronous resets, leave clock enables alone.
```

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

```
    dump [options] [selection]

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

    -m
        also dump the module headers, even if only parts of a single
        module is selected

    -n
        only dump the module headers if the entire module is selected

    -o <filename>
        write to the specified file.

    -a <filename>
        like -outfile but append instead of overwrite
```

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

```
    echo on

Print all commands to log before executing them.


    echo off

Do not print all commands to log before executing them. (default)
```

C.46 ecp5_gsr – ECP5: handle GSR

```
    ecp5_gsr [options] [selection]

Trim active low async resets connected to GSR and resolve GSR parameter,
if a GSR or SGSR primitive is used in the design.

If any cell has the GSR parameter set to "AUTO", this will be resolved
to "ENABLED" if a GSR primitive is present and the (* nogsr *) attribute
is not set, otherwise it will be resolved to "DISABLED".
```

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

```
    edgetypes [options] [selection]

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

C.48 efinix_fixcarry – Efinix: fix carry chain

```
    efinix_fixcarry [options] [selection]

Add Efinix adders to fix carry chain if needed.
```

C.49 equiv_add – add a $equiv cell

```
    equiv_add [-try] gold_sig gate_sig

This command adds an $equiv cell for the specified signals.


    equiv_add [-try] -cell gold_cell gate_cell

This command adds $equiv cells for the ports of the specified cells.
```

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

```
    equiv_induct [options] [selection]

Uses a version of temporal induction to prove $equiv cells.

Only selected $equiv cells are proven and only selected cells are used to
perform the proof.

    -undef
        enable modelling of undef states

    -seq <N>
        the max. number of time steps to be considered (default = 4)

This command is very effective in proving complex sequential circuits, when
the internal state of the circuit quickly propagates to $equiv cells.

However, this command uses a weak definition of 'equivalence': This command
proves that the two circuits will not diverge after they produce equal
outputs (observable points via $equiv) for at least <N> cycles (the <N>
specified via -seq).

Combined with simulation this is very powerful because simulation can give
you confidence that the circuits start out synced for at least <N> cycles
after reset.
```

C.51 equiv_make – prepare a circuit for equivalence checking

```
    equiv_make [options] gold_module gate_module equiv_module

This creates a module annotated with $equiv cells from two presumably
equivalent modules. Use commands such as 'equiv_simple' and 'equiv_status'
to work with the created equivalent checking module.

    -inames
        Also match cells and wires with $... names.

    -blacklist <file>
        Do not match cells or signals that match the names in the file.

    -encfile <file>
        Match FSM encodings using the description from the file.
        See 'help fsm_recode' for details.

Note: The circuit created by this command is not a miter (with something like
a trigger output), but instead uses $equiv cells to encode the equivalence
checking problem. Use 'miter -equiv' if you want to create a miter circuit.
```

C.52 equiv_mark – mark equivalence checking regions

```
    equiv_mark [options] [selection]

This command marks the regions in an equivalence checking module. Region 0 is
the proven part of the circuit. Regions with higher numbers are connected
unproven subcricuits. The integer attribute 'equiv_region' is set on all
wires and cells.
```

C.53 equiv_miter – extract miter from equiv circuit

```
    equiv_miter [options] miter_module [selection]

This creates a miter module for further analysis of the selected $equiv cells.

    -trigger
        Create a trigger output

    -cmp
        Create cmp_* outputs for individual unproven $equiv cells

    -assert
        Create a $assert cell for each unproven $equiv cell

    -undef
        Create compare logic that handles undefs correctly
```

C.54 equiv_opt – prove equivalence for optimized circuit

```
    equiv_opt [options] [command]

This command uses temporal induction to check circuit equivalence before and
after an optimization pass.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to the start of the command list, and empty to
        label is synonymous to the end of the command list.

    -map <filename>
        expand the modules in this file before proving equivalence. this is
        useful for handling architecture-specific primitives.

    -blacklist <file>
        Do not match cells or signals that match the names in the file
        (passed to equiv_make).

    -assert
        produce an error if the circuits are not equivalent.
```

```

    -multiclock
        run clk2fflogic before equivalence checking.

    -async2sync
        run async2sync before equivalence checking.

    -undef
        enable modelling of undef states during equiv_induct.

The following commands are executed by this verification command:

    run_pass:
        hierarchy -auto-top
        design -save preopt
        [command]
        design -stash postopt

    prepare:
        design -copy-from preopt  -as gold A:top
        design -copy-from postopt -as gate A:top

    techmap:    (only with -map)
        techmap -wb -D EQUIV -autoproc -map <filename> ...

    prove:
        clk2fflogic    (only with -multiclock)
        async2sync     (only with -async2sync)
        equiv_make -blacklist <filename> ... gold gate equiv
        equiv_induct [-undef] equiv
        equiv_status [-assert] equiv

    restore:
        design -load preopt
```

C.55 equiv_purge – purge equivalence checking module

```
    equiv_purge [options] [selection]

This command removes the proven part of an equivalence checking module, leaving
only the unproven segments in the design. This will also remove and add module
ports as needed.
```

C.56 equiv_remove – remove $equiv cells

```
    equiv_remove [options] [selection]

This command removes the selected $equiv cells. If neither -gold nor -gate is
used then only proven cells are removed.

```

```
    -gold
        keep gold circuit

    -gate
        keep gate circuit
```

C.57 equiv_simple – try proving simple $equiv instances

```
    equiv_simple [options] [selection]

This command tries to prove $equiv cells using a simple direct SAT approach.

    -v
        verbose output

    -undef
        enable modelling of undef states

    -short
        create shorter input cones that stop at shared nodes. This yields
        simpler SAT problems but sometimes fails to prove equivalence.

    -nogroup
        disabling grouping of $equiv cells by output wire

    -seq <N>
        the max. number of time steps to be considered (default = 1)
```

C.58 equiv_status – print status of equivalent checking module

```
    equiv_status [options] [selection]

This command prints status information for all selected $equiv cells.

    -assert
        produce an error if any unproven $equiv cell is found
```

C.59 equiv_struct – structural equivalence checking

```
    equiv_struct [options] [selection]

This command adds additional $equiv cells based on the assumption that the
gold and gate circuit are structurally equivalent. Note that this can introduce
bad $equiv cells in cases where the netlists are not structurally equivalent,
for example when analyzing circuits with cells with commutative inputs. This
command will also de-duplicate gates.

```

```
    -fwd
        by default this command performans forward sweeps until nothing can
        be merged by forwards sweeps, then backward sweeps until forward
        sweeps are effective again. with this option set only forward sweeps
        are performed.

    -fwonly <cell_type>
        add the specified cell type to the list of cell types that are only
        merged in forward sweeps and never in backward sweeps. $equiv is in
        this list automatically.

    -icells
        by default, the internal RTL and gate cell types are ignored. add
        this option to also process those cell types with this command.

    -maxiter <N>
        maximum number of iterations to run before aborting
```

C.60 eval – evaluate the circuit given an input

```
    eval [options] [selection]

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

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

    -set-undef
        set all unspecified source signals to undef (x)

    -table <signal>
        create a truth table using the specified input signals

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

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

```
    exec [options] -- [command]

Execute a command in the operating system shell.  All supplied arguments are
concatenated and passed as a command to popen(3).  Whitespace is not guaranteed
to be preserved, even if quoted.  stdin and stderr are not connected, while stdout is
logged unless the "-q" option is specified.



    -q
        Suppress stdout and stderr from subprocess
```

```

    -expect-return <int>
        Generate an error if popen() does not return specified value.
        May only be specified once; the final specified value is controlling
        if specified multiple times.

    -expect-stdout <regex>
        Generate an error if the specified regex does not match any line
        in subprocess's stdout.  May be specified multiple times.

    -not-expect-stdout <regex>
        Generate an error if the specified regex matches any line
        in subprocess's stdout.  May be specified multiple times.


    Example: exec -q -expect-return 0 -- echo "bananapie" | grep "nana"
```

C.62 expose – convert internal signals to module ports

```
    expose [options] [selection]

This command exposes all selected internal signals of a module as additional
outputs.

    -dff
        only consider wires that are directly driven by register cell.

    -cut
        when exposing a wire, create an input/output pair and cut the internal
        signal path at that wire.

    -input
        when exposing a wire, create an input port and disconnect the internal
        driver.

    -shared
        only expose those signals that are shared among the selected modules.
        this is useful for preparing modules for equivalence checking.

    -evert
        also turn connections to instances of other modules to additional
        inputs and outputs and remove the module instances.

    -evert-dff
        turn flip-flops to sets of inputs and outputs.

    -sep <separator>
        when creating new wire/port names, the original object name is suffixed
        with this separator (default: '.') and the port name or a type
        designator for the exposed signal.
```

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

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

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

    -map <map_file>
        use the modules in this file as reference. This option can be used
        multiple times.

    -map %<design-name>
        use the modules in this in-memory design as reference. This option can
        be used multiple times.

    -verbose
        print debug output while analyzing

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

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

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

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

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

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

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

    -ignore_parameters
        Do not use parameters when matching cells.

    -ignore_param <cell_type> <parameter_name>
        Do not use this parameter when matching cells.
```

```

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

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

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

    -mine_cells_span <min> <max>
        only mine for subcircuits with the specified number of cells
        default value: 3 5

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

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

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

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

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

C.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

This command finds chains of $_AND_, $_OR_, and $_XOR_ cells and replaces them
with their corresponding $reduce_* cells. Because this command only operates on
these cell types, it is recommended to map the design to only these cell types
using the ‘abc -g‘ command. Note that, in some cases, it may be more effective
to map the design to only $_AND_ cells, run extract_reduce, map the remaining
parts of the design to AND/OR/XOR cells, and run extract_reduce a second time.

    -allow-off-chain
        Allows matching of cells that have loads outside the chain. These cells
        will be replicated and folded into the $reduce_* cell, but the original
        cell will remain, driving its original loads.
```

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

```
    extractinv [options] [selection]

Searches the design for all cells with invertible pins controlled by a cell
parameter (eg. IS_CLK_INVERTED on many Xilinx cells) and removes the parameter.
```

```
If the parameter was set to 1, inserts an explicit inverter cell in front of
the pin instead.  Normally used for output to ISE, which does not support the
inversion parameters.

To mark a cell port as invertible, use (* invertible_pin = "param_name" *)
on the wire in the blackbox module.  The parameter value should have
the same width as the port, and will be effectively XORed with it.

    -inv <celltype> <portname_out>:<portname_in>
        Specifies the cell type to use for the inverters and its port names.
        This option is required.
```

C.68 flatten – flatten design

```
    flatten [options] [selection]

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

Cells and/or modules with the 'keep_hierarchy' attribute set will not be
flattened by this command.

    -wb
        Ignore the 'whitebox' attribute on cell implementations.
```

C.69 flowmap – pack LUTs with FlowMap

```
    flowmap [options] [selection]

This pass uses the FlowMap technology mapping algorithm to pack logic gates
into k-LUTs with optimal depth. It allows mapping any circuit elements that can
be evaluated with the `eval` pass, including cells with multiple output ports
and multi-bit input and output ports.

    -maxlut k
        perform technology mapping for a k-LUT architecture. if not specified,
        defaults to 3.

    -minlut n
        only produce n-input or larger LUTs. if not specified, defaults to 1.

    -cells <cell>[,<cell>,...]
        map specified cells. if not specified, maps $_NOT_, $_AND_, $_OR_,
        $_XOR_ and $_MUX_, which are the outputs of the `simplemap` pass.

    -relax
        perform depth relaxation and area minimization.

    -r-alpha n, -r-beta n, -r-gamma n
```

```
        parameters of depth relaxation heuristic potential function.
        if not specified, alpha=8, beta=2, gamma=1.

    -optarea n
        optimize for area by trading off at most n logic levels for fewer LUTs.
        n may be zero, to optimize for area without increasing depth.
        implies -relax.

    -debug
        dump intermediate graphs.

    -debug-relax
        explain decisions performed during depth relaxation.
```

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

```
    fmcombine [options] module_name gold_cell gate_cell

This pass takes two cells, which are instances of the same module, and replaces
them with one instance of a special 'combined' module, that effectively
contains two copies of the original module, plus some formal properties.

This is useful for formal test benches that check what differences in behavior
a slight difference in input causes in a module.


    -initeq
        Insert assumptions that initially all FFs in both circuits have the
        same initial values.

    -anyeq
        Do not duplicate $anyseq/$anyconst cells.

    -fwd
        Insert forward hint assumptions into the combined module.

    -bwd
        Insert backward hint assumptions into the combined module.
        (Backward hints are logically equivalend to fordward hits, but
        some solvers are faster with bwd hints, or even both -bwd and -fwd.)

    -nop
        Don't insert hint assumptions into the combined module.
        (This should not provide any speedup over the original design, but
        strangely sometimes it does.)

If none of -fwd, -bwd, and -nop is given, then -fwd is used as default.
```

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

```
    fminit [options] <selection>

This pass creates init constraints (for example for reset sequences) in a formal
model.

    -seq <signal> <sequence>
        Set sequence using comma-separated list of values, use 'z for
        unconstrained bits. The last value is used for the remainder of the
        trace.

    -set <signal> <value>
        Add constant value constraint

    -posedge <signal>
    -negedge <signal>
        Set clock for init sequences
```

C.72 freduce – perform functional reduction

```
    freduce [options] [selection]

This pass performs functional reduction in the circuit. I.e. if two nodes are
equivalent, they are merged to one node and one of the redundant drivers is
disconnected. A subsequent call to 'clean' will remove the redundant drivers.

    -v, -vv
        enable verbose or very verbose output

    -inv
        enable explicit handling of inverted signals

    -stop <n>
        stop after <n> reduction operations. this is mostly used for
        debugging the freduce command itself.

    -dump <prefix>
        dump the design to <prefix>_<module>_<num>.il after each reduction
        operation. this is mostly used for debugging the freduce command.

This pass is undef-aware, i.e. it considers don't-care values for detecting
equivalent nodes.

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

C.73 fsm – extract and optimize finite state machines

```
    fsm [options] [selection]

```

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

    fsm_detect          unless got option -nodetect
    fsm_extract

    fsm_opt
    opt_clean
    fsm_opt

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

    fsm_recode          unless got option -norecode

    fsm_info

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

Options:

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

    -fullexpand
        call expand with -full option

    -encoding type
    -fm_set_fsm_file file
    -encfile file
        passed through to fsm_recode pass
```

C.74 fsm_detect – finding FSMs in design

```
    fsm_detect [selection]

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

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

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

C.75 fsm_expand – expand FSM cells by merging logic into it

```
    fsm_expand [-full] [selection]

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

By default, fsm_expand is still a bit conservative regarding merging larger
word-wide cells. Call with -full to consider all cells for merging.
```

C.76 fsm_export – exporting FSMs to KISS2 files

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

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

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

    -o filename
        filename of the first exported FSM

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

C.77 fsm_extract – extracting FSMs in design

```
    fsm_extract [selection]

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

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

C.78 fsm_info – print information on finite state machines

```
    fsm_info [selection]

```

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

C.79 fsm_map – mapping FSMs to basic logic

```
    fsm_map [selection]

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

C.80 fsm_opt – optimize finite state machines

```
    fsm_opt [selection]

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

C.81 fsm_recode – recoding finite state machines

```
    fsm_recode [options] [selection]

This pass reassign the state encodings for FSM cells. At the moment only
one-hot encoding and binary encoding is supported.
    -encoding <type>
        specify the encoding scheme used for FSMs without the
        'fsm_encoding' attribute or with the attribute set to `auto'.

    -fm_set_fsm_file <file>
        generate a file containing the mapping from old to new FSM encoding
        in form of Synopsys Formality set_fsm_* commands.

    -encfile <file>
        write the mappings from old to new FSM encoding to a file in the
        following format:

            .fsm <module_name> <state_signal>
            .map <old_bitpattern> <new_bitpattern>
```

C.82 glift – create GLIFT models and optimization problems

```
    glift <command> [options] [selection]

Augments the current or specified module with gate-level information flow tracking
(GLIFT) logic using the "constructive mapping" approach. Also can set up QBF-SAT
```

```
optimization problems in order to optimize GLIFT models or trade off precision and
complexity.


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 `-create-instrumented-model`.
    (default: use a complex model and give that wire the "keep" and "minimize" attributes)

  -no-cost-model
    Do not model taint tracking logic area and do not create a `__glift_weight` wire.
    Only applicable in combination with `-create-instrumented-model`.
    (default: model area and give that wire the "keep" and "minimize" attributes)

  -instrument-more
    Allow choice from more versions of (even simpler) taint tracking logic. A total
    of 8 versions of taint tracking logic will be added per gate, including the 4
    versions from `-create-instrumented-model` and these additional versions:

      y_t = a_t
      y_t = b_t
      y_t = 1
      y_t = 0

    Only applicable in combination with `-create-instrumented-model`.
    (default: do not add more versions of taint tracking logic.
```

C.83 greenpak4_dffinv – merge greenpak4 inverters and DFF/latches

```
    greenpak4_dffinv [options] [selection]

Merge GP_INV cells with GP_DFF* and GP_DLATCH* cells.
```

C.84 help – display help messages

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

    help -cells ..........  list all cell types
    help <celltype>  .....  print help message for given cell type
    help <celltype>+  ....  print verilog code for given cell type
```

C.85 hierarchy – check, expand and clean up design hierarchy

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

In parametric designs, a module might exists in several variations with
different parameter values. This pass looks at all modules in the current
design and re-runs the language frontends for the parametric modules as
needed. It also resolves assignments to wired logic data types (wand/wor),
```

```
resolves positional module parameters, unrolls array instances, and more.

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

    -simcheck
        like -check, but also throw an error if blackbox modules are
        instantiated, and throw an error if the design has no top module.

    -purge_lib
        by default the hierarchy command will not remove library (blackbox)
        modules. use this option to also remove unused blackbox modules.

    -libdir <directory>
        search for files named <module_name>.v in the specified directory
        for unknown modules and automatically run read_verilog for each
        unknown module.

    -keep_positionals
        per default this pass also converts positional arguments in cells
        to arguments using port names. This option disables this behavior.

    -keep_portwidths
        per default this pass adjusts the port width on cells that are
        module instances when the width does not match the module port. This
        option disables this behavior.

    -nodefaults
        do not resolve input port default values

    -nokeep_asserts
        per default this pass sets the "keep" attribute on all modules
        that directly or indirectly contain one or more formal properties.
        This option disables this behavior.

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

        when the -top option is used, the 'top' attribute will be set on the
        specified top module. otherwise a module with the 'top' attribute set
        will implicitly be used as top module, if such a module exists.

    -auto-top
        automatically determine the top of the design hierarchy and mark it.

    -chparam name value
       elaborate the top module using this parameter value. Modules on which
       this parameter does not exist may cause a warning message to be output.
       This option can be specified multiple times to override multiple
       parameters. String values must be passed in double quotes (").

In -generate mode this pass generates blackbox modules for the given cell
```

```
types (wildcards supported). For this the design is searched for cells that
match the given types and then the given port declarations are used to
determine the direction of the ports. The syntax for a port declaration is:

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

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

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

C.86 hilomap – technology mapping of constant hi- and/or lo-drivers

```
    hilomap [options] [selection]

Map constants to 'tielo' and 'tiehi' driver cells.

    -hicell <celltype> <portname>
        Replace constant hi bits with this cell.

    -locell <celltype> <portname>
        Replace constant lo bits with this cell.

    -singleton
        Create only one hi/lo cell and connect all constant bits
        to that cell. Per default a separate cell is created for
        each constant bit.
```

C.87 history – show last interactive commands

```
    history

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

C.88 ice40_braminit – iCE40: perform SB_RAM40_4K initialization from file

```
    ice40_braminit

```

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

C.89 ice40_dsp – iCE40: map multipliers

```
    ice40_dsp [options] [selection]

Map multipliers ($mul/SB_MAC16) and multiply-accumulate ($mul/SB_MAC16 + $add)
cells into iCE40 DSP resources.
Currently, only the 16x16 multiply mode is supported and not the 2 x 8x8 mode.

Pack input registers (A, B, {C,D}; with optional hold), pipeline registers
({F,J,K,G}, H), output registers (O -- full 32-bits or lower 16-bits only; with
optional hold), and post-adder into into the SB_MAC16 resource.

Multiply-accumulate operations using the post-adder with feedback on the {C,D}
input will be folded into the DSP. In this scenario only, resetting the
the accumulator to an arbitrary value can be inferred to use the {C,D} input.
```

C.90 ice40_opt – iCE40: perform simple optimizations

```
    ice40_opt [options] [selection]

This command executes the following script:

    do
        <ice40 specific optimizations>
        opt_expr -mux_undef -undriven [-full]
        opt_merge
        opt_dff
        opt_clean
    while <changed design>
```

C.91 ice40_wrapcarry – iCE40: wrap carries

```
    ice40_wrapcarry [selection]

Wrap manually instantiated SB_CARRY cells, along with their associated SB_LUT4s,
into an internal $__ICE40_CARRY_WRAPPER cell for preservation across technology
mapping.

Attributes on both cells will have their names prefixed with 'SB_CARRY.' or
'SB_LUT4.' and attached to the wrapping cell.
A (* keep *) attribute on either cell will be logically OR-ed together.

    -unwrap
        unwrap $__ICE40_CARRY_WRAPPER cells back into SB_CARRYs and SB_LUT4s,
        including restoring their attributes.
```

C.92 insbuf – insert buffer cells for connected wires

```
    insbuf [options] [selection]

Insert buffer cells into the design for directly connected wires.

    -buf <celltype> <in-portname> <out-portname>
        Use the given cell type instead of $_BUF_. (Notice that the next
        call to "clean" will remove all $_BUF_ in the design.)
```

C.93 iopadmap – technology mapping of i/o pads (or buffers)

```
    iopadmap [options] [selection]

Map module inputs/outputs to PAD cells from a library. This pass
can only map to very simple PAD cells. Use 'techmap' to further map
the resulting cells to more sophisticated PAD cells.

    -inpad <celltype> <in_port>[:<ext_port>]
        Map module input ports to the given cell type with the
        given output port name. if a 2nd portname is given, the
        signal is passed through the pad cell, using the 2nd
        portname as the port facing the module port.

    -outpad <celltype> <out_port>[:<ext_port>]
    -inoutpad <celltype> <io_port>[:<ext_port>]
        Similar to -inpad, but for output and inout ports.

    -toutpad <celltype> <oe_port>:<out_port>[:<ext_port>]
        Merges $_TBUF_ cells into the output pad cell. This takes precedence
        over the other -outpad cell. The first portname is the enable input
        of the tristate driver, which can be prefixed with `~` for negative
        polarity enable.

    -tinoutpad <celltype> <oe_port>:<in_port>:<out_port>[:<ext_port>]
        Merges $_TBUF_ cells into the inout pad cell. This takes precedence
        over the other -inoutpad cell. The first portname is the enable input
        of the tristate driver and the 2nd portname is the internal output
        buffering the external signal.  Like with `-toutpad`, the enable can
        be marked as negative polarity by prefixing the name with `~`.

    -ignore <celltype> <portname>[:<portname>]*
        Skips mapping inputs/outputs that are already connected to given
        ports of the given cell.  Can be used multiple times.  This is in
        addition to the cells specified as mapping targets.

    -widthparam <param_name>
        Use the specified parameter name to set the port width.

    -nameparam <param_name>
        Use the specified parameter to set the port name.

```

```
    -bits
        create individual bit-wide buffers even for ports that
        are wider. (the default behavior is to create word-wide
        buffers using -widthparam to set the word size on the cell.)

Tristate PADS (-toutpad, -tinoutpad) always operate in -bits mode.
```

C.94 json – write design in JSON format

```
    json [options] [selection]

Write a JSON netlist of all selected objects.

    -o <filename>
        write to the specified file.

    -aig
        also include AIG models for the different gate types

    -compat-int
        emit 32-bit or smaller fully-defined parameter values directly
        as JSON numbers (for compatibility with old parsers)

See 'help write_json' for a description of the JSON format used.
```

C.95 log – print text and log files

```
    log string

Print the given string to the screen and/or the log file. This is useful for TCL
scripts, because the TCL command "puts" only goes to stdout but not to
logfiles.

    -stdout
        Print the output to stdout too. This is useful when all Yosys is executed
        with a script and the -q (quiet operation) argument to notify the user.

    -stderr
        Print the output to stderr too.

    -nolog
        Don't use the internal log() command. Use either -stdout or -stderr,
        otherwise no output will be generated at all.

    -n
        do not append a newline
```

C.96 logger – set logger properties

```
    logger [options]

This command sets global logger properties, also available using command line
options.

    -[no]time
        enable/disable display of timestamp in log output.

    -[no]stderr
        enable/disable logging errors to stderr.

    -warn regex
        print a warning for all log messages matching the regex.

    -nowarn regex
        if a warning message matches the regex, it is printed as regular
        message instead.

    -werror regex
        if a warning message matches the regex, it is printed as error
        message instead and the tool terminates with a nonzero return code.

    -[no]debug
        globally enable/disable debug log messages.

    -experimental <feature>
        do not print warnings for the specified experimental feature

    -expect <type> <regex> <expected_count>
        expect log, warning or error to appear. matched errors will terminate
        with exit code 0.

    -expect-no-warnings
        gives error in case there is at least one warning that is not expected.

    -check-expected
        verifies that the patterns previously set up by -expect have actually
        been met, then clears the expected log list.  If this is not called
        manually, the check will happen at yosys exist time instead.
```

C.97 ls – list modules or objects in modules

```
    ls [selection]

When no active module is selected, this prints a list of modules.

When an active module is selected, this prints a list of objects in the module.
```

C.98 ltp – print longest topological path

```
    ltp [options] [selection]

This command prints the longest topological path in the design. (Only considers
paths within a single module, so the design must be flattened.)

    -noff
        automatically exclude FF cell types
```

C.99 lut2mux – convert $lut to $_MUX_

```
    lut2mux [options] [selection]

This pass converts $lut cells to $_MUX_ gates.
```

C.100 maccmap – mapping macc cells

```
    maccmap [-unmap] [selection]

This pass maps $macc cells to yosys $fa and $alu cells. When the -unmap option
is used then the $macc cell is mapped to $add, $sub, etc. cells instead.
```

C.101 memory – translate memories to basic cells

```
    memory [-nomap] [-nordff] [-nowiden] [-nosat] [-memx] [-bram <bram_rules>] [selection]

This pass calls all the other memory_* passes in a useful order:

    opt_mem
    opt_mem_priority
    opt_mem_feedback
    memory_dff                          (skipped if called with -nordff or -memx)
    opt_clean
    memory_share [-nowiden] [-nosat]
    opt_mem_widen
    memory_memx                         (when called with -memx)
    opt_clean
    memory_collect
    memory_bram -rules <bram_rules>     (when called with -bram)
    memory_map                          (skipped if called with -nomap)

This converts memories to word-wide DFFs and address decoders
or multiport memory blocks if called with the -nomap option.
```

C.102 memory_bram – map memories to block rams

```
    memory_bram -rules <rule_file> [selection]

This pass converts the multi-port $mem memory cells into block ram instances.
The given rules file describes the available resources and how they should be
used.

The rules file contains configuration options, a set of block ram description
and a sequence of match rules.

The option 'attr_icase' configures how attribute values are matched. The value 0
means case-sensitive, 1 means case-insensitive.

A block ram description looks like this:

    bram RAMB1024X32     # name of BRAM cell
      init 1             # set to '1' if BRAM can be initialized
      abits 10           # number of address bits
      dbits 32           # number of data bits
      groups 2           # number of port groups
      ports  1 1         # number of ports in each group
      wrmode 1 0         # set to '1' if this groups is write ports
      enable 4 1         # number of enable bits
      transp 0 2         # transparent (for read ports)
      clocks 1 2         # clock configuration
      clkpol 2 2         # clock polarity configuration
    endbram

For the option 'transp' the value 0 means non-transparent, 1 means transparent
and a value greater than 1 means configurable. All groups with the same
value greater than 1 share the same configuration bit.

For the option 'clocks' the value 0 means non-clocked, and a value greater
than 0 means clocked. All groups with the same value share the same clock
signal.

For the option 'clkpol' the value 0 means negative edge, 1 means positive edge
and a value greater than 1 means configurable. All groups with the same value
greater than 1 share the same configuration bit.

Using the same bram name in different bram blocks will create different variants
of the bram. Verilog configuration parameters for the bram are created as needed.

It is also possible to create variants by repeating statements in the bram block
and appending '@<label>' to the individual statements.

A match rule looks like this:

    match RAMB1024X32
      max waste 16384    # only use this bram if <= 16k ram bits are unused
      min efficiency 80  # only use this bram if efficiency is at least 80%
    endmatch

```

```
It is possible to match against the following values with min/max rules:

    words  ........  number of words in memory in design
    abits  ........  number of address bits on memory in design
    dbits  ........  number of data bits on memory in design
    wports  .......  number of write ports on memory in design
    rports  .......  number of read ports on memory in design
    ports  ........  number of ports on memory in design
    bits  .........  number of bits in memory in design
    dups ..........  number of duplications for more read ports

    awaste  .......  number of unused address slots for this match
    dwaste  .......  number of unused data bits for this match
    bwaste  .......  number of unused bram bits for this match
    waste  ........  total number of unused bram bits (bwaste*dups)
    efficiency  ...  total percentage of used and non-duplicated bits

    acells  .......  number of cells in 'address-direction'
    dcells  .......  number of cells in 'data-direction'
    cells  ........  total number of cells (acells*dcells*dups)

A match containing the command 'attribute' followed by a list of space
separated 'name[=string_value]' values requires that the memory contains any
one of the given attribute name and string values (where specified), or name
and integer 1 value (if no string_value given, since Verilog will interpret
'(* attr *)' as '(* attr=1 *)').
A name prefixed with '!' indicates that the attribute must not exist.

The interface for the created bram instances is derived from the bram
description. Use 'techmap' to convert the created bram instances into
instances of the actual bram cells of your target architecture.

A match containing the command 'or_next_if_better' is only used if it
has a higher efficiency than the next match (and the one after that if
the next also has 'or_next_if_better' set, and so forth).

A match containing the command 'make_transp' will add external circuitry
to simulate 'transparent read', if necessary.

A match containing the command 'make_outreg' will add external flip-flops
to implement synchronous read ports, if necessary.

A match containing the command 'shuffle_enable A' will re-organize
the data bits to accommodate the enable pattern of port A.
```

C.103 memory_collect – creating multi-port memory cells

```
    memory_collect [selection]

This pass collects memories and memory ports and creates generic multiport
memory cells.
```

C.104 memory_dff – merge input/output DFFs into memory read ports

```
    memory_dff [options] [selection]

This pass detects DFFs at memory read ports and merges them into the memory port.
I.e. it consumes an asynchronous memory port and the flip-flops at its
interface and yields a synchronous memory port.
```

C.105 memory_map – translate multiport memories to basic cells

```
    memory_map [options] [selection]

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

    -attr !<name>
        do not map memories that have attribute <name> set.

    -attr <name>[=<value>]
        for memories that have attribute <name> set, only map them if its value
        is a string <value> (if specified), or an integer 1 (otherwise). if this
        option is specified multiple times, map the memory if the attribute is
        to any of the values.

    -iattr
        for -attr, ignore case of <value>.
```

C.106 memory_memx – emulate vlog sim behavior for mem ports

```
    memory_memx [selection]

This pass adds additional circuitry that emulates the Verilog simulation
behavior for out-of-bounds memory reads and writes.
```

C.107 memory_narrow – split up wide memory ports

```
    memory_narrow [options] [selection]

This pass splits up wide memory ports into several narrow ports.
```

C.108 memory_nordff – extract read port FFs from memories

```
    memory_nordff [options] [selection]

This pass extracts FFs from memory read ports. This results in a netlist
similar to what one would get from not calling memory_dff.
```

C.109 memory_share – consolidate memory ports

```
    memory_share [-nosat] [-nowiden] [selection]

This pass merges share-able memory ports into single memory ports.

The following methods are used to consolidate the number of memory ports:

  - When multiple write ports access the same address then this is converted
    to a single write port with a more complex data and/or enable logic path.

  - When multiple read or write ports access adjacent aligned addresses, they are
    merged to a single wide read or write port.  This transformation can be
    disabled with the "-nowiden" option.

  - When multiple write ports are never accessed at the same time (a SAT
    solver is used to determine this), then the ports are merged into a single
    write port.  This transformation can be disabled with the "-nosat" option.

Note that in addition to the algorithms implemented in this pass, the $memrd
and $memwr cells are also subject to generic resource sharing passes (and other
optimizations) such as "share" and "opt_merge".
```

C.110 memory_unpack – unpack multi-port memory cells

```
    memory_unpack [selection]

This pass converts the multi-port $mem memory cells into individual $memrd and
$memwr cells. It is the counterpart to the memory_collect pass.
```

C.111 miter – automatically create a miter circuit

```
    miter -equiv [options] gold_name gate_name miter_name

Creates a miter circuit for equivalence checking. The gold- and gate- modules
must have the same interfaces. The miter circuit will have all inputs of the
two source modules, prefixed with 'in_'. The miter circuit has a 'trigger'
output that goes high if an output mismatch between the two source modules is
detected.

```

```
    -ignore_gold_x
        a undef (x) bit in the gold module output will match any value in
        the gate module output.

    -make_outputs
        also route the gold- and gate-outputs to 'gold_*' and 'gate_*' outputs
        on the miter circuit.

    -make_outcmp
        also create a cmp_* output for each gold/gate output pair.

    -make_assert
        also create an 'assert' cell that checks if trigger is always low.

    -flatten
        call 'flatten -wb; opt_expr -keepdc -undriven;;' on the miter circuit.


    miter -assert [options] module [miter_name]

Creates a miter circuit for property checking. All input ports are kept,
output ports are discarded. An additional output 'trigger' is created that
goes high when an assert is violated. Without a miter_name, the existing
module is modified.

    -make_outputs
        keep module output ports.

    -flatten
        call 'flatten -wb; opt_expr -keepdc -undriven;;' on the miter circuit.
```

C.112 mutate – generate or apply design mutations

```
    mutate -list N [options] [selection]

Create a list of N mutations using an even sampling.

    -o filename
        Write list to this file instead of console output

    -s filename
        Write a list of all src tags found in the design to the specified file

    -seed N
        RNG seed for selecting mutations

    -none
        Include a "none" mutation in the output

    -ctrl name width value
        Add -ctrl options to the output. Use 'value' for first mutation, then
        simply count up from there.
```

```

    -mode name
    -module name
    -cell name
    -port name
    -portbit int
    -ctrlbit int
    -wire name
    -wirebit int
    -src string
        Filter list of mutation candidates to those matching
        the given parameters.

    -cfg option int
        Set a configuration option. Options available:
          weight_pq_w weight_pq_b weight_pq_c weight_pq_s
          weight_pq_mw weight_pq_mb weight_pq_mc weight_pq_ms
          weight_cover pick_cover_prcnt


    mutate -mode MODE [options]

Apply the given mutation.

    -ctrl name width value
        Add a control signal with the given name and width. The mutation is
        activated if the control signal equals the given value.

    -module name
    -cell name
    -port name
    -portbit int
    -ctrlbit int
        Mutation parameters, as generated by 'mutate -list N'.

    -wire name
    -wirebit int
    -src string
        Ignored. (They are generated by -list for documentation purposes.)
```

C.113 muxcover – cover trees of MUX cells with wider MUXes

```
    muxcover [options] [selection]

Cover trees of $_MUX_ cells with $_MUX{4,8,16}_ cells

    -mux4[=cost], -mux8[=cost], -mux16[=cost]
        Cover $_MUX_ trees using the specified types of MUXes (with optional
        integer costs). If none of these options are given, the effect is the
        same as if all of them are.
        Default costs: $_MUX4_ = 220, $_MUX8_ = 460,
                       $_MUX16_ = 940
```

```

    -mux2=cost
        Use the specified cost for $_MUX_ cells when making covering decisions.
        Default cost: $_MUX_ = 100

    -dmux=cost
        Use the specified cost for $_MUX_ cells used in decoders.
        Default cost: 90

    -nodecode
        Do not insert decoder logic. This reduces the number of possible
        substitutions, but guarantees that the resulting circuit is not
        less efficient than the original circuit.

    -nopartial
        Do not consider mappings that use $_MUX<N>_ to select from less
        than <N> different signals.
```

C.114 muxpack – $mux/$pmux cascades to $pmux

```
    muxpack [selection]

This pass converts cascaded chains of $pmux cells (e.g. those create from case
constructs) and $mux cells (e.g. those created by if-else constructs) into
$pmux cells.

This optimisation is conservative --- it will only pack $mux or $pmux cells
whose select lines are driven by '$eq' cells with other such cells if it can be
certain that their select inputs are mutually exclusive.
```

C.115 nlutmap – map to LUTs of different sizes

```
    nlutmap [options] [selection]

This pass uses successive calls to 'abc' to map to an architecture. That
provides a small number of differently sized LUTs.

    -luts N_1,N_2,N_3,...
        The number of LUTs with 1, 2, 3, ... inputs that are
        available in the target architecture.

    -assert
        Create an error if not all logic can be mapped

Excess logic that does not fit into the specified LUTs is mapped back
to generic logic gates ($_AND_, etc.).
```

C.116 onehot – optimize $eq cells for onehot signals

```
    onehot [options] [selection]

This pass optimizes $eq cells that compare one-hot signals against constants

    -v, -vv
        verbose output
```

C.117 opt – perform simple optimizations

```
    opt [options] [selection]

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

    opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-keepdc]
    opt_merge [-share_all] -nomux

    do
        opt_muxtree
        opt_reduce [-fine] [-full]
        opt_merge [-share_all]
        opt_share  (-full only)
        opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat]  (except when called with -noff)
        opt_clean [-purge]
        opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-keepdc]
    while <changed design>

When called with -fast the following script is used instead:

    do
        opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-keepdc]
        opt_merge [-share_all]
        opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat]  (except when called with -noff)
        opt_clean [-purge]
    while <changed design in opt_dff>

Note: Options in square brackets (such as [-keepdc]) are passed through to
the opt_* commands when given to 'opt'.
```

C.118 opt_clean – remove unused cells and wires

```
    opt_clean [options] [selection]

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

```
after the passes that do the actual work.

This pass only operates on completely selected modules without processes.

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

C.119 opt_demorgan – Optimize reductions with DeMorgan equivalents

```
    opt_demorgan [selection]

This pass pushes inverters through $reduce_* cells if this will reduce the
overall gate count of the circuit
```

C.120 opt_dff – perform DFF optimizations

```
    opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat] [selection]

This pass converts flip-flops to a more suitable type by merging clock enables
and synchronous reset multiplexers, removing unused control inputs, or potentially
removes the flip-flop altogether, converting it to a constant driver.

    -nodffe
        disables dff -> dffe conversion, and other transforms recognizing clock enable

    -nosdff
        disables dff -> sdff conversion, and other transforms recognizing sync resets

    -simple-dffe
        only enables clock enable recognition transform for obvious cases

    -sat
        additionally invoke SAT solver to detect and remove flip-flops (with
        non-constant inputs) that can also be replaced with a constant driver

    -keepdc
        some optimizations change the behavior of the circuit with respect to
        don't-care bits. for example in 'a+0' a single x-bit in 'a' will cause
        all result bits to be set to x. this behavior changes when 'a+0' is
        replaced by 'a'. the -keepdc option disables all such optimizations.
```

C.121 opt_expr – perform const folding and simple expression rewriting

```
    opt_expr [options] [selection]

This pass performs const folding on internal cell types with constant inputs.
It also performs some simple expression rewriting.

    -mux_undef
        remove 'undef' inputs from $mux, $pmux and $_MUX_ cells

    -mux_bool
        replace $mux cells with inverters or buffers when possible

    -undriven
        replace undriven nets with undef (x) constants

    -noclkinv
        do not optimize clock inverters by changing FF types

    -fine
        perform fine-grain optimizations

    -full
        alias for -mux_undef -mux_bool -undriven -fine

    -keepdc
        some optimizations change the behavior of the circuit with respect to
        don't-care bits. for example in 'a+0' a single x-bit in 'a' will cause
        all result bits to be set to x. this behavior changes when 'a+0' is
        replaced by 'a'. the -keepdc option disables all such optimizations.
```

C.122 opt_lut – optimize LUT cells

```
    opt_lut [options] [selection]

This pass combines cascaded $lut cells with unused inputs.

    -dlogic <type>:<cell-port>=<LUT-input>[:<cell-port>=<LUT-input>...]
        preserve connections to dedicated logic cell <type> that has ports
        <cell-port> connected to LUT inputs <LUT-input>. this includes
        the case where both LUT and dedicated logic input are connected to
        the same constant.

    -limit N
        only perform the first N combines, then stop. useful for debugging.
```

C.123 opt_lut_ins – discard unused LUT inputs

```
    opt_lut_ins [options] [selection]

This pass removes unused inputs from LUT cells (that is, inputs that can not
```

```
influence the output signal given this LUT's value).  While such LUTs cannot
be directly emitted by ABC, they can be a result of various post-ABC
transformations, such as mapping wide LUTs (not all sub-LUTs will use the
full set of inputs) or optimizations such as xilinx_dffopt.

    -tech <technology>
        Instead of generic $lut cells, operate on LUT cells specific
        to the given technology.  Valid values are: xilinx, ecp5, gowin.
```

C.124 opt_mem – optimize memories

```
    opt_mem [options] [selection]

This pass performs various optimizations on memories in the design.
```

C.125 opt_mem_feedback – convert memory read-to-write port feedback paths to write enables

```
    opt_mem_feedback [selection]

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

C.126 opt_mem_priority – remove priority relations between write ports that can never collide

```
    opt_mem_priority [selection]

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

C.127 opt_mem_widen – optimize memories where all ports are wide

```
    opt_mem_widen [options] [selection]

This pass looks for memories where all ports are wide and adjusts the base
memory width up until that stops being the case.
```

C.128 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.
```

C.129 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.130 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.131 opt_share – merge mutually exclusive cells of the same type that share an input signal

```
    opt_share [selection]

This pass identifies mutually exclusive cells of the same type that:
    (a) share an input signal,
    (b) drive the same $mux, $_MUX_, or $pmux multiplexing cell,

allowing the cell to be merged and the multiplexer to be moved from
multiplexing its output to multiplexing the non-shared input signals.
```

C.132 paramap – renaming cell parameters

```
    paramap [options] [selection]

This command renames cell parameters and/or maps key/value pairs to
other key/value pairs.

    -tocase <name>
        Match attribute names case-insensitively and set it to the specified
        name.

    -rename <old_name> <new_name>
        Rename attributes as specified

    -map <old_name>=<old_value> <new_name>=<new_value>
        Map key/value pairs as indicated.

    -imap <old_name>=<old_value> <new_name>=<new_value>
        Like -map, but use case-insensitive match for <old_value> when
        it is a string value.

    -remove <name>=<value>
        Remove attributes matching this pattern.

For example, mapping Diamond-style ECP5 "init" attributes to Yosys-style:

    paramap -tocase INIT t:LUT4
```

C.133 peepopt – collection of peephole optimizers

```
    peepopt [options] [selection]

This pass applies a collection of peephole optimizers to the current design.
```

C.134 plugin – load and list loaded plugins

```
    plugin [options]

Load and list loaded plugins.

    -i <plugin_filename>
        Load (install) the specified plugin.

    -a <alias_name>
        Register the specified alias name for the loaded plugin

    -l
        List loaded plugins
```

C.135 pmux2shiftx – transform $pmux cells to $shiftx cells

```
    pmux2shiftx [options] [selection]

This pass transforms $pmux cells to $shiftx cells.

    -v, -vv
        verbose output

    -min_density <percentage>
        specifies the minimum density for the shifter
        default: 50

    -min_choices <int>
        specified the minimum number of choices for a control signal
        default: 3

    -onehot ignore|pmux|shiftx
        select strategy for one-hot encoded control signals
        default: pmux

    -norange
        disable $sub inference for "range decoders"
```

C.136 pmuxtree – transform $pmux cells to trees of $mux cells

```
    pmuxtree [selection]

This pass transforms $pmux cells to trees of $mux cells.
```

C.137 portlist – list (top-level) ports

```
    portlist [options] [selection]

This command lists all module ports found in the selected modules.

If no selection is provided then it lists the ports on the top module.

  -m
    print verilog blackbox module definitions instead of port lists
```

C.138 prep – generic synthesis script

```
    prep [options]

This command runs a conservative RTL synthesis. A typical application for this
is the preparation stage of a verification flow. This command does not operate
on partly selected designs.

    -top <module>
        use the specified module as top module (default='top')

    -auto-top
        automatically determine the top of the design hierarchy

    -flatten
        flatten the design before synthesis. this will pass '-auto-top' to
        'hierarchy' if no top module is specified.

    -ifx
        passed to 'proc'. uses verilog simulation behavior for verilog if/case
        undef handling. this also prevents 'wreduce' from being run.

    -memx
        simulate verilog simulation behavior for out-of-bounds memory accesses
        using the 'memory_memx' pass.

    -nomem
        do not run any of the memory_* passes

    -rdff
        call 'memory_dff'. This enables merging of FFs into
        memory read ports.

    -nokeepdc
        do not call opt_* with -keepdc

    -run <from_label>[:<to_label>]
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.


```

```
The following commands are executed by this synthesis command:

    begin:
        hierarchy -check [-top <top> | -auto-top]

    coarse:
        proc [-ifx]
        flatten    (if -flatten)
        opt_expr -keepdc
        opt_clean
        check
        opt -noff -keepdc
        wreduce -keepdc [-memx]
        memory_dff    (if -rdff)
        memory_memx    (if -memx)
        opt_clean
        memory_collect
        opt -noff -keepdc -fast

    check:
        stat
        check
```

C.139 printattrs – print attributes of selected objects

```
    printattrs [selection]

Print all attributes of the selected objects.
```

C.140 proc – translate processes to netlists

```
    proc [options] [selection]

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

    proc_clean
    proc_rmdead
    proc_prune
    proc_init
    proc_arst
    proc_mux
    proc_dlatch
    proc_dff
    proc_memwr
    proc_clean
    opt_expr -keepdc

This replaces the processes in the design with multiplexers,
flip-flops and latches.

```

```
The following options are supported:

    -nomux
        Will omit the proc_mux pass.

    -global_arst [!]<netname>
        This option is passed through to proc_arst.

    -ifx
        This option is passed through to proc_mux. proc_rmdead is not
        executed in -ifx mode.

    -noopt
        Will omit the opt_expr pass.
```

C.141 proc_arst – detect asynchronous resets

```
    proc_arst [-global_arst [!]<netname>] [selection]

This pass identifies asynchronous resets in the processes and converts them
to a different internal representation that is suitable for generating
flip-flop cells with asynchronous resets.

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

C.142 proc_clean – remove empty parts of processes

```
    proc_clean [options] [selection]

    -quiet
        do not print any messages.

This pass removes empty parts of processes and ultimately removes a process
if it contains only empty structures.
```

C.143 proc_dff – extract flip-flops from processes

```
    proc_dff [selection]

This pass identifies flip-flops in the processes and converts them to
d-type flip-flop cells.
```

C.144 proc_dlatch – extract latches from processes

```
    proc_dlatch [selection]

This pass identifies latches in the processes and converts them to
d-type latches.
```

C.145 proc_init – convert initial block to init attributes

```
    proc_init [selection]

This pass extracts the 'init' actions from processes (generated from Verilog
'initial' blocks) and sets the initial value to the 'init' attribute on the
respective wire.
```

C.146 proc_memwr – extract memory writes from processes

```
    proc_memwr [selection]

This pass converts memory writes in processes into $memwr cells.
```

C.147 proc_mux – convert decision trees to multiplexers

```
    proc_mux [options] [selection]

This pass converts the decision trees in processes (originating from if-else
and case statements) to trees of multiplexer cells.

    -ifx
        Use Verilog simulation behavior with respect to undef values in
        'case' expressions and 'if' conditions.
```

C.148 proc_prune – remove redundant assignments

```
    proc_prune [selection]

This pass identifies assignments in processes that are always overwritten by
a later assignment to the same signal and removes them.
```

C.149 proc_rmdead – eliminate dead trees in decision trees

```
    proc_rmdead [selection]

This pass identifies unreachable branches in decision trees and removes them.
```

C.150 qbfsat – solve a 2QBF-SAT problem in the circuit

```
    qbfsat [options] [selection]

This command solves an "exists-forall" 2QBF-SAT problem defined over the currently
selected module. Existentially-quantified variables are declared by assigning a wire
"$anyconst". Universally-quantified variables may be explicitly declared by assigning
a wire "$allconst", but module inputs will be treated as universally-quantified
variables by default.

    -nocleanup
        Do not delete temporary files and directories. Useful for debugging.

    -dump-final-smt2 <file>
        Pass the --dump-smt2 option to yosys-smtbmc.

    -assume-outputs
        Add an "$assume" cell for the conjunction of all one-bit module output wires.

    -assume-negative-polarity
        When adding $assume cells for one-bit module output wires, assume they are
        negative polarity signals and should always be low, for example like the
        miters created with the ‘miter‘ command.

    -nooptimize
        Ignore "\minimize" and "\maximize" attributes, do not emit "(maximize)" or
        "(minimize)" in the SMT-LIBv2, and generally make no attempt to optimize anything.

    -nobisection
        If a wire is marked with the "\minimize" or "\maximize" attribute, do not
        attempt to optimize that value with the default iterated solving and threshold
        bisection approach. Instead, have yosys-smtbmc emit a "(minimize)" or "(maximize)"
        command in the SMT-LIBv2 output and hope that the solver supports optimizing
        quantified bitvector problems.

    -solver <solver>
        Use a particular solver. Choose one of: "z3", "yices", and "cvc4".
        (default: yices)

    -solver-option <name> <value>
        Set the specified solver option in the SMT-LIBv2 problem file.

    -timeout <value>
        Set the per-iteration timeout in seconds.
        (default: no timeout)

    -O0, -O1, -O2
        Control the use of ABC to simplify the QBF-SAT problem before solving.

    -sat
        Generate an error if the solver does not return "sat".

    -unsat
        Generate an error if the solver does not return "unsat".
```

```

    -show-smtbmc
        Print the output from yosys-smtbmc.

    -specialize
        If the problem is satisfiable, replace each "$anyconst" cell with its
        corresponding constant value from the model produced by the solver.

    -specialize-from-file <solution file>
        Do not run the solver, but instead only attempt to replace each "$anyconst"
        cell in the current module with a constant value provided by the specified file.

    -write-solution <solution file>
        If the problem is satisfiable, write the corresponding constant value for each
        "$anyconst" cell from the model produced by the solver to the specified file.
```

C.151 qwp – quadratic wirelength placer

```
    qwp [options] [selection]

This command runs quadratic wirelength placement on the selected modules and
annotates the cells in the design with 'qwp_position' attributes.

    -ltr
        Add left-to-right constraints: constrain all inputs on the left border
        outputs to the right border.

    -alpha
        Add constraints for inputs/outputs to be placed in alphanumerical
        order along the y-axis (top-to-bottom).

    -grid N
        Number of grid divisions in x- and y-direction. (default=16)

    -dump <html_file_name>
        Dump a protocol of the placement algorithm to the html file.

    -v
        Verbose solver output for profiling or debugging

Note: This implementation of a quadratic wirelength placer uses exact
dense matrix operations. It is only a toy-placer for small circuits.
```

C.152 read – load HDL designs

```
    read {-vlog95|-vlog2k|-sv2005|-sv2009|-sv2012|-sv|-formal} <verilog-file>..

Load the specified Verilog/SystemVerilog files. (Full SystemVerilog support
is only available via Verific.)

```

```
Additional -D<macro>[=<value>] options may be added after the option indicating
the language version (and before file names) to set additional verilog defines.


    read {-f|-F} <command-file>

Load and execute the specified command file. (Requires Verific.)
Check verific command for more information about supported commands in file.


    read -define <macro>[=<value>]..

Set global Verilog/SystemVerilog defines.


    read -undef <macro>..

Unset global Verilog/SystemVerilog defines.


    read -incdir <directory>

Add directory to global Verilog/SystemVerilog include directories.


    read -verific
    read -noverific

Subsequent calls to 'read' will either use or not use Verific. Calling 'read'
with -verific will result in an error on Yosys binaries that are built without
Verific support. The default is to use Verific if it is available.
```

C.153 read_aiger – read AIGER file

```
    read_aiger [options] [filename]

Load module from an AIGER file into the current design.

    -module_name <module_name>
        name of module to be created (default: <filename>)

    -clk_name <wire_name>
        if specified, AIGER latches to be transformed into $_DFF_P_ cells
        clocked by wire of this name. otherwise, $_FF_ cells will be used

    -map <filename>
        read file with port and latch symbols

    -wideports
        merge ports that match the pattern 'name[int]' into a single
        multi-bit port 'name'

```

```
    -xaiger
        read XAIGER extensions
```

C.154 read_blif – read BLIF file

```
    read_blif [options] [filename]

Load modules from a BLIF file into the current design.

    -sop
        Create $sop cells instead of $lut cells

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

C.155 read_ilang – (deprecated) alias of read_rtlil

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

C.156 read_json – read JSON file

```
    read_json [filename]

Load modules from a JSON file into the current design See "help write_json"
for a description of the file format.
```

C.157 read_liberty – read cells from liberty file

```
    read_liberty [filename]

Read cells from liberty file as modules into current design.

    -lib
        only create empty blackbox modules

    -nooverwrite
        ignore re-definitions of modules. (the default behavior is to
        create an error message if the existing module is not a blackbox
        module, and overwrite the existing module if it is  a blackbox module.)

    -overwrite
        overwrite existing modules with the same name

    -ignore_miss_func
```

```
        ignore cells with missing function specification of outputs

    -ignore_miss_dir
        ignore cells with a missing or invalid direction
        specification on a pin

    -ignore_miss_data_latch
        ignore latches with missing data and/or enable pins

    -setattr <attribute_name>
        set the specified attribute (to the value 1) on all loaded modules
```

C.158 read_rtlil – read modules from RTLIL file

```
    read_rtlil [filename]

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

    -nooverwrite
        ignore re-definitions of modules. (the default behavior is to
        create an error message if the existing module is not a blackbox
        module, and overwrite the existing module if it is a blackbox module.)

    -overwrite
        overwrite existing modules with the same name

    -lib
        only create empty blackbox modules
```

C.159 read_verilog – read modules from Verilog file

```
    read_verilog [options] [filename]

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

    -sv
        enable support for SystemVerilog features. (only a small subset
        of SystemVerilog is supported)

    -formal
        enable support for SystemVerilog assertions and some Yosys extensions
        replace the implicit -D SYNTHESIS with -D FORMAL

    -nosynthesis
        don't add implicit -D SYNTHESIS

    -noassert
        ignore assert() statements
```

```

    -noassume
        ignore assume() statements

    -norestrict
        ignore restrict() statements

    -assume-asserts
        treat all assert() statements like assume() statements

    -assert-assumes
        treat all assume() statements like assert() statements

    -debug
        alias for -dump_ast1 -dump_ast2 -dump_vlog1 -dump_vlog2 -yydebug

    -dump_ast1
        dump abstract syntax tree (before simplification)

    -dump_ast2
        dump abstract syntax tree (after simplification)

    -no_dump_ptr
        do not include hex memory addresses in dump (easier to diff dumps)

    -dump_vlog1
        dump ast as Verilog code (before simplification)

    -dump_vlog2
        dump ast as Verilog code (after simplification)

    -dump_rtlil
        dump generated RTLIL netlist

    -yydebug
        enable parser debug output

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

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

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

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

```

```
        This is potentially dangerous. Usually the front-end has good
        reasons for converting an array to a list of registers.
        Prohibiting this step will likely result in incorrect synthesis
        results.

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

    -nomeminit
        do not infer $meminit cells and instead convert initialized
        memories to registers directly in the front-end.

    -ppdump
        dump Verilog code after pre-processor

    -nopp
        do not run the pre-processor

    -nodpi
        disable DPI-C support

    -noblackbox
        do not automatically add a (* blackbox *) attribute to an
        empty module.

    -lib
        only create empty blackbox modules. This implies -DBLACKBOX.
        modules with the (* whitebox *) attribute will be preserved.
        (* lib_whitebox *) will be treated like (* whitebox *).

    -nowb
        delete (* whitebox *) and (* lib_whitebox *) attributes from
        all modules.

    -specify
        parse and import specify blocks

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

    -icells
        interpret cell types starting with '$' as internal cell types

    -pwires
        add a wire for each module parameter

    -nooverwrite
        ignore re-definitions of modules. (the default behavior is to
        create an error message if the existing module is not a black box
        module, and overwrite the existing module otherwise.)

```

```
    -overwrite
        overwrite existing modules with the same name

    -defer
        only read the abstract syntax tree and defer actual compilation
        to a later 'hierarchy' command. Useful in cases where the default
        parameters of modules yield invalid or not synthesizable code.

    -noautowire
        make the default of `default_nettype be "none" instead of "wire".

    -setattr <attribute_name>
        set the specified attribute (to the value 1) on all loaded modules

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

    -Idir
        add 'dir' to the directories which are used when searching include
        files

The command 'verilog_defaults' can be used to register default options for
subsequent calls to 'read_verilog'.

Note that the Verilog frontend does a pretty good job of processing valid
verilog input, but has not very good error reporting. It generally is
recommended to use a simulator (for example Icarus Verilog) for checking
the syntax of the code, rather than to rely on read_verilog for that.

Depending on if read_verilog is run in -formal mode, either the macro
SYNTHESIS or FORMAL is defined automatically, unless -nosynthesis is used.
In addition, read_verilog always defines the macro YOSYS.

See the Yosys README file for a list of non-standard Verilog features
supported by the Yosys Verilog front-end.
```

C.160 rename – rename object in the design

```
    rename old_name new_name

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



    rename -output old_name new_name

Like above, but also make the wire an output. This will fail if the object is
not a wire.


```

```
    rename -src [selection]

Assign names auto-generated from the src attribute to all selected wires and
cells with private names.


    rename -wire [selection]

Assign auto-generated names based on the wires they drive to all selected
cells with private names. Ignores cells driving privatly named wires.


    rename -enumerate [-pattern <pattern>] [selection]

Assign short auto-generated names to all selected wires and cells with private
names. The -pattern option can be used to set the pattern for the new names.
The character % in the pattern is replaced with a integer number. The default
pattern is '_%_'.


    rename -hide [selection]

Assign private names (the ones with $-prefix) to all selected wires and cells
with public names. This ignores all selected ports.


    rename -top new_name

Rename top module.
```

C.161 rmports – remove module ports with no connections

```
    rmports [selection]

This pass identifies ports in the selected modules which are not used or
driven and removes them.
```

C.162 sat – solve a SAT problem in the circuit

```
    sat [options] [selection]

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

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

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

```

    -enable_undef
        enable modeling of undef value (aka 'x-bits')
        this option is implied by -set-def, -set-undef et. cetera

    -max_undef
        maximize the number of undef bits in solutions, giving a better
        picture of which input bits are actually vital to the solution.

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

    -set-def <signal>
        add a constraint that all bits of the given signal must be defined

    -set-any-undef <signal>
        add a constraint that at least one bit of the given signal is undefined

    -set-all-undef <signal>
        add a constraint that all bits of the given signal are undefined

    -set-def-inputs
        add -set-def constraints for all module inputs

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

    -show-inputs, -show-outputs, -show-ports
        add all module (input/output) ports to the list of shown signals

    -show-regs, -show-public, -show-all
        show all registers, show signals with 'public' names, show all signals

    -ignore_div_by_zero
        ignore all solutions that involve a division by zero

    -ignore_unknown_cells
        ignore all cells that can not be matched to a SAT model

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

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

        note: for large <N> it can be significantly faster to use
        -tempinduct-baseonly -maxsteps <N> instead of -seq <N>.

    -set-at <N> <signal> <value>
    -unset-at <N> <signal>
        set or unset the specified signal to the specified value in the
        given timestep. this has priority over a -set for the same signal.

```

```
    -set-assumes
        set all assumptions provided via $assume cells

    -set-def-at <N> <signal>
    -set-any-undef-at <N> <signal>
    -set-all-undef-at <N> <signal>
        add undef constraints in the given timestep.

    -set-init <signal> <value>
        set the initial value for the register driving the signal to the value

    -set-init-undef
        set all initial states (not set using -set-init) to undef

    -set-init-def
        do not force a value for the initial state but do not allow undef

    -set-init-zero
        set all initial states (not set using -set-init) to zero

    -dump_vcd <vcd-file-name>
        dump SAT model (counter example in proof) to VCD file

    -dump_json <json-file-name>
        dump SAT model (counter example in proof) to a WaveJSON file.

    -dump_cnf <cnf-file-name>
        dump CNF of SAT problem (in DIMACS format). in temporal induction
        proofs this is the CNF of the first induction step.

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

    -tempinduct
        Perform a temporal induction proof. In a temporal induction proof it is
        proven that the condition holds forever after the number of time steps
        specified using -seq.

    -tempinduct-def
        Perform a temporal induction proof. Assume an initial state with all
        registers set to defined values for the induction step.

    -tempinduct-baseonly
        Run only the basecase half of temporal induction (requires -maxsteps)

    -tempinduct-inductonly
        Run only the induction half of temporal induction

    -tempinduct-skip <N>
        Skip the first <N> steps of the induction proof.

        note: this will assume that the base case holds for <N> steps.
        this must be proven independently with "-tempinduct-baseonly
        -maxsteps <N>". Use -initsteps if you just want to set a
```

```
        minimal induction length.

    -prove <signal> <value>
        Attempt to proof that <signal> is always <value>.

    -prove-x <signal> <value>
        Like -prove, but an undef (x) bit in the lhs matches any value on
        the right hand side. Useful for equivalence checking.

    -prove-asserts
        Prove that all asserts in the design hold.

    -prove-skip <N>
        Do not enforce the prove-condition for the first <N> time steps.

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

    -initsteps <N>
        Set initial length for the induction.
        This will speed up the search of the right induction length
        for deep induction proofs.

    -stepsize <N>
        Increase the size of the induction proof in steps of <N>.
        This will speed up the search of the right induction length
        for deep induction proofs.

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

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

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

    -falsify
        Return an error and stop the synthesis script if the proof succeeds.

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

C.163 scatter – add additional intermediate nets

```
    scatter [selection]

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

```

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

C.164 scc – detect strongly connected components (logic loops)

```
    scc [options] [selection]

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

    -expect <num>
        expect to find exactly <num> SCCs. A different number of SCCs will
        produce an error.

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

    -nofeedback
        do not count cells that have their output fed back into one of their
        inputs as single-cell scc.

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

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

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

    -specify
        examine specify rules to detect logic loops in whitebox/blackbox cells
```

C.165 scratchpad – get/set values in the scratchpad

```
    scratchpad [options]

This pass allows to read and modify values from the scratchpad of the current
design. Options:

    -get <identifier>
        print the value saved in the scratchpad under the given identifier.

    -set <identifier> <value>
```

```
        save the given value in the scratchpad under the given identifier.

    -unset <identifier>
        remove the entry for the given identifier from the scratchpad.

    -copy <identifier_from> <identifier_to>
        copy the value of the first identifier to the second identifier.

    -assert <identifier> <value>
        assert that the entry for the given identifier is set to the given value.

    -assert-set <identifier>
        assert that the entry for the given identifier exists.

    -assert-unset <identifier>
        assert that the entry for the given identifier does not exist.

The identifier may not contain whitespace. By convention, it is usually prefixed
by the name of the pass that uses it, e.g. 'opt.did_something'. If the value
contains whitespace, it must be enclosed in double quotes.
```

C.166 script – execute commands from file or wire

```
    script <filename> [<from_label>:<to_label>]
    script -scriptwire [selection]

This command executes the yosys commands in the specified file (default
behaviour), or commands embedded in the constant text value connected to the
selected wires.

In the default (file) case, the 2nd argument can be used to only execute the
section of the file between the specified labels. An empty from label is
synonymous with the beginning of the file and an empty to label is synonymous
with the end of the file.

If only one label is specified (without ':') then only the block
marked with that label (until the next label) is executed.

In "-scriptwire" mode, the commands on the selected wire(s) will be executed
in the scope of (and thus, relative to) the wires' owning module(s). This
'-module' mode can be exited by using the 'cd' command.
```

C.167 select – modify and view the list of selected objects

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

```

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

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

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

    -set <name>
        do not modify the current selection. instead save the new selection
        under the given name (see @<name> below). to save the current selection,
        use "select -set <name> %"

    -unset <name>
        do not modify the current selection. instead remove a previously saved
        selection under the given name (see @<name> below).
    -assert-none
        do not modify the current selection. instead assert that the given
        selection is empty. i.e. produce an error if any object matching the
        selection is found.

    -assert-any
        do not modify the current selection. instead assert that the given
        selection is non-empty. i.e. produce an error if no object matching
        the selection is found.

    -assert-count N
        do not modify the current selection. instead assert that the given
        selection contains exactly N objects.

    -assert-max N
        do not modify the current selection. instead assert that the given
        selection contains less than or exactly N objects.

    -assert-min N
        do not modify the current selection. instead assert that the given
        selection contains at least N objects.

    -list
        list all objects in the current selection

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

    -read <filename>
        read the specified file (written by -write)

    -count
        count all objects in the current selection
```

```

    -clear
        clear the current selection. this effectively selects the whole
        design. it also resets the selected module (see -module). use the
        command 'select *' to select everything but stay in the current module.

    -none
        create an empty selection. the current module is unchanged.

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

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

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

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

    <mod_pattern>
        select the specified module(s)

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

Pushing (selecting) object when in -module mode:

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

By default, patterns will not match black/white-box modules or theircontents. To include such objects, pref

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

    A:<pattern>, A:<pattern>=<pattern>
        all modules with an attribute matching the given pattern
        in addition to = also <, <=, >=, and > are supported

    N:<pattern>
        all modules with a name matching the given pattern
        (i.e. 'N:' is optional as it is the default matching rule)

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

    w:<pattern>
```

```
        all wires with a name matching the given wildcard pattern

    i:<pattern>, o:<pattern>, x:<pattern>
        all inputs (i:), outputs (o:) or any ports (x:) with matching names

    s:<size>, s:<min>:<max>
        all wires with a matching width

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

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

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

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

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

    a:<pattern>=<pattern>
        all objects with a matching attribute name-value-pair.
        in addition to = also <, <=, >=, and > are supported

    r:<pattern>, r:<pattern>=<pattern>
        cells with matching parameters. also with <, <=, >= and >.

    n:<pattern>
        all objects with a name matching the given pattern
        (i.e. 'n:' is optional as it is the default matching rule)

    @<name>
        push the selection saved prior with 'select -set <name> ...'

The following actions can be performed on the top sets on the stack:

    %
        push a copy of the current selection to the stack

    %%
        replace the stack with a union of all elements on it

    %n
        replace top set with its invert

    %u
        replace the two top sets on the stack with their union

    %i
        replace the two top sets on the stack with their intersection

```

```
    %d
        pop the top set from the stack and subtract it from the new top

    %D
        like %d but swap the roles of two top sets on the stack

    %c
        create a copy of the top set from the stack and push it

    %x[<num1>|*][.<num2>][:<rule>[:<rule>..]]
        expand top set <num1> num times according to the specified rules.
        (i.e. select all cells connected to selected wires and select all
        wires connected to selected cells) The rules specify which cell
        ports to use for this. the syntax for a rule is a '-' for exclusion
        and a '+' for inclusion, followed by an optional comma separated
        list of cell types followed by an optional comma separated list of
        cell ports in square brackets. a rule can also be just a cell or wire
        name that limits the expansion (is included but does not go beyond).
        select at most <num2> objects. a warning message is printed when this
        limit is reached. When '*' is used instead of <num1> then the process
        is repeated until no further object are selected.

    %ci[<num1>|*][.<num2>][:<rule>[:<rule>..]]
    %co[<num1>|*][.<num2>][:<rule>[:<rule>..]]
        similar to %x, but only select input (%ci) or output cones (%co)

    %xe[...] %cie[...] %coe
        like %x, %ci, and %co but only consider combinatorial cells

    %a
        expand top set by selecting all wires that are (at least in part)
        aliases for selected wires.

    %s
        expand top set by adding all modules that implement cells in selected
        modules

    %m
        expand top set by selecting all modules that contain selected objects

    %M
        select modules that implement selected cells

    %C
        select cells that implement selected modules

    %R[<num>]
        select <num> random objects from top selection (default 1)

Example: the following command selects all wires that are connected to a
'GATE' input of a 'SWITCH' cell:

    select */t:SWITCH %x:+[GATE] */t:SWITCH %d
```

C.168 setattr – set/unset attributes on objects

```
    setattr [ -mod ] [ -set name value | -unset name ]... [selection]

Set/unset the given attributes on the selected objects. String values must be
passed in double quotes (").

When called with -mod, this command will set and unset attributes on modules
instead of objects within modules.
```

C.169 setparam – set/unset parameters on objects

```
    setparam [ -type cell_type ] [ -set name value | -unset name ]... [selection]

Set/unset the given parameters on the selected cells. String values must be
passed in double quotes (").

The -type option can be used to change the cell type of the selected cells.
```

C.170 setundef – replace undef values with defined constants

```
    setundef [options] [selection]

This command replaces undef (x) constants with defined (0/1) constants.

    -undriven
        also set undriven nets to constant values

    -expose
        also expose undriven nets as inputs (use with -undriven)

    -zero
        replace with bits cleared (0)

    -one
        replace with bits set (1)

    -undef
        replace with undef (x) bits, may be used with -undriven

    -anyseq
        replace with $anyseq drivers (for formal)

    -anyconst
        replace with $anyconst drivers (for formal)

    -random <seed>
        replace with random bits using the specified integer as seed
        value for the random number generator.
```

```

    -init
        also create/update init values for flip-flops

    -params
        replace undef in cell parameters
```

C.171 share – perform sat-based resource sharing

```
    share [options] [selection]

This pass merges shareable resources into a single resource. A SAT solver
is used to determine if two resources are share-able.

  -force
    Per default the selection of cells that is considered for sharing is
    narrowed using a list of cell types. With this option all selected
    cells are considered for resource sharing.

    IMPORTANT NOTE: If the -all option is used then no cells with internal
    state must be selected!

  -aggressive
    Per default some heuristics are used to reduce the number of cells
    considered for resource sharing to only large resources. This options
    turns this heuristics off, resulting in much more cells being considered
    for resource sharing.

  -fast
    Only consider the simple part of the control logic in SAT solving, resulting
    in much easier SAT problems at the cost of maybe missing some opportunities
    for resource sharing.

  -limit N
    Only perform the first N merges, then stop. This is useful for debugging.
```

C.172 shell – enter interactive command mode

```
    shell

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

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

    yosys>
        the entire design is selected

```

```
    yosys*>
        only part of the design is selected

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

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

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

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

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

C.173 show – generate schematics using graphviz

```
    show [options] [selection]

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

    -viewer <viewer>
        Run the specified command with the graphics file as parameter.
        On Windows, this pauses yosys until the viewer exits.

    -format <format>
        Generate a graphics file in the specified format. Use 'dot' to just
        generate a .dot file, or other <format> strings such as 'svg' or 'ps'
        to generate files in other formats (this calls the 'dot' command).

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

        note: in most cases it is better to load the library before calling
        show with 'read_verilog -lib <filename>'. it is also possible to
        load liberty files with 'read_liberty -lib <filename>'.

    -prefix <prefix>
        generate <prefix>.* instead of ~/.yosys_show.*

    -color <color> <object>
        assign the specified color to the specified object. The object can be
        a single selection wildcard expressions or a saved set of objects in
        the @<name> syntax (see "help select" for details).

    -label <text> <object>
        assign the specified label text to the specified object. The object can
```

```
        be a single selection wildcard expressions or a saved set of objects in
        the @<name> syntax (see "help select" for details).

    -colors <seed>
        Randomly assign colors to the wires. The integer argument is the seed
        for the random number generator. Change the seed value if the colored
        graph still is ambiguous. A seed of zero deactivates the coloring.

    -colorattr <attribute_name>
        Use the specified attribute to assign colors. A unique color is
        assigned to each unique value of this attribute.

    -width
        annotate buses with a label indicating the width of the bus.

    -signed
        mark ports (A, B) that are declared as signed (using the [AB]_SIGNED
        cell parameter) with an asterisk next to the port name.

    -stretch
        stretch the graph so all inputs are on the left side and all outputs
        (including inout ports) are on the right side.

    -pause
        wait for the user to press enter to before returning

    -enum
        enumerate objects with internal ($-prefixed) names

    -long
        do not abbreviate objects with internal ($-prefixed) names

    -notitle
        do not add the module name as graph title to the dot file

    -nobg
        don't run viewer in the background, IE wait for the viewer tool to
        exit before returning

When no <format> is specified, 'dot' is used. When no <format> and <viewer> is
specified, 'xdot' is used to display the schematic (POSIX systems only).

The generated output files are '~/.yosys_show.dot' and '~/.yosys_show.<format>',
unless another prefix is specified using -prefix <prefix>.

Yosys on Windows and YosysJS use different defaults: The output is written
to 'show.dot' in the current directory and new viewer is launched each time
the 'show' command is executed.
```

C.174 shregmap – map shift registers

```
    shregmap [options] [selection]
```

```

This pass converts chains of $_DFF_[NP]_ gates to target specific shift register
primitives. The generated shift register will be of type $__SHREG_DFF_[NP]_ and
will use the same interface as the original $_DFF_*_ cells. The cell parameter
'DEPTH' will contain the depth of the shift register. Use a target-specific
'techmap' map file to convert those cells to the actual target cells.

    -minlen N
        minimum length of shift register (default = 2)
        (this is the length after -keep_before and -keep_after)

    -maxlen N
        maximum length of shift register (default = no limit)
        larger chains will be mapped to multiple shift register instances

    -keep_before N
        number of DFFs to keep before the shift register (default = 0)

    -keep_after N
        number of DFFs to keep after the shift register (default = 0)

    -clkpol pos|neg|any
        limit match to only positive or negative edge clocks. (default = any)

    -enpol pos|neg|none|any_or_none|any
        limit match to FFs with the specified enable polarity. (default = none)

    -match <cell_type>[:<d_port_name>:<q_port_name>]
        match the specified cells instead of $_DFF_N_ and $_DFF_P_. If
        ':<d_port_name>:<q_port_name>' is omitted then 'D' and 'Q' is used
        by default. E.g. the option '-clkpol pos' is just an alias for
        '-match $_DFF_P_', which is an alias for '-match $_DFF_P_:D:Q'.

    -params
        instead of encoding the clock and enable polarity in the cell name by
        deriving from the original cell name, simply name all generated cells
        $__SHREG_ and use CLKPOL and ENPOL parameters. An ENPOL value of 2 is
        used to denote cells without enable input. The ENPOL parameter is
        omitted when '-enpol none' (or no -enpol option) is passed.

    -zinit
        assume the shift register is automatically zero-initialized, so it
        becomes legal to merge zero initialized FFs into the shift register.

    -init
        map initialized registers to the shift reg, add an INIT parameter to
        generated cells with the initialization value. (first bit to shift out
        in LSB position)

    -tech greenpak4
        map to greenpak4 shift registers.
```

C.175 sim – simulate the circuit

```
    sim [options] [top-level]

This command simulates the circuit using the given top-level module.

    -vcd <filename>
        write the simulation results to the given VCD file

    -fst <filename>
        write the simulation results to the given FST file

    -aiw <filename>
        write the simulation results to an AIGER witness file
        (requires a *.aim file via -map)

    -x
        ignore constant x outputs in simulation file.

    -date
        include date and full version info in output.

    -clock <portname>
        name of top-level clock input

    -clockn <portname>
        name of top-level clock input (inverse polarity)

    -reset <portname>
        name of top-level reset input (active high)

    -resetn <portname>
        name of top-level inverted reset input (active low)

    -rstlen <integer>
        number of cycles reset should stay active (default: 1)

    -zinit
        zero-initialize all uninitialized regs and memories

    -timescale <string>
        include the specified timescale declaration in the vcd

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

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

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

    -r
        read simulation results file (file formats supported: FST)
```

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

    -scope <name>
        scope of simulation top model

    -at <time>
        sets start and stop time

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

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

    -sim
        simulation with stimulus from FST (default)

    -sim-cmp
        co-simulation expect exact match

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

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

    -q
        disable per-cycle/sample log message

    -d
        enable debug output
```

C.176 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,
```

C.177 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.178 splitnets – split up multi-bit nets

```
    splitnets [options] [selection]

This command splits multi-bit nets into single-bit nets.

    -format char1[char2[char3]]
        the first char is inserted between the net name and the bit index, the
        second char is appended to the netname. e.g. -format () creates net
        names like 'mysignal(42)'. the 3rd character is the range separation
        character when creating multi-bit wires. the default is '[]:'.

    -ports
        also split module ports. per default only internal signals are split.

    -driver
        don't blindly split nets in individual bits. instead look at the driver
        and split nets so that no driver drives only part of a net.
```

C.179 sta – perform static timing analysis

```
    sta [options] [selection]

This command performs static timing analysis on the design. (Only considers
paths within a single module, so the design must be flattened.)
```

C.180 stat – print some statistics

```
    stat [options] [selection]

Print some statistics (number of objects) on the selected portion of the
design.

    -top <module>
        print design hierarchy with this module as top. if the design is fully
        selected and a module has the 'top' attribute set, this module is used
        default value for this option.

    -liberty <liberty_file>
        use cell area information from the provided liberty file

    -tech <technology>
        print area estemate for the specified technology. Currently supported
        values for <technology>: xilinx, cmos

    -width
        annotate internal cell types with their word width.
        e.g. $add_8 for an 8 bit wide $add cell.
```

C.181 submod – moving part of a module to a new submodule

```
    submod [options] [selection]

This pass identifies all cells with the 'submod' attribute and moves them to
a newly created module. The value of the attribute is used as name for the
cell that replaces the group of cells with the same attribute value.

This pass can be used to create a design hierarchy in flat design. This can
be useful for analyzing or reverse-engineering a design.

This pass only operates on completely selected modules with no processes
or memories.

    -copy
        by default the cells are 'moved' from the source module and the source
        module will use an instance of the new module after this command is
        finished. call with -copy to not modify the source module.

```

```
    -name <name>
        don't use the 'submod' attribute but instead use the selection. only
        objects from one module might be selected. the value of the -name option
        is used as the value of the 'submod' attribute instead.

    -hidden
        instead of creating submodule ports with public names, create ports with
        private names so that a subsequent 'flatten; clean' call will restore the
        original module with original public names.
```

C.182 supercover – add hi/lo cover cells for each wire bit

```
    supercover [options] [selection]

This command adds two cover cells for each bit of each selected wire, one
checking for a hi signal level and one checking for lo level.
```

C.183 synth – generic synthesis script

```
    synth [options]

This command runs the default synthesis script. This command does not operate
on partly selected designs.

    -top <module>
        use the specified module as top module (default='top')

    -auto-top
        automatically determine the top of the design hierarchy

    -flatten
        flatten the design before synthesis. this will pass '-auto-top' to
        'hierarchy' if no top module is specified.

    -encfile <file>
        passed to 'fsm_recode' via 'fsm'

    -lut <k>
        perform synthesis for a k-LUT architecture.

    -nofsm
        do not run FSM optimization

    -noabc
        do not run abc (as if yosys was compiled without ABC support)

    -noalumacc
        do not run 'alumacc' pass. i.e. keep arithmetic operators in
        their direct form ($add, $sub, etc.).

```

```
    -nordff
        passed to 'memory'. prohibits merging of FFs into memory read ports

    -noshare
        do not run SAT-based resource sharing

    -run <from_label>[:<to_label>]
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -abc9
        use new ABC9 flow (EXPERIMENTAL)

    -flowmap
        use FlowMap LUT techmapping instead of ABC


The following commands are executed by this synthesis command:

    begin:
        hierarchy -check [-top <top> | -auto-top]

    coarse:
        proc
        flatten      (if -flatten)
        opt_expr
        opt_clean
        check
        opt -nodffe -nosdff
        fsm          (unless -nofsm)
        opt
        wreduce
        peepopt
        opt_clean
        techmap -map +/cmp2lut.v -map +/cmp2lcu.v     (if -lut)
        alumacc      (unless -noalumacc)
        share        (unless -noshare)
        opt
        memory -nomap
        opt_clean

    fine:
        opt -fast -full
        memory_map
        opt -full
        techmap
        techmap -map +/gate2lut.v    (if -noabc and -lut)
        clean; opt_lut               (if -noabc and -lut)
        flowmap -maxlut K            (if -flowmap and -lut)
        opt -fast
        abc -fast           (unless -noabc, unless -lut)
        abc -fast -lut k    (unless -noabc, if -lut)
        opt -fast           (unless -noabc)
```

```

    check:
        hierarchy -check
        stat
        check
```

C.184 synth_achronix – synthesis for Acrhonix Speedster22i FPGAs.

```
    synth_achronix [options]

This command runs synthesis for Achronix Speedster eFPGAs. This work is still experimental.

    -top <module>
        use the specified module as top module (default='top')

    -vout <file>
        write the design to the specified Verilog netlist file. writing of an
        output file is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -retime
        run 'abc' with '-dff -D 1' options


The following commands are executed by this synthesis command:

    begin:
        read_verilog -sv -lib +/achronix/speedster22i/cells_sim.v
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten
        tribuf -logic
        deminout

    coarse:
        synth -run coarse

    fine:
        opt -fast -mux_undef -undriven -fine -full
        memory_map
        opt -undriven -fine
        opt -fine
```

```
        techmap -map +/techmap.v
        opt -full
        clean -purge
        setundef -undriven -zero
        dfflegalize -cell $_DFF_P_ x
        abc -markgroups -dff -D 1    (only if -retime)

    map_luts:
        abc -lut 4
        clean

    map_cells:
        iopadmap -bits -outpad $__outpad I:O -inpad $__inpad O:I
        techmap -map +/achronix/speedster22i/cells_map.v
        clean -purge

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

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

C.185 synth_anlogic – synthesis for Anlogic FPGAs

```
    synth_anlogic [options]

This command runs synthesis for Anlogic FPGAs.

    -top <module>
        use the specified module as top module

    -edif <file>
        write the design to the specified EDIF file. writing of an output file
        is omitted if this parameter is not specified.

    -json <file>
        write the design to the specified JSON file. writing of an output file
        is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -retime
        run 'abc' with '-dff -D 1' options
```

```

    -nolutram
        do not use EG_LOGIC_DRAM16X4 cells in output netlist

    -nobram
        do not use EG_PHY_BRAM or EG_PHY_BRAM32K cells in output netlist


The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib +/anlogic/cells_sim.v +/anlogic/eagle_bb.v
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten
        tribuf -logic
        deminout

    coarse:
        synth -run coarse

    map_bram:    (skip if -nobram)
        memory_bram -rules +/anlogic/brams.txt
        techmap -map +/anlogic/brams_map.v
        setundef -zero -params t:EG_PHY_BRAM
        setundef -zero -params t:EG_PHY_BRAM32K

    map_lutram:    (skip if -nolutram)
        memory_bram -rules +/anlogic/lutrams.txt
        techmap -map +/anlogic/lutrams_map.v
        setundef -zero -params t:EG_LOGIC_DRAM16X4

    map_ffram:
        opt -fast -mux_undef -undriven -fine
        memory_map
        opt -undriven -fine

    map_gates:
        techmap -map +/techmap.v -map +/anlogic/arith_map.v
        opt -fast
        abc -dff -D 1    (only if -retime)

    map_ffs:
        dfflegalize -cell $_DFFE_P??P_ r -cell $_SDFFE_P??P_ r -cell $_DLATCH_N??_ r
        techmap -D NO_LUT -map +/anlogic/cells_map.v
        opt_expr -mux_undef
        simplemap

    map_luts:
        abc -lut 4:6
        clean

```

```
    map_cells:
        techmap -map +/anlogic/cells_map.v
        clean

    map_anlogic:
        anlogic_fixcarry
        anlogic_eqn

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    edif:
        write_edif <file-name>

    json:
        write_json <file-name>
```

C.186 synth_coolrunner2 – synthesis for Xilinx Coolrunner-II CPLDs

```
    synth_coolrunner2 [options]

This command runs synthesis for Coolrunner-II CPLDs. This work is experimental.
It is intended to be used with https://github.com/azonenberg/openfpga as the
place-and-route.

    -top <module>
        use the specified module as top module (default='top')

    -json <file>
        write the design to the specified JSON file. writing of an output file
        is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -retime
        run 'abc' with '-dff -D 1' options


The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib +/coolrunner2/cells_sim.v
```

```
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten
        tribuf -logic

    coarse:
        synth -run coarse

    fine:
        extract_counter -dir up -allow_arst no
        techmap -map +/coolrunner2/cells_counter_map.v
        clean
        opt -fast -full
        techmap -map +/techmap.v -map +/coolrunner2/cells_latch.v
        opt -fast
        dfflibmap -prepare -liberty +/coolrunner2/xc2_dff.lib

    map_tff:
        abc -g AND,XOR
        clean
        extract -map +/coolrunner2/tff_extract.v

    map_pla:
        abc -sop -I 40 -P 56
        clean

    map_cells:
        dfflibmap -liberty +/coolrunner2/xc2_dff.lib
        dffinit -ff FDCP Q INIT
        dffinit -ff FDCP_N Q INIT
        dffinit -ff FTCP Q INIT
        dffinit -ff FTCP_N Q INIT
        dffinit -ff LDCP Q INIT
        dffinit -ff LDCP_N Q INIT
        coolrunner2_sop
        clean
        iopadmap -bits -inpad IBUF O:I -outpad IOBUFE I:IO -inoutpad IOBUFE O:IO -toutpad IOBUFE E:I:IO -ti
        attrmvcp -attr src -attr LOC t:IOBUFE n:*
        attrmvcp -attr src -attr LOC -driven t:IBUF n:*
        coolrunner2_fixup
        splitnets
        clean

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    json:
        write_json <file-name>
```

C.187 synth_easic – synthesis for eASIC platform

```
    synth_easic [options]

This command runs synthesis for eASIC platform.

    -top <module>
        use the specified module as top module

    -vlog <file>
        write the design to the specified structural Verilog file. writing of
        an output file is omitted if this parameter is not specified.

    -etools <path>
        set path to the eTools installation. (default=/opt/eTools)

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -retime
        run 'abc' with '-dff -D 1' options


The following commands are executed by this synthesis command:

    begin:
        read_liberty -lib <etools_phys_clk_lib>
        read_liberty -lib <etools_logic_lut_lib>
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten

    coarse:
        synth -run coarse

    fine:
        opt -fast -mux_undef -undriven -fine
        memory_map
        opt -undriven -fine
        techmap
        opt -fast
        abc -dff -D 1     (only if -retime)
        opt_clean    (only if -retime)

    map:
        dfflibmap -liberty <etools_phys_clk_lib>
        abc -liberty <etools_logic_lut_lib>
```

```
        opt_clean

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    vlog:
        write_verilog -noexpr -attr2comment <file-name>
```

C.188 synth_ecp5 – synthesis for ECP5 FPGAs

```
    synth_ecp5 [options]

This command runs synthesis for ECP5 FPGAs.

    -top <module>
        use the specified module as top module

    -blif <file>
        write the design to the specified BLIF file. writing of an output file
        is omitted if this parameter is not specified.

    -edif <file>
        write the design to the specified EDIF file. writing of an output file
        is omitted if this parameter is not specified.

    -json <file>
        write the design to the specified JSON file. writing of an output file
        is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -dff
        run 'abc'/'abc9' with -dff option

    -retime
        run 'abc' with '-dff -D 1' options

    -noccu2
        do not use CCU2 cells in output netlist

    -nodffe
        do not use flipflops with CE in output netlist

```

```
    -nobram
        do not use block RAM cells in output netlist

    -nolutram
        do not use LUT RAM cells in output netlist

    -nowidelut
        do not use PFU muxes to implement LUTs larger than LUT4s

    -asyncprld
        use async PRLD mode to implement ALDFF (EXPERIMENTAL)

    -abc2
        run two passes of 'abc' for slightly improved logic density

    -abc9
        use new ABC9 flow (EXPERIMENTAL)

    -vpr
        generate an output netlist (and BLIF file) suitable for VPR
        (this feature is experimental and incomplete)

    -nodsp
        do not map multipliers to MULT18X18D


The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib -specify +/ecp5/cells_sim.v +/ecp5/cells_bb.v
        hierarchy -check -top <top>

    coarse:
        proc
        flatten
        tribuf -logic
        deminout
        opt_expr
        opt_clean
        check
        opt -nodffe -nosdff
        fsm
        opt
        wreduce
        peepopt
        opt_clean
        share
        techmap -map +/cmp2lut.v -D LUT_WIDTH=4
        opt_expr
        opt_clean
        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)
        chtype -set $mul t:$__soft_mul    (unless -nodsp)
        alumacc
```

```
        opt
        memory -nomap
        opt_clean

    map_bram:    (skip if -nobram)
        memory_bram -rules +/ecp5/brams.txt
        techmap -map +/ecp5/brams_map.v

    map_lutram:    (skip if -nolutram)
        memory_bram -rules +/ecp5/lutrams.txt
        techmap -map +/ecp5/lutrams_map.v

    map_ffram:
        opt -fast -mux_undef -undriven -fine
        memory_map -iattr -attr !ram_block -attr !rom_block -attr logic_block -attr syn_ramstyle=auto -attr
        opt -undriven -fine

    map_gates:
        techmap -map +/techmap.v -map +/ecp5/arith_map.v
        opt -fast
        abc -dff -D 1    (only if -retime)

    map_ffs:
        opt_clean
        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, $_*DFFE_* only if not -nodffe)
        zinit -all w:* t:$_DFF_?_ t:$_DFFE_??_ t:$_SDFF*    (only if -abc9 and -dff)
        techmap -D NO_LUT -map +/ecp5/cells_map.v
        opt_expr -undriven -mux_undef
        simplemap
        ecp5_gsr
        attrmvcp -copy -attr syn_useioff
        opt_clean

    map_luts:
        abc          (only if -abc2)
        techmap -map +/ecp5/latches_map.v    (skip if -asyncprld)
        abc -dress -lut 4:7
        clean

    map_cells:
        techmap -map +/ecp5/cells_map.v    (skip if -vpr)
        opt_lut_ins -tech ecp5
        clean

    check:
        autoname
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    blif:
        opt_clean -purge                                     (vpr mode)
```

```
        write_blif -attr -cname -conn -param <file-name>     (vpr mode)
        write_blif -gates -attr -param <file-name>           (non-vpr mode)

    edif:
        write_edif <file-name>

    json:
        write_json <file-name>
```

C.189 synth_efinix – synthesis for Efinix FPGAs

```
    synth_efinix [options]

This command runs synthesis for Efinix FPGAs.

    -top <module>
        use the specified module as top module

    -edif <file>
        write the design to the specified EDIF file. writing of an output file
        is omitted if this parameter is not specified.

    -json <file>
        write the design to the specified JSON file. writing of an output file
        is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -retime
        run 'abc' with '-dff -D 1' options

    -nobram
        do not use EFX_RAM_5K cells in output netlist


The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib +/efinix/cells_sim.v
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten
        tribuf -logic
        deminout
```

```

    coarse:
        synth -run coarse
        memory_bram -rules +/efinix/brams.txt
        techmap -map +/efinix/brams_map.v
        setundef -zero -params t:EFX_RAM_5K

    map_ffram:
        opt -fast -mux_undef -undriven -fine
        memory_map
        opt -undriven -fine

    map_gates:
        techmap -map +/techmap.v -map +/efinix/arith_map.v
        opt -fast
        abc -dff -D 1    (only if -retime)

    map_ffs:
        dfflegalize -cell $_DFFE_????_ 0 -cell $_SDFFE_????_ 0 -cell $_SDFFCE_????_ 0 -cell $_DLATCH_?_ x
        techmap -D NO_LUT -map +/efinix/cells_map.v
        opt_expr -mux_undef
        simplemap

    map_luts:
        abc -lut 4
        clean

    map_cells:
        techmap -map +/efinix/cells_map.v
        clean

    map_gbuf:
        clkbufmap -buf $__EFX_GBUF O:I
        techmap -map +/efinix/gbuf_map.v
        efinix_fixcarry
        clean

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    edif:
        write_edif <file-name>

    json:
        write_json <file-name>
```

C.190 synth_gatemate – synthesis for Cologne Chip GateMate FPGAs

```
    synth_gatemate [options]

This command runs synthesis for Cologne Chip AG GateMate FPGAs.

    -top <module>
        use the specified module as top module.

    -vlog <file>
        write the design to the specified verilog file. Writing of an output
        file is omitted if this parameter is not specified.

    -json <file>
        write the design to the specified JSON file. Writing of an output file
        is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). An empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis.

    -nobram
        do not use CC_BRAM_20K or CC_BRAM_40K cells in output netlist.

    -noaddf
        do not use CC_ADDF full adder cells in output netlist.

    -nomult
        do not use CC_MULT multiplier cells in output netlist.

    -nomx8, -nomx4
        do not use CC_MX{8,4} multiplexer cells in output netlist.

    -dff
        run 'abc' with -dff option

    -retime
        run 'abc' with '-dff -D 1' options

    -noiopad
        disable I/O buffer insertion (useful for hierarchical or
        out-of-context flows).

    -noclkbuf
        disable automatic clock buffer insertion.

The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib -specify +/gatemate/cells_sim.v +/gatemate/cells_bb.v
        hierarchy -check -top <top>

```

```
    prepare:
        proc
        flatten
        tribuf -logic
        deminout
        opt_expr
        opt_clean
        check
        opt -nodffe -nosdff
        fsm
        opt
        wreduce
        peepopt
        opt_clean
        muxpack
        share
        techmap -map +/cmp2lut.v -D LUT_WIDTH=4
        opt_expr
        opt_clean

    map_mult:    (skip if '-nomult')
        techmap -map +/gatemate/mul_map.v

    coarse:
        alumacc
        opt
        memory -nomap
        opt_clean

    map_bram:    (skip if '-nobram')
        memory_bram -rules +/gatemate/brams.txt
        setundef -zero -params t:$__CC_BRAM_CASCADE t:$__CC_BRAM_40K_SDP t:$__CC_BRAM_20K_SDP t:$__CC_BRAM_
        techmap -map +/gatemate/brams_map.v

    map_ffram:
        opt -fast -mux_undef -undriven -fine
        memory_map
        opt -undriven -fine

    map_gates:
        techmap -map +/techmap.v  -map +/gatemate/arith_map.v
        opt -fast

    map_io:    (skip if '-noiopad')
        iopadmap -bits -inpad CC_IBUF Y:I -outpad CC_OBUF A:O -toutpad CC_TOBUF ~T:A:O -tinoutpad CC_IOBUF
        clean

    map_regs:
        opt_clean
        dfflegalize -cell $_DFFE_????_ x -cell $_DLATCH_???_ x
        techmap -map +/gatemate/reg_map.v
        opt_expr -mux_undef
        simplemap
        opt_clean
```

```

    map_muxs:
        muxcover  -mux4 -mux8
        opt -full
        techmap -map +/gatemate/mux_map.v

    map_luts:
        abc  -dress -lut 4
        clean

    map_cells:
        techmap -map +/gatemate/lut_map.v
        clean

    map_bufg:    (skip if '-noclkbuf')
        clkbufmap -buf CC_BUFG O:I
        clean

    check:
        hierarchy -check
        stat -width
        check -noinit
        blackbox =A:whitebox

    vlog:
        opt_clean -purge
        write_verilog -noattr <file-name>

    json:
        write_json <file-name>
```

C.191 synth_gowin – synthesis for Gowin FPGAs

```
    synth_gowin [options]

This command runs synthesis for Gowin FPGAs. This work is experimental.

    -top <module>
        use the specified module as top module (default='top')

    -vout <file>
        write the design to the specified Verilog netlist file. writing of an
        output file is omitted if this parameter is not specified.

    -json <file>
        write the design to the specified JSON netlist file. writing of an
        output file is omitted if this parameter is not specified.
        This disables features not yet supported by nexpnr-gowin.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
```

```
        synonymous to the end of the command list.

    -nodffe
        do not use flipflops with CE in output netlist

    -nobram
        do not use BRAM cells in output netlist

    -nolutram
        do not use distributed RAM cells in output netlist

    -noflatten
        do not flatten design before synthesis

    -retime
        run 'abc' with '-dff -D 1' options

    -nowidelut
        do not use muxes to implement LUTs larger than LUT4s

    -noiopads
        do not emit IOB at top level ports

    -noalu
        do not use ALU cells

    -abc9
        use new ABC9 flow (EXPERIMENTAL)


The following commands are executed by this synthesis command:

    begin:
        read_verilog -specify -lib +/gowin/cells_sim.v
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten
        tribuf -logic
        deminout

    coarse:
        synth -run coarse

    map_bram:    (skip if -nobram)
        memory_bram -rules +/gowin/brams.txt
        techmap -map +/gowin/brams_map.v

    map_lutram:    (skip if -nolutram)
        memory_bram -rules +/gowin/lutrams.txt
        techmap -map +/gowin/lutrams_map.v
        setundef -params -zero t:RAM16S4

```

```
    map_ffram:
        opt -fast -mux_undef -undriven -fine
        memory_map
        opt -undriven -fine

    map_gates:
        techmap -map +/techmap.v -map +/gowin/arith_map.v
        opt -fast
        abc -dff -D 1    (only if -retime)
        iopadmap -bits -inpad IBUF O:I -outpad OBUF I:O -toutpad TBUF ~OEN:I:O -tinoutpad IOBUF ~OEN:O:I:IO
(unless -noiopads)

    map_ffs:
        opt_clean
        dfflegalize -cell $_DFF_?_ 0 -cell $_DFFE_?P_ 0 -cell $_SDFF_?P?_ r -cell $_SDFFE_?P?P_ r -cell $_D
        techmap -map +/gowin/cells_map.v
        opt_expr -mux_undef
        simplemap

    map_luts:
        abc -lut 4:8
        clean

    map_cells:
        techmap -map +/gowin/cells_map.v
        opt_lut_ins -tech gowin
        setundef -undriven -params -zero
        hilomap -singleton -hicell VCC V -locell GND G
        splitnets -ports    (only if -vout used)
        clean
        autoname

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    vout:
        write_verilog -simple-lhs -decimal -attr2comment -defparam -renameprefix gen <file-name>
        write_json <file-name>
```

C.192 synth_greenpak4 – synthesis for GreenPAK4 FPGAs

```
    synth_greenpak4 [options]

This command runs synthesis for GreenPAK4 FPGAs. This work is experimental.
It is intended to be used with https://github.com/azonenberg/openfpga as the
place-and-route.

    -top <module>
        use the specified module as top module (default='top')
```

```

    -part <part>
        synthesize for the specified part. Valid values are SLG46140V,
        SLG46620V, and SLG46621V (default).

    -json <file>
        write the design to the specified JSON file. writing of an output file
        is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -retime
        run 'abc' with '-dff -D 1' options


The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib +/greenpak4/cells_sim.v
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten
        tribuf -logic

    coarse:
        synth -run coarse

    fine:
        extract_counter -pout GP_DCMP,GP_DAC -maxwidth 14
        clean
        opt -fast -mux_undef -undriven -fine
        memory_map
        opt -undriven -fine
        techmap -map +/techmap.v -map +/greenpak4/cells_latch.v
        dfflibmap -prepare -liberty +/greenpak4/gp_dff.lib
        opt -fast -noclkinv -noff
        abc -dff -D 1    (only if -retime)

    map_luts:
        nlutmap -assert -luts 0,6,8,2     (for -part SLG46140V)
        nlutmap -assert -luts 2,8,16,2    (for -part SLG46620V)
        nlutmap -assert -luts 2,8,16,2    (for -part SLG46621V)
        clean

    map_cells:
        shregmap -tech greenpak4
```

```
        dfflibmap -liberty +/greenpak4/gp_dff.lib
        dffinit -ff GP_DFF Q INIT
        dffinit -ff GP_DFFR Q INIT
        dffinit -ff GP_DFFS Q INIT
        dffinit -ff GP_DFFSR Q INIT
        iopadmap -bits -inpad GP_IBUF OUT:IN -outpad GP_OBUF IN:OUT -inoutpad GP_OBUF OUT:IN -toutpad GP_OB
        attrmvcp -attr src -attr LOC t:GP_OBUF t:GP_OBUFT t:GP_IOBUF n:*
        attrmvcp -attr src -attr LOC -driven t:GP_IBUF n:*
        techmap -map +/greenpak4/cells_map.v
        greenpak4_dffinv
        clean

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    json:
        write_json <file-name>
```

C.193 synth_ice40 – synthesis for iCE40 FPGAs

```
    synth_ice40 [options]

This command runs synthesis for iCE40 FPGAs.

    -device < hx | lp | u >
        relevant only for '-abc9' flow, optimise timing for the specified device.
        default: hx

    -top <module>
        use the specified module as top module

    -blif <file>
        write the design to the specified BLIF file. writing of an output file
        is omitted if this parameter is not specified.

    -edif <file>
        write the design to the specified EDIF file. writing of an output file
        is omitted if this parameter is not specified.

    -json <file>
        write the design to the specified JSON file. writing of an output file
        is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
```

```
        do not flatten design before synthesis

    -dff
        run 'abc'/'abc9' with -dff option

    -retime
        run 'abc' with '-dff -D 1' options

    -nocarry
        do not use SB_CARRY cells in output netlist

    -nodffe
        do not use SB_DFFE* cells in output netlist

    -dffe_min_ce_use <min_ce_use>
        do not use SB_DFFE* cells if the resulting CE line would go to less
        than min_ce_use SB_DFFE* in output netlist

    -nobram
        do not use SB_RAM40_4K* cells in output netlist

    -dsp
        use iCE40 UltraPlus DSP cells for large arithmetic

    -noabc
        use built-in Yosys LUT techmapping instead of abc

    -abc2
        run two passes of 'abc' for slightly improved logic density

    -vpr
        generate an output netlist (and BLIF file) suitable for VPR
        (this feature is experimental and incomplete)

    -abc9
        use new ABC9 flow (EXPERIMENTAL)

    -flowmap
        use FlowMap LUT techmapping instead of abc (EXPERIMENTAL)


The following commands are executed by this synthesis command:

    begin:
        read_verilog -D ICE40_HX -lib -specify +/ice40/cells_sim.v
        hierarchy -check -top <top>
        proc

    flatten:    (unless -noflatten)
        flatten
        tribuf -logic
        deminout

    coarse:
```

```
        opt_expr
        opt_clean
        check
        opt -nodffe -nosdff
        fsm
        opt
        wreduce
        peepopt
        opt_clean
        share
        techmap -map +/cmp2lut.v -D LUT_WIDTH=4
        opt_expr
        opt_clean
        memory_dff
        wreduce t:$mul
        techmap -map +/mul2dsp.v -map +/ice40/dsp_map.v -D DSP_A_MAXWIDTH=16 -D DSP_B_MAXWIDTH=16 -D DSP_A_
(if -dsp)
        select a:mul2dsp                  (if -dsp)
        setattr -unset mul2dsp            (if -dsp)
        opt_expr -fine                    (if -dsp)
        wreduce                           (if -dsp)
        select -clear                     (if -dsp)
        ice40_dsp                         (if -dsp)
        chtype -set $mul t:$__soft_mul    (if -dsp)
        alumacc
        opt
        memory -nomap
        opt_clean

    map_bram:    (skip if -nobram)
        memory_bram -rules +/ice40/brams.txt
        techmap -map +/ice40/brams_map.v
        ice40_braminit

    map_ffram:
        opt -fast -mux_undef -undriven -fine
        memory_map -iattr -attr !ram_block -attr !rom_block -attr logic_block -attr syn_ramstyle=auto -attr
        opt -undriven -fine

    map_gates:
        ice40_wrapcarry
        techmap -map +/techmap.v -map +/ice40/arith_map.v
        opt -fast
        abc -dff -D 1    (only if -retime)
        ice40_opt

    map_ffs:
        dfflegalize -cell $_DFF_?_ 0 -cell $_DFFE_?P_ 0 -cell $_DFF_?P?_ 0 -cell $_DFFE_?P?P_ 0 -cell $_SDF
        techmap -map +/ice40/ff_map.v
        opt_expr -mux_undef
        simplemap
        ice40_opt -full

    map_luts:
```

```
        abc          (only if -abc2)
        ice40_opt    (only if -abc2)
        techmap -map +/ice40/latches_map.v
        simplemap                                  (if -noabc or -flowmap)
        techmap -map +/gate2lut.v -D LUT_WIDTH=4    (only if -noabc)
        flowmap -maxlut 4    (only if -flowmap)
        abc -dress -lut 4     (skip if -noabc)
        ice40_wrapcarry -unwrap
        techmap -map +/ice40/ff_map.v
        clean
        opt_lut -dlogic SB_CARRY:I0=1:I1=2:CI=3 -dlogic SB_CARRY:CO=3

    map_cells:
        techmap -map +/ice40/cells_map.v    (skip if -vpr)
        clean

    check:
        autoname
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    blif:
        opt_clean -purge                                    (vpr mode)
        write_blif -attr -cname -conn -param <file-name>     (vpr mode)
        write_blif -gates -attr -param <file-name>           (non-vpr mode)

    edif:
        write_edif <file-name>

    json:
        write_json <file-name>
```

C.194 synth_intel – synthesis for Intel (Altera) FPGAs.

```
    synth_intel [options]

This command runs synthesis for Intel FPGAs.

    -family <max10 | cyclone10lp | cycloneiv | cycloneive>
        generate the synthesis netlist for the specified family.
        MAX10 is the default target if no family argument specified.
        For Cyclone IV GX devices, use cycloneiv argument; for Cyclone IV E, use cycloneive.
        For Cyclone V and Cyclone 10 GX, use the synth_intel_alm backend instead.

    -top <module>
        use the specified module as top module (default='top')

    -vqm <file>
        write the design to the specified Verilog Quartus Mapping File. Writing of an
        output file is omitted if this parameter is not specified.
```

```
        Note that this backend has not been tested and is likely incompatible
        with recent versions of Quartus.

    -vpr <file>
        write BLIF files for VPR flow experiments. The synthesized BLIF output file is not
        compatible with the Quartus flow. Writing of an
        output file is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -iopads
        use IO pad cells in output netlist

    -nobram
        do not use block RAM cells in output netlist

    -noflatten
        do not flatten design before synthesis

    -retime
        run 'abc' with '-dff -D 1' options

The following commands are executed by this synthesis command:

    begin:

    family:
        read_verilog -sv -lib +/intel/max10/cells_sim.v
        read_verilog -sv -lib +/intel/common/m9k_bb.v
        read_verilog -sv -lib +/intel/common/altpll_bb.v
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten
        tribuf -logic
        deminout

    coarse:
        synth -run coarse

    map_bram:    (skip if -nobram)
        memory_bram -rules +/intel/common/brams_m9k.txt    (if applicable for family)
        techmap -map +/intel/common/brams_map_m9k.v    (if applicable for family)

    map_ffram:
        opt -fast -mux_undef -undriven -fine -full
        memory_map
        opt -undriven -fine
        techmap -map +/techmap.v
        opt -full
```

```
        clean -purge
        setundef -undriven -zero
        abc -markgroups -dff -D 1    (only if -retime)

    map_ffs:
        dfflegalize -cell $_DFFE_PNOP_ 01
        techmap -map +/intel/common/ff_map.v

    map_luts:
        abc -lut 4
        clean

    map_cells:
        iopadmap -bits -outpad $__outpad I:O -inpad $__inpad O:I    (if -iopads)
        techmap -map +/intel/max10/cells_map.v
        clean -purge

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    vqm:
        write_verilog -attr2comment -defparam -nohex -decimal -renameprefix syn_ <file-name>

    vpr:
        opt_clean -purge
        write_blif <file-name>


WARNING: THE 'synth_intel' COMMAND IS EXPERIMENTAL.
```

C.195 synth_intel_alm – synthesis for ALM-based Intel (Altera) FPGAs.

```
    synth_intel_alm [options]

This command runs synthesis for ALM-based Intel FPGAs.

    -top <module>
        use the specified module as top module

    -family <family>
        target one of:
        "cyclonev"    - Cyclone V (default)
        "arriav"      - Arria V (non-GZ)        "cyclone10gx" - Cyclone 10GX

    -vqm <file>
        write the design to the specified Verilog Quartus Mapping File. Writing of an
        output file is omitted if this parameter is not specified. Implies -quartus.

```

```
    -noflatten
        do not flatten design before synthesis; useful for per-module area statistics

    -quartus
        output a netlist using Quartus cells instead of MISTRAL_* cells

    -dff
        pass DFFs to ABC to perform sequential logic optimisations (EXPERIMENTAL)

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -nolutram
        do not use LUT RAM cells in output netlist

    -nobram
        do not use block RAM cells in output netlist

    -nodsp
        do not map multipliers to MISTRAL_MUL cells

    -noiopad
        do not instantiate IO buffers

    -noclkbuf
        do not insert global clock buffers

The following commands are executed by this synthesis command:

    begin:
        read_verilog -specify -lib -D <family> +/intel_alm/common/alm_sim.v
        read_verilog -specify -lib -D <family> +/intel_alm/common/dff_sim.v
        read_verilog -specify -lib -D <family> +/intel_alm/common/dsp_sim.v
        read_verilog -specify -lib -D <family> +/intel_alm/common/mem_sim.v
        read_verilog -specify -lib -D <family> +/intel_alm/common/misc_sim.v
        read_verilog -specify -lib -D <family> -icells +/intel_alm/common/abc9_model.v
        read_verilog -lib +/intel/common/altpll_bb.v
        read_verilog -lib +/intel_alm/common/megafunction_bb.v
        hierarchy -check -top <top>

    coarse:
        proc
        flatten    (skip if -noflatten)
        tribuf -logic
        deminout
        opt_expr
        opt_clean
        check
        opt -nodffe -nosdff
        fsm
        opt
        wreduce
```

```
        peepopt
        opt_clean
        share
        techmap -map +/cmp2lut.v -D LUT_WIDTH=6
        opt_expr
        opt_clean
        techmap -map +/mul2dsp.v [...]    (unless -nodsp)
        alumacc
        iopadmap -bits -outpad MISTRAL_OB I:PAD -inpad MISTRAL_IB O:PAD -toutpad MISTRAL_IO OE:O:PAD -tinou
(unless -noiopad)
        techmap -map +/intel_alm/common/arith_alm_map.v -map +/intel_alm/common/dsp_map.v
        opt
        memory -nomap
        opt_clean

    map_bram:    (skip if -nobram)
        memory_bram -rules +/intel_alm/common/bram_<bram_type>.txt
        techmap -map +/intel_alm/common/bram_<bram_type>_map.v

    map_lutram:    (skip if -nolutram)
        memory_bram -rules +/intel_alm/common/lutram_mlab.txt    (for Cyclone V / Cyclone 10GX)

    map_ffram:
        memory_map
        opt -full

    map_ffs:
        techmap
        dfflegalize -cell $_DFFE_PNOP_ 0 -cell $_SDFFCE_PPOP_ 0
        techmap -map +/intel_alm/common/dff_map.v
        opt -full -undriven -mux_undef
        clean -purge
        clkbufmap -buf MISTRAL_CLKBUF Q:A    (unless -noclkbuf)

    map_luts:
        techmap -map +/intel_alm/common/abc9_map.v
        abc9 [-dff] -maxlut 6 -W 600
        techmap -map +/intel_alm/common/abc9_unmap.v
        techmap -map +/intel_alm/common/alm_map.v
        opt -fast
        autoname
        clean

    check:
        hierarchy -check
        stat
        check
        blackbox =A:whitebox

    quartus:
        rename -hide w:*[* w:*]*
        setundef -zero
        hilomap -singleton -hicell __MISTRAL_VCC Q -locell __MISTRAL_GND Q
        techmap -D <family> -map +/intel_alm/common/quartus_rename.v
```

```

    vqm:
        write_verilog -attr2comment -defparam -nohex -decimal <file-name>
```

C.196 synth_machxo2 – synthesis for MachXO2 FPGAs. This work is experimental.

```
    synth_machxo2 [options]

This command runs synthesis for MachXO2 FPGAs.

    -top <module>
        use the specified module as top module

    -blif <file>
        write the design to the specified BLIF file. writing of an output file
        is omitted if this parameter is not specified.

    -edif <file>
        write the design to the specified EDIF file. writing of an output file
        is omitted if this parameter is not specified.

    -json <file>
        write the design to the specified JSON file. writing of an output file
        is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -noiopad
        do not insert IO buffers

    -vpr
        generate an output netlist (and BLIF file) suitable for VPR
        (this feature is experimental and incomplete)


The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib -icells +/machxo2/cells_sim.v
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten
        tribuf -logic
```

```
        deminout

    coarse:
        synth -run coarse

    fine:
        memory_map
        opt -full
        techmap -map +/techmap.v
        opt -fast

    map_ios:    (unless -noiopad)
        iopadmap -bits -outpad $__FACADE_OUTPAD I:O -inpad $__FACADE_INPAD O:I -toutpad $__FACADE_TOUTPAD ~
        attrmvcp -attr src -attr LOC t:$__FACADE_OUTPAD %x:+[O] t:$__FACADE_TOUTPAD %x:+[O] t:$__FACADE_TIN
        attrmvcp -attr src -attr LOC -driven t:$__FACADE_INPAD %x:+[I]

    map_ffs:
        dfflegalize -cell $_DFF_P_ 0

    map_luts:
        abc -lut 4 -dress
        clean

    map_cells:
        techmap -map +/machxo2/cells_map.v
        clean

    check:
        hierarchy -check
        stat
        blackbox =A:whitebox

    blif:
        opt_clean -purge                                     (vpr mode)
        write_blif -attr -cname -conn -param <file-name>     (vpr mode)
        write_blif -gates -attr -param <file-name>           (non-vpr mode)

    edif:
        write_edif <file-name>

    json:
        write_json <file-name>
```

C.197 synth_nexus – synthesis for Lattice Nexus FPGAs

```
    synth_nexus [options]

This command runs synthesis for Lattice Nexus FPGAs.

    -top <module>
        use the specified module as top module

```

```
    -family <device>
        run synthesis for the specified Nexus device
        supported values: lifcl, lfd2nx

    -json <file>
        write the design to the specified JSON file. writing of an output file
        is omitted if this parameter is not specified.

    -vm <file>
        write the design to the specified structural Verilog file. writing of
        an output file is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -dff
        run 'abc'/'abc9' with -dff option

    -retime
        run 'abc' with '-dff -D 1' options

    -noccu2
        do not use CCU2 cells in output netlist

    -nodffe
        do not use flipflops with CE in output netlist

    -nolram
        do not use large RAM cells in output netlist
        note that large RAM must be explicitly requested with a (* lram *)
        attribute on the memory.

    -nobram
        do not use block RAM cells in output netlist

    -nolutram
        do not use LUT RAM cells in output netlist

    -nowidelut
        do not use PFU muxes to implement LUTs larger than LUT4s

    -noiopad
        do not insert IO buffers

    -nodsp
        do not infer DSP multipliers

    -abc9
        use new ABC9 flow (EXPERIMENTAL)
```

```

The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib -specify +/nexus/cells_sim.v +/nexus/cells_xtra.v
        hierarchy -check -top <top>

    coarse:
        proc
        flatten
        tribuf -logic
        deminout
        opt_expr
        opt_clean
        check
        opt -nodffe -nosdff
        fsm
        opt
        wreduce
        peepopt
        opt_clean
        share
        techmap -map +/cmp2lut.v -D LUT_WIDTH=4
        opt_expr
        opt_clean
        techmap -map +/mul2dsp.v [...]    (unless -nodsp)
        techmap -map +/nexus/dsp_map.v    (unless -nodsp)
        alumacc
        opt
        memory -nomap
        opt_clean

    map_lram:    (skip if -nolram)
        memory_bram -rules +/nexus/lrams.txt
        setundef -zero -params t:$__NX_PDPSC512K
        techmap -map +/nexus/lrams_map.v

    map_bram:    (skip if -nobram)
        memory_bram -rules +/nexus/brams.txt
        setundef -zero -params t:$__NX_PDP16K
        techmap -map +/nexus/brams_map.v

    map_lutram:    (skip if -nolutram)
        memory_bram -rules +/nexus/lutrams.txt
        setundef -zero -params t:$__NEXUS_DPR16X4
        techmap -map +/nexus/lutrams_map.v

    map_ffram:
        opt -fast -mux_undef -undriven -fine
        memory_map -iattr -attr !ram_block -attr !rom_block -attr logic_block -attr syn_ramstyle=auto -attr
        opt -undriven -fine

    map_gates:
        techmap -map +/techmap.v -map +/nexus/arith_map.v
```

```
        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')
        opt -fast
        abc -dff -D 1    (only if -retime)

    map_ffs:
        opt_clean
        dfflegalize -cell $_DFF_P_ 01 -cell $_DFF_PP?_ r -cell $_SDFF_PP?_ r -cell $_DLATCH_?_ x [-cell $_D
($_*DFFE_* only if not -nodffe)
        zinit -all w:* t:$_DFF_?_ t:$_DFFE_??_ t:$_SDFF*    (only if -abc9 and -dff
        techmap -D NO_LUT -map +/nexus/cells_map.v
        opt_expr -undriven -mux_undef
        simplemap
        attrmvcp -copy -attr syn_useioff
        opt_clean

    map_luts:
        techmap -map +/nexus/latches_map.v
        abc -dress -lut 4:5
        clean

    map_cells:
        techmap -map +/nexus/cells_map.v
        setundef -zero
        hilomap -singleton -hicell VHI Z -locell VLO Z
        clean

    check:
        autoname
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    json:
        write_json <file-name>

    vm:
        write_verilog <file-name>
```

C.198 synth_quicklogic – Synthesis for QuickLogic FPGAs

```
   synth_quicklogic [options]
This command runs synthesis for QuickLogic FPGAs

    -top <module>
         use the specified module as top module

    -family <family>
        run synthesis for the specified QuickLogic architecture
        generate the synthesis netlist for the specified family.
        supported values:
```

```
        - pp3: PolarPro 3

    -blif <file>
        write the design to the specified BLIF file. writing of an output file
        is omitted if this parameter is not specified.

    -verilog <file>
        write the design to the specified verilog file. writing of an output file
        is omitted if this parameter is not specified.

    -abc
        use old ABC flow, which has generally worse mapping results but is less
        likely to have bugs.

The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib -specify +/quicklogic/cells_sim.v +/quicklogic/pp3_cells_sim.v
        read_verilog -lib -specify +/quicklogic/lut_sim.v
        hierarchy -check -top <top>

    coarse:
        proc
        flatten
        tribuf -logic
        deminout
        opt_expr
        opt_clean
        check
        opt -nodffe -nosdff
        fsm
        opt
        wreduce
        peepopt
        opt_clean
        share
        techmap -map +/cmp2lut.v -D LUT_WIDTH=4
        opt_expr
        opt_clean
        alumacc
        pmuxtree
        opt
        memory -nomap
        opt_clean

    map_ffram:
        opt -fast -mux_undef -undriven -fine
        memory_map -iattr -attr !ram_block -attr !rom_block -attr logic_block -attr syn_ramstyle=auto -attr
        opt -undriven -fine

    map_gates:
        techmap
        opt -fast
        muxcover -mux8 -mux4
```

```

    map_ffs:
        opt_expr
        dfflegalize -cell $_DFFSRE_PPPP_ 0 -cell $_DLATCH_?_ x
        techmap -map +/quicklogic/pp3_cells_map.v -map +/quicklogic/pp3_ffs_map.v
        opt_expr -mux_undef

    map_luts:
        techmap -map +/quicklogic/pp3_latches_map.v
        read_verilog -lib -specify -icells +/quicklogic/abc9_model.v
        techmap -map +/quicklogic/abc9_map.v
        abc9 -maxlut 4 -dff
        techmap -map +/quicklogic/abc9_unmap.v
        clean

    map_cells:
        techmap -map +/quicklogic/pp3_lut_map.v
        clean

    check:
        autoname
        hierarchy -check
        stat
        check -noinit

    iomap:
        clkbufmap -inpad ckpad Q:P
        iopadmap -bits -outpad outpad A:P -inpad inpad Q:P -tinoutpad bipad EN:Q:A:P A:top

    finalize:
        setundef -zero -params -undriven
        hilomap -hicell logic_1 A -locell logic_0 A -singleton A:top
        opt_clean -purge
        check
        blackbox =A:whitebox

    blif:
        write_blif -attr -param -auto-top

    verilog:
        write_verilog -noattr -nohex <file-name>
```

C.199 synth_sf2 – synthesis for SmartFusion2 and IGLOO2 FPGAs

```
    synth_sf2 [options]

This command runs synthesis for SmartFusion2 and IGLOO2 FPGAs.

    -top <module>
        use the specified module as top module

```

```
    -edif <file>
        write the design to the specified EDIF file. writing of an output file
        is omitted if this parameter is not specified.

    -vlog <file>
        write the design to the specified Verilog file. writing of an output file
        is omitted if this parameter is not specified.

    -json <file>
        write the design to the specified JSON file. writing of an output file
        is omitted if this parameter is not specified.

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -noflatten
        do not flatten design before synthesis

    -noiobs
        run synthesis in "block mode", i.e. do not insert IO buffers

    -clkbuf
        insert direct PAD->global_net buffers

    -retime
        run 'abc' with '-dff -D 1' options


The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib +/sf2/cells_sim.v
        hierarchy -check -top <top>

    flatten:    (unless -noflatten)
        proc
        flatten
        tribuf -logic
        deminout

    coarse:
        synth -run coarse

    fine:
        opt -fast -mux_undef -undriven -fine
        memory_map
        opt -undriven -fine
        techmap -map +/techmap.v -map +/sf2/arith_map.v
        opt -fast
        abc -dff -D 1    (only if -retime)

    map_ffs:
```

```
        dfflegalize -cell $_DFFE_PN?P_ x -cell $_SDFFCE_PN?P_ x -cell $_DLATCH_PN?_ x
        techmap -D NO_LUT -map +/sf2/cells_map.v
        opt_expr -mux_undef
        simplemap

    map_luts:
        abc -lut 4
        clean

    map_cells:
        techmap -map +/sf2/cells_map.v
        clean

    map_iobs:
        clkbufmap -buf CLKINT Y:A [-inpad CLKBUF Y:PAD]    (unless -noiobs, -inpad only passed if -clkbuf)
        iopadmap -bits -inpad INBUF Y:PAD -outpad OUTBUF D:PAD -toutpad TRIBUFF E:D:PAD -tinoutpad BIBUF E:
(unless -noiobs
        clean

    check:
        hierarchy -check
        stat
        check -noinit
        blackbox =A:whitebox

    edif:
        write_edif -gndvccy <file-name>

    vlog:
        write_verilog <file-name>

    json:
        write_json <file-name>
```

C.200 synth_xilinx – synthesis for Xilinx FPGAs

```
    synth_xilinx [options]

This command runs synthesis for Xilinx FPGAs. This command does not operate on
partly selected designs. At the moment this command creates netlists that are
compatible with 7-Series Xilinx devices.

    -top <module>
        use the specified module as top module

    -family <family>
        run synthesis for the specified Xilinx architecture
        generate the synthesis netlist for the specified family.
        supported values:
        - xcup: Ultrascale Plus
        - xcu: Ultrascale
        - xc7: Series 7 (default)
```

```
        - xc6s: Spartan 6
        - xc6v: Virtex 6
        - xc5v: Virtex 5 (EXPERIMENTAL)
        - xc4v: Virtex 4 (EXPERIMENTAL)
        - xc3sda: Spartan 3A DSP (EXPERIMENTAL)
        - xc3sa: Spartan 3A (EXPERIMENTAL)
        - xc3se: Spartan 3E (EXPERIMENTAL)
        - xc3s: Spartan 3 (EXPERIMENTAL)
        - xc2vp: Virtex 2 Pro (EXPERIMENTAL)
        - xc2v: Virtex 2 (EXPERIMENTAL)
        - xcve: Virtex E, Spartan 2E (EXPERIMENTAL)
        - xcv: Virtex, Spartan 2 (EXPERIMENTAL)

    -edif <file>
        write the design to the specified edif file. writing of an output file
        is omitted if this parameter is not specified.

    -blif <file>
        write the design to the specified BLIF file. writing of an output file
        is omitted if this parameter is not specified.

    -ise
        generate an output netlist suitable for ISE

    -nobram
        do not use block RAM cells in output netlist

    -nolutram
        do not use distributed RAM cells in output netlist

    -nosrl
        do not use distributed SRL cells in output netlist

    -nocarry
        do not use XORCY/MUXCY/CARRY4 cells in output netlist

    -nowidelut
        do not use MUXF[5-9] resources to implement LUTs larger than native for the target

    -nodsp
        do not use DSP48*s to implement multipliers and associated logic

    -noiopad
        disable I/O buffer insertion (useful for hierarchical or
        out-of-context flows)

    -noclkbuf
        disable automatic clock buffer insertion

    -uram
        infer URAM288s for large memories (xcup only)

    -widemux <int>
        enable inference of hard multiplexer resources (MUXF[78]) for muxes at or
```

```
        above this number of inputs (minimum value 2, recommended value >= 5).
        default: 0 (no inference)

    -run <from_label>:<to_label>
        only run the commands between the labels (see below). an empty
        from label is synonymous to 'begin', and empty to label is
        synonymous to the end of the command list.

    -flatten
        flatten design before synthesis

    -dff
        run 'abc'/'abc9' with -dff option

    -retime
        run 'abc' with '-D 1' option to enable flip-flop retiming.
        implies -dff.

    -abc9
        use new ABC9 flow (EXPERIMENTAL)


The following commands are executed by this synthesis command:

    begin:
        read_verilog -lib -specify +/xilinx/cells_sim.v
        read_verilog -lib +/xilinx/cells_xtra.v
        hierarchy -check -auto-top

    prepare:
        proc
        flatten    (with '-flatten')
        tribuf -logic
        deminout
        opt_expr
        opt_clean
        check
        opt -nodffe -nosdff
        fsm
        opt
        wreduce [-keepdc]    (option for '-widemux')
        peepopt
        opt_clean
        muxpack        ('-widemux' only)
        pmux2shiftx    (skip if '-nosrl' and '-widemux=0')
        clean          (skip if '-nosrl' and '-widemux=0')

    map_dsp:    (skip if '-nodsp')
        memory_dff
        techmap -map +/mul2dsp.v -map +/xilinx/{family}_dsp_map.v {options}
        select a:mul2dsp
        setattr -unset mul2dsp
        opt_expr -fine
        wreduce
```

```
        select -clear
        xilinx_dsp -family <family>
        chtype -set $mul t:$__soft_mul

    coarse:
        techmap -map +/cmp2lut.v -map +/cmp2lcu.v -D LUT_WIDTH=[46]
        alumacc
        share
        opt
        memory -nomap
        opt_clean

    map_uram:    (only if '-uram')
        memory_bram -rules +/xilinx/{family}_urams.txt
        techmap -map +/xilinx/{family}_urams_map.v

    map_bram:    (skip if '-nobram')
        memory_bram -rules +/xilinx/{family}_brams.txt
        techmap -map +/xilinx/{family}_brams_map.v

    map_lutram:    (skip if '-nolutram')
        memory_bram -rules +/xilinx/lut[46]_lutrams.txt
        techmap -map +/xilinx/lutrams_map.v

    map_ffram:
        opt -fast -full
        memory_map

    fine:
        simplemap t:$mux    ('-widemux' only)
        muxcover <internal options>    ('-widemux' only)
        opt -full
        xilinx_srl -variable -minlen 3    (skip if '-nosrl')
        techmap  -map +/techmap.v -D LUT_SIZE=[46] [-map +/xilinx/mux_map.v] -map +/xilinx/arith_map.v
        opt -fast

    map_cells:
        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')
        techmap -map +/techmap.v -map +/xilinx/cells_map.v
        clean

    map_ffs:
        dfflegalize -cell $_DFFE_?P?P_ 01 -cell $_SDFFE_?P?P_ 01 -cell $_DLATCH_?P?_ 01    (for xc6v, xc7,
        zinit -all w:* t:$_SDFFE_*    ('-dff' only)
        techmap -map +/xilinx/ff_map.v    ('-abc9' only)

    map_luts:
        opt_expr -mux_undef -noclkinv
        abc -luts 2:2,3,6:5[,10,20] [-dff] [-D 1]    (option for '-nowidelut', '-dff', '-retime')
        clean
        techmap -map +/xilinx/ff_map.v    (only if not '-abc9')
        xilinx_srl -fixed -minlen 3    (skip if '-nosrl')
        techmap -map +/xilinx/lut_map.v -map +/xilinx/cells_map.v -D LUT_WIDTH=[46]
```

```
        xilinx_dffopt [-lut4]
        opt_lut_ins -tech xilinx

    finalize:
        clkbufmap -buf BUFG O:I    (skip if '-noclkbuf')
        extractinv -inv INV O:I    (only if '-ise')
        clean

    check:
        hierarchy -check
        stat -tech xilinx
        check -noinit
        blackbox =A:whitebox

    edif:
        write_edif -pvector bra

    blif:
        write_blif
```

C.201 tcl – execute a TCL script file

```
    tcl <filename> [args]

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

The tcl command 'yosys -import' can be used to import all yosys
commands directly as tcl commands to the tcl shell. Yosys commands
'proc' and 'rename' are wrapped to tcl commands 'procs' and 'renames'
in order to avoid a name collision with the built in commands.

If any arguments are specified, these arguments are provided to the script via
the standard $argc and $argv variables.
```

C.202 techmap – generic technology mapper

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

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

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

    -map %<design-name>
```

```
        like -map above, but with an in-memory design instead of a file.

    -extern
        load the cell implementations as separate modules into the design
        instead of inlining them.

    -max_iter <number>
        only run the specified number of iterations on each module.
        default: unlimited

    -recursive
        instead of the iterative breadth-first algorithm use a recursive
        depth-first algorithm. both methods should yield equivalent results,
        but may differ in performance.

    -autoproc
        Automatically call "proc" on implementations that contain processes.

    -wb
        Ignore the 'whitebox' attribute on cell implementations.

    -assert
        this option will cause techmap to exit with an error if it can't map
        a selected cell. only cell types that end on an underscore are accepted
        as final cell types by this mode.

    -D <define>, -I <incdir>
        this options are passed as-is to the Verilog frontend for loading the
        map file. Note that the Verilog frontend is also called with the
        '-nooverwrite' option set.

When a module in the map file has the 'techmap_celltype' attribute set, it will
match cells with a type that match the text value of this attribute. Otherwise
the module name will be used to match the cell.  Multiple space-separated cell
types can be listed, and wildcards using [] will be expanded (ie. "$_DFF_[PN]_"
is the same as "$_DFF_P_ $_DFF_N_").

When a module in the map file has the 'techmap_simplemap' attribute set, techmap
will use 'simplemap' (see 'help simplemap') to map cells matching the module.

When a module in the map file has the 'techmap_maccmap' attribute set, techmap
will use 'maccmap' (see 'help maccmap') to map cells matching the module.

When a module in the map file has the 'techmap_wrap' attribute set, techmap
will create a wrapper for the cell and then run the command string that the
attribute is set to on the wrapper module.

When a port on a module in the map file has the 'techmap_autopurge' attribute
set, and that port is not connected in the instantiation that is mapped, then
then a cell port connected only to such wires will be omitted in the mapped
version of the circuit.

All wires in the modules from the map file matching the pattern _TECHMAP_*
or *._TECHMAP_* are special wires that are used to pass instructions from
```

```
the mapping module to the techmap command. At the moment the following special
wires are supported:

    _TECHMAP_FAIL_
        When this wire is set to a non-zero constant value, techmap will not
        use this module and instead try the next module with a matching
        'techmap_celltype' attribute.

        When such a wire exists but does not have a constant value after all
        _TECHMAP_DO_* commands have been executed, an error is generated.

    _TECHMAP_DO_*
        This wires are evaluated in alphabetical order. The constant text value
        of this wire is a yosys command (or sequence of commands) that is run
        by techmap on the module. A common use case is to run 'proc' on modules
        that are written using always-statements.

        When such a wire has a non-constant value at the time it is to be
        evaluated, an error is produced. That means it is possible for such a
        wire to start out as non-constant and evaluate to a constant value
        during processing of other _TECHMAP_DO_* commands.

        A _TECHMAP_DO_* command may start with the special token 'CONSTMAP; '.
        in this case techmap will create a copy for each distinct configuration
        of constant inputs and shorted inputs at this point and import the
        constant and connected bits into the map module. All further commands
        are executed in this copy. This is a very convenient way of creating
        optimized specializations of techmap modules without using the special
        parameters described below.

        A _TECHMAP_DO_* command may start with the special token 'RECURSION; '.
        then techmap will recursively replace the cells in the module with their
        implementation. This is not affected by the -max_iter option.

        It is possible to combine both prefixes to 'RECURSION; CONSTMAP; '.

    _TECHMAP_REMOVEINIT_<port-name>_
        When this wire is set to a constant value, the init attribute of the wire(s)
        connected to this port will be consumed.  This wire must have the same
        width as the given port, and for every bit that is set to 1 in the value,
        the corresponding init attribute bit will be changed to 1'bx.  If all
        bits of an init attribute are left as x, it will be removed.

In addition to this special wires, techmap also supports special parameters in
modules in the map file:

    _TECHMAP_CELLTYPE_
        When a parameter with this name exists, it will be set to the type name
        of the cell that matches the module.

    _TECHMAP_CELLNAME_
        When a parameter with this name exists, it will be set to the name
        of the cell that matches the module.

```

```
    _TECHMAP_CONSTMSK_<port-name>_
    _TECHMAP_CONSTVAL_<port-name>_
        When this pair of parameters is available in a module for a port, then
        former has a 1-bit for each constant input bit and the latter has the
        value for this bit. The unused bits of the latter are set to undef (x).

    _TECHMAP_WIREINIT_<port-name>_
        When a parameter with this name exists, it will be set to the initial
        value of the wire(s) connected to the given port, as specified by the init
        attribute. If the attribute doesn't exist, x will be filled for the
        missing bits.  To remove the init attribute bits used, use the
        _TECHMAP_REMOVEINIT_*_ wires.

    _TECHMAP_BITS_CONNMAP_
    _TECHMAP_CONNMAP_<port-name>_
        For an N-bit port, the _TECHMAP_CONNMAP_<port-name>_ parameter, if it
        exists, will be set to an N*_TECHMAP_BITS_CONNMAP_ bit vector containing
        N words (of _TECHMAP_BITS_CONNMAP_ bits each) that assign each single
        bit driver a unique id. The values 0-3 are reserved for 0, 1, x, and z.
        This can be used to detect shorted inputs.

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

A cell with the name _TECHMAP_REPLACE_ in the map file will inherit the name
and attributes of the cell that is being replaced.
A cell with a name of the form `_TECHMAP_REPLACE_.<suffix>` in the map file will
be named thus but with the `_TECHMAP_REPLACE_' prefix substituted with the name
of the cell being replaced.
Similarly, a wire named in the form `_TECHMAP_REPLACE_.<suffix>` will cause a
new wire alias to be created and named as above but with the `_TECHMAP_REPLACE_'
prefix also substituted.

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

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

C.203 tee – redirect command output to file

```
    tee [-q] [-o logfile|-a logfile] cmd

Execute the specified command, optionally writing the commands output to the
specified logfile(s).

    -q
        Do not print output to the normal destination (console and/or log file).

    -o logfile
        Write output to this file, truncate if exists.
```

```

    -a logfile
        Write output to this file, append if exists.

    +INT, -INT
        Add/subtract INT from the -v setting for this command.
```

C.204 test_abcloop – automatically test handling of loops in abc command

```
    test_abcloop [options]

Test handling of logic loops in ABC.

    -n {integer}
        create this number of circuits and test them (default = 100).

    -s {positive_integer}
        use this value as rng seed value (default = unix time).
```

C.205 test_autotb – generate simple test benches

```
    test_autotb [options] [filename]

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

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

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

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

The attribute 'gentb_skip' can be attached to modules to suppress testbench
generation.

    -n <int>
        number of iterations the test bench should run (default = 1000)

    -seed <int>
        seed used for pseudo-random number generation (default = 0).
        a value of 0 will cause an arbitrary seed to be chosen, based on
        the current system time.
```

C.206 test_cell – automatically test the implementation of a cell type

```
    test_cell [options] {cell-types}

Tests the internal implementation of the given cell type (for example '$add')
by comparing SAT solver, EVAL and TECHMAP implementations of the cell types..

Run with 'all' instead of a cell type to run the test on all supported
cell types. Use for example 'all /$add' for all cell types except $add.

    -n {integer}
        create this number of cell instances and test them (default = 100).

    -s {positive_integer}
        use this value as rng seed value (default = unix time).

    -f {rtlil_file}
        don't generate circuits. instead load the specified RTLIL file.

    -w {filename_prefix}
        don't test anything. just generate the circuits and write them
        to RTLIL files with the specified prefix

    -map {filename}
        pass this option to techmap.

    -simlib
        use "techmap -D SIMLIB_NOCHECKS -map +/simlib.v -max_iter 2 -autoproc"

    -aigmap
        instead of calling "techmap", call "aigmap"

    -muxdiv
        when creating test benches with dividers, create an additional mux
        to mask out the division-by-zero case

    -script {script_file}
        instead of calling "techmap", call "script {script_file}".

    -const
        set some input bits to random constant values

    -nosat
        do not check SAT model or run SAT equivalence checking

    -noeval
        do not check const-eval models

    -edges
        test cell edges db creator against sat-based implementation

    -v
```

```
        print additional debug information to the console

    -vlog {filename}
        create a Verilog test bench to test simlib and write_verilog
```

C.207 test_pmgen – test pass for pmgen

```
    test_pmgen -reduce_chain [options] [selection]

Demo for recursive pmgen patterns. Map chains of AND/OR/XOR to $reduce_*.


    test_pmgen -reduce_tree [options] [selection]

Demo for recursive pmgen patterns. Map trees of AND/OR/XOR to $reduce_*.


    test_pmgen -eqpmux [options] [selection]

Demo for recursive pmgen patterns. Optimize EQ/NE/PMUX circuits.


    test_pmgen -generate [options] <pattern_name>

Create modules that match the specified pattern.
```

C.208 torder – print cells in topological order

```
    torder [options] [selection]

This command prints the selected cells in topological order.

    -stop <cell_type> <cell_port>
        do not use the specified cell port in topological sorting

    -noautostop
        by default Q outputs of internal FF cells and memory read port outputs
        are not used in topological sorting. this option deactivates that.
```

C.209 trace – redirect command output to file

```
    trace cmd

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

C.210 tribuf – infer tri-state buffers

```
    tribuf [options] [selection]

This pass transforms $mux cells with 'z' inputs to tristate buffers.

    -merge
        merge multiple tri-state buffers driving the same net
        into a single buffer.

    -logic
        convert tri-state buffers that do not drive output ports
        to non-tristate logic. this option implies -merge.
```

C.211 uniquify – create unique copies of modules

```
    uniquify [selection]

By default, a module that is instantiated by several other modules is only
kept once in the design. This preserves the original modularity of the design
and reduces the overall size of the design in memory. But it prevents certain
optimizations and other operations on the design. This pass creates unique
modules for all selected cells. The created modules are marked with the
'unique' attribute.

This commands only operates on modules that by themself have the 'unique'
attribute set (the 'top' module is unique implicitly).
```

C.212 verific – load Verilog and VHDL designs using Verific

```
    verific {-vlog95|-vlog2k|-sv2005|-sv2009|-sv2012|-sv} <verilog-file>..

Load the specified Verilog/SystemVerilog files into Verific.

All files specified in one call to this command are one compilation unit.
Files passed to different calls to this command are treated as belonging to
different compilation units.

Additional -D<macro>[=<value>] options may be added after the option indicating
the language version (and before file names) to set additional verilog defines.
The macros YOSYS, SYNTHESIS, and VERIFIC are defined implicitly.



    verific -formal <verilog-file>..

Like -sv, but define FORMAL instead of SYNTHESIS.



    verific {-f|-F} [-vlog95|-vlog2k|-sv2005|-sv2009|-sv2012|-sv|-formal] <command-file>
```

```

Load and execute the specified command file.
Override verilog parsing mode can be set.
The macros YOSYS, SYNTHESIS/FORMAL, and VERIFIC are defined implicitly.

Command file parser supports following commands:
    +define    - defines macro
    -u         - upper case all identifier (makes Verilog parser case insensitive)
    -v         - register library name (file)
    -y         - register library name (directory)
    +incdir    - specify include dir
    +libext    - specify library extension
    +liborder  - add library in ordered list
    +librescan - unresolved modules will be always searched starting with the first
                 library specified by -y/-v options.
    -f/-file   - nested -f option
    -F         - nested -F option

    parse mode:
        -ams
        +systemverilogext
        +v2k
        +verilog1995ext
        +verilog2001ext
        -sverilog


    verific [-work <libname>] {-sv|-vhdl|...} <hdl-file>

Load the specified Verilog/SystemVerilog/VHDL file into the specified library.
(default library when -work is not present: "work")


    verific [-L <libname>] {-sv|-vhdl|...} <hdl-file>

Look up external definitions in the specified library.
(-L may be used more than once)


    verific -vlog-incdir <directory>..

Add Verilog include directories.


    verific -vlog-libdir <directory>..

Add Verilog library directories. Verific will search in this directories to
find undefined modules.


    verific -vlog-define <macro>[=<value>]..

Add Verilog defines.

```

```

    verific -vlog-undef <macro>..

Remove Verilog defines previously set with -vlog-define.


    verific -set-error <msg_id>..
    verific -set-warning <msg_id>..
    verific -set-info <msg_id>..
    verific -set-ignore <msg_id>..

Set message severity. <msg_id> is the string in square brackets when a message
is printed, such as VERI-1209.


    verific -import [options] <top-module>..

Elaborate the design for the specified top modules, import to Yosys and
reset the internal state of Verific.

Import options:

  -all
    Elaborate all modules, not just the hierarchy below the given top
    modules. With this option the list of modules to import is optional.

  -gates
    Create a gate-level netlist.

  -flatten
    Flatten the design in Verific before importing.

  -extnets
    Resolve references to external nets by adding module ports as needed.

  -autocover
    Generate automatic cover statements for all asserts

  -fullinit
    Keep all register initializations, even those for non-FF registers.

  -chparam name value
    Elaborate the specified top modules (all modules when -all given) using
    this parameter value. Modules on which this parameter does not exist will
    cause Verific to produce a VERI-1928 or VHDL-1676 message. This option
    can be specified multiple times to override multiple parameters.
    String values must be passed in double quotes (").

  -v, -vv
    Verbose log messages. (-vv is even more verbose than -v.)

The following additional import options are useful for debugging the Verific
bindings (for Yosys and/or Verific developers):

```

```
  -k
    Keep going after an unsupported verific primitive is found. The
    unsupported primitive is added as blockbox module to the design.
    This will also add all SVA related cells to the design parallel to
    the checker logic inferred by it.

  -V
    Import Verific netlist as-is without translating to Yosys cell types.

  -nosva
    Ignore SVA properties, do not infer checker logic.

  -L <int>
    Maximum number of ctrl bits for SVA checker FSMs (default=16).

  -n
    Keep all Verific names on instances and nets. By default only
    user-declared names are preserved.

  -d <dump_file>
    Dump the Verific netlist as a verilog file.


    verific [-work <libname>] -pp [options] <filename> [<module>]..

Pretty print design (or just module) to the specified file from the
specified library. (default library when -work is not present: "work")

Pretty print options:

  -verilog
    Save output for Verilog/SystemVerilog design modules (default).

  -vhdl
    Save output for VHDL design units.


    verific -app <application>..

Execute YosysHQ formal application on loaded Verilog files.

Application options:

    -module <module>
        Run formal application only on specified module.

    -blacklist <filename[:lineno]>
        Do not run application on modules from files that match the filename
        or filename and line number if provided in such format.
        Parameter can also contain comma separated list of file locations.

    -blfile <file>
        Do not run application on locations specified in file, they can represent filename
        or filename and location in file.
```

```

Applications:

  WARNING: Applications only available in commercial build.


    verific -template <name> <top_module>..

Generate template for specified top module of loaded design.

Template options:

  -out
    Specifies output file for generated template, by default output is stdout

  -chparam name value
    Generate template using this parameter value. Otherwise default parameter
    values will be used for templat generate functionality. This option
    can be specified multiple times to override multiple parameters.
    String values must be passed in double quotes (").

Templates:

  WARNING: Templates only available in commercial build.



    verific -cfg [<name> [<value>]]

Get/set Verific runtime flags.


Use YosysHQ Tabby CAD Suite if you need Yosys+Verific.
https://www.yosyshq.com/

Contact office@yosyshq.com for free evaluation
binaries of YosysHQ Tabby CAD Suite.
```

C.213 verilog_defaults – set default options for read_verilog

```
    verilog_defaults -add [options]

Add the specified options to the list of default options to read_verilog.


    verilog_defaults -clear

Clear the list of Verilog default options.


    verilog_defaults -push
    verilog_defaults -pop
```

```

Push or pop the list of default options to a stack. Note that -push does
not imply -clear.
```

C.214 verilog_defines – define and undefine verilog defines

```
    verilog_defines [options]

Define and undefine verilog preprocessor macros.

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

    -Uname[=definition]
        undefine the preprocessor symbol 'name'

    -reset
        clear list of defined preprocessor symbols

    -list
        list currently defined preprocessor symbols
```

C.215 wbflip – flip the whitebox attribute

```
    wbflip [selection]

Flip the whitebox attribute on selected cells. I.e. if it's set, unset it, and
vice-versa. Blackbox cells are not effected by this command.
```

C.216 wreduce – reduce the word size of operations if possible

```
    wreduce [options] [selection]

This command reduces the word size of operations. For example it will replace
the 32 bit adders in the following code with adders of more appropriate widths:

    module test(input [3:0] a, b, c, output [7:0] y);
        assign y = a + b + c + 1;
    endmodule

Options:

    -memx
        Do not change the width of memory address ports. Use this options in
        flows that use the 'memory_memx' pass.

```

```
    -keepdc
        Do not optimize explicit don't-care values.
```

C.217 write_aiger – write design to AIGER file

```
    write_aiger [options] [filename]

Write the current design to an AIGER file. The design must be flattened and
must not contain any cell types except $_AND_, $_NOT_, simple FF types,
$assert and $assume cells, and $initstate cells.

$assert and $assume cells are converted to AIGER bad state properties and
invariant constraints.

    -ascii
        write ASCII version of AIGER format

    -zinit
        convert FFs to zero-initialized FFs, adding additional inputs for
        uninitialized FFs.

    -miter
        design outputs are AIGER bad state properties

    -symbols
        include a symbol table in the generated AIGER file

    -map <filename>
        write an extra file with port and latch symbols

    -vmap <filename>
        like -map, but more verbose

    -I, -O, -B, -L
        If the design contains no input/output/assert/flip-flop then create one
        dummy input/output/bad_state-pin or latch to make the tools reading the
        AIGER file happy.
```

C.218 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

    -iname, -iattr
        enable -cname and -attr functionality for .names statements
        (the .cname and .attr statements will be included in the BLIF
        output after the truth table for the .names statement)

    -blackbox
        write blackbox cells with .blackbox statement.

    -impltf
        do not write definitions for the $true, $false and $undef wires.
```

C.219 write_btor – write design to BTOR file

```
    write_btor [options] [filename]

Write a BTOR description of the current design.

  -v
    Add comments and indentation to BTOR output file

  -s
    Output only a single bad property for all asserts

  -c
    Output cover properties using 'bad' statements instead of asserts

  -i <filename>
    Create additional info file with auxiliary information

  -x
    Output symbols for internal netnames (starting with '$')
```

C.220 write_cxxrtl – convert design to C++ RTL simulation

```
    write_cxxrtl [options] [filename]

Write C++ code that simulates the design. The generated code requires a driver
that instantiates the design, toggles its clock, and interacts with its ports.

The following driver may be used as an example for a design with a single clock
driving rising edge triggered flip-flops:

    #include "top.cc"

    int main() {
      cxxrtl_design::p_top top;
      top.step();
      while (1) {
        /* user logic */
        top.p_clk.set(false);
        top.step();
        top.p_clk.set(true);
        top.step();
      }
    }

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();
      }
    };

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

    }

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:

    cxxrtl_blackbox
        only valid on modules. if specified, the module contents are ignored,
        and the generated code includes only the module interface and a factory
        function, which will be called to instantiate the module.

    cxxrtl_edge
        only valid on inputs of black boxes. must be one of "p", "n", "a".
        if specified on signal 'clk', the generated code includes edge detectors
        'posedge_p_clk()' (if "p"), 'negedge_p_clk()' (if "n"), or both (if
        "a"), simplifying implementation of clocked black boxes.

    cxxrtl_template
```

```
        only valid on black boxes. must contain a space separated sequence of
        identifiers that have a corresponding black box parameters. for each
        of them, the generated code includes a ‘size_t‘ template parameter.

    cxxrtl_width
        only valid on ports of black boxes. must be a constant expression, which
        is directly inserted into generated code.

    cxxrtl_comb, cxxrtl_sync
        only valid on outputs of black boxes. if specified, indicates that every
        bit of the output port is driven, correspondingly, by combinatorial or
        synchronous logic. this knowledge is used for scheduling optimizations.
        if neither is specified, the output will be pessimistically treated as
        driven by both combinatorial and synchronous logic.

The following options are supported by this backend:

    -print-wire-types, -print-debug-wire-types
        enable additional debug logging, for pass developers.

    -header
        generate separate interface (.h) and implementation (.cc) files.
        if specified, the backend must be called with a filename, and filename
        of the interface is derived from filename of the implementation.
        otherwise, interface and implementation are generated together.

    -namespace <ns-name>
        place the generated code into namespace <ns-name>. if not specified,
        "cxxrtl_design" is used.

    -nohierarchy
        use design hierarchy as-is. in most designs, a top module should be
        present as it is exposed through the C API and has unbuffered outputs
        for improved performance; it will be determined automatically if absent.

    -noflatten
        don’t flatten the design. fully flattened designs can evaluate within
        one delta cycle if they have no combinatorial feedback.
        note that the debug interface and waveform dumps use full hierarchical
        names for all wires even in flattened designs.

    -noproc
        don’t convert processes to netlists. in most designs, converting
        processes significantly improves evaluation performance at the cost of
        slight increase in compilation time.

    -O <level>
        set the optimization level. the default is -O6. higher optimization
        levels dramatically decrease compile and run time, and highest level
        possible for a design should be used.

    -O0
        no optimization.

```

```
    -O1
        unbuffer internal wires if possible.

    -O2
        like -O1, and localize internal wires if possible.

    -O3
        like -O2, and inline internal wires if possible.

    -O4
        like -O3, and unbuffer public wires not marked (*keep*) if possible.

    -O5
        like -O4, and localize public wires not marked (*keep*) if possible.

    -O6
        like -O5, and inline public wires not marked (*keep*) if possible.

    -g <level>
        set the debug level. the default is -g4. higher debug levels provide
        more visibility and generate more code, but do not pessimize evaluation.

    -g0
        no debug information. the C API is disabled.

    -g1
        include bare minimum of debug information necessary to access all design
        state. the C API is enabled.

    -g2
        like -g1, but include debug information for all public wires that are
        directly accessible through the C++ interface.

    -g3
        like -g2, and include debug information for public wires that are tied
        to a constant or another public wire.

    -g4
        like -g3, and compute debug information on demand for all public wires
        that were optimized out.
```

C.221 write_edif – write design to EDIF netlist file

```
    write_edif [options] [filename]

Write the current design to an EDIF netlist file.

    -top top_module
        set the specified module as design top module

    -nogndvcc
        do not create "GND" and "VCC" cells. (this will produce an error
```

```
        if the design contains constant nets. use "hilomap" to map to custom
        constant drivers first)

    -gndvccy
        create "GND" and "VCC" cells with "Y" outputs. (the default is "G"
        for "GND" and "P" for "VCC".)

    -attrprop
        create EDIF properties for cell attributes

    -keep
        create extra KEEP nets by allowing a cell to drive multiple nets.

    -pvector {par|bra|ang}
        sets the delimiting character for module port rename clauses to
        parentheses, square brackets, or angle brackets.

Unfortunately there are different "flavors" of the EDIF file format. This
command generates EDIF files for the Xilinx place&route tools. It might be
necessary to make small modifications to this command when a different tool
is targeted.
```

C.222 write_file – write a text to a file

```
    write_file [options] output_file [input_file]

Write the text from the input file to the output file.

    -a
        Append to output file (instead of overwriting)


Inside a script the input file can also can a here-document:

    write_file hello.txt <<EOT
    Hello World!
    EOT
```

C.223 write_firrtl – write design to a FIRRTL file

```
    write_firrtl [options] [filename]

Write a FIRRTL netlist of the current design.
The following commands are executed by this command:
        pmuxtree
        bmuxmap
        demuxmap
```

C.224 write_ilang – (deprecated) alias of write_rtlil

```
See ‘help write_rtlil‘.
```

C.225 write_intersynth – write design to InterSynth netlist file

```
    write_intersynth [options] [filename]

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

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

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

    -selected
        only write selected modules. modules must be selected entirely or
        not at all.

http://bygone.clairexen.net/intersynth/
```

C.226 write_json – write design to a JSON file

```
    write_json [options] [filename]

Write a JSON netlist of the current design.

    -aig
        include AIG models for the different gate types

    -compat-int
        emit 32-bit or smaller fully-defined parameter values directly
        as JSON numbers (for compatibility with old parsers)


The general syntax of the JSON output created by this command is as follows:

    {
      "creator": "Yosys <version info>",
      "modules": {
        <module_name>: {
          "attributes": {
            <attribute_name>: <attribute_value>,
            ...
```

```
          },
          "parameter_default_values": {
            <parameter_name>: <parameter_value>,
            ...
          },
          "ports": {
            <port_name>: <port_details>,
            ...
          },
          "cells": {
            <cell_name>: <cell_details>,
            ...
          },
          "memories": {
            <memory_name>: <memory_details>,
            ...
          },
          "netnames": {
            <net_name>: <net_details>,
            ...
          }
        }
      },
      "models": {
        ...
      },
    }

Where <port_details> is:

    {
      "direction": <"input" | "output" | "inout">,
      "bits": <bit_vector>
      "offset": <the lowest bit index in use, if non-0>
      "upto": <1 if the port bit indexing is MSB-first>
      "signed": <1 if the port is signed>
    }

The "offset" and "upto" fields are skipped if their value would be 0.They don't affect connection semantics

    {
      "hide_name": <1 | 0>,
      "type": <cell_type>,
      "model": <AIG model name, if -aig option used>,
      "parameters": {
        <parameter_name>: <parameter_value>,
        ...
      },
      "attributes": {
        <attribute_name>: <attribute_value>,
        ...
      },
      "port_directions": {
        <port_name>: <"input" | "output" | "inout">,
```

```
          ...
        },
        "connections": {
          <port_name>: <bit_vector>,
          ...
        },
      }

And <memory_details> is:

      {
        "hide_name": <1 | 0>,
        "attributes": {
          <attribute_name>: <attribute_value>,
          ...
        },
        "width": <memory width>
        "start_offset": <the lowest valid memory address>
        "size": <memory size>
      }

And <net_details> is:

      {
        "hide_name": <1 | 0>,
        "bits": <bit_vector>
        "offset": <the lowest bit index in use, if non-0>
        "upto": <1 if the port bit indexing is MSB-first>
        "signed": <1 if the port is signed>
      }

The "hide_name" fields are set to 1 when the name of this cell or net is
automatically created and is likely not of interest for a regular user.

The "port_directions" section is only included for cells for which the
interface is known.

Module and cell ports and nets can be single bit wide or vectors of multiple
bits. Each individual signal bit is assigned a unique integer. The <bit_vector>
values referenced above are vectors of this integers. Signal bits that are
connected to a constant driver are denoted as string "0", "1", "x", or
"z" instead of a number.

Bit vectors (including integers) are written as string holding the binaryrepresentation of the value. Strin

For example the following Verilog code:

    module test(input x, y);
      (* keep *) foo #(.P(42), .Q(1337))
          foo_inst (.A({x, y}), .B({y, x}), .C({4'd10, {4{x}}}));
    endmodule

Translates to the following JSON output:

```

```
{
  "creator": "Yosys 0.9+2406 (git sha1 fb1168d8, clang 9.0.1 -fPIC -Os)",
  "modules": {
    "test": {
      "attributes": {
        "cells_not_processed": "00000000000000000000000000000001",
        "src": "test.v:1.1-4.10"
      },
      "ports": {
        "x": {
          "direction": "input",
          "bits": [ 2 ]
        },
        "y": {
          "direction": "input",
          "bits": [ 3 ]
        }
      },
      "cells": {
        "foo_inst": {
          "hide_name": 0,
          "type": "foo",
          "parameters": {
            "P": "00000000000000000000000000101010",
            "Q": "00000000000000000000010100111001"
          },
          "attributes": {
            "keep": "00000000000000000000000000000001",
            "module_not_derived": "00000000000000000000000000000001",
            "src": "test.v:3.1-3.55"
          },
          "connections": {
            "A": [ 3, 2 ],
            "B": [ 2, 3 ],
            "C": [ 2, 2, 2, 2, "0", "1", "0", "1" ]
          }
        }
      },
      "netnames": {
        "x": {
          "hide_name": 0,
          "bits": [ 2 ],
          "attributes": {
            "src": "test.v:1.19-1.20"
          }
        },
        "y": {
          "hide_name": 0,
          "bits": [ 3 ],
          "attributes": {
            "src": "test.v:1.22-1.23"
          }
        }
      }
```

```
        }
      }
    }

The models are given as And-Inverter-Graphs (AIGs) in the following form:

    "models": {
      <model_name>: [
        /*   0 */ [ <node-spec> ],
        /*   1 */ [ <node-spec> ],
        /*   2 */ [ <node-spec> ],
        ...
      ],
      ...
    },

The following node-types may be used:

    [ "port", <portname>, <bitindex>, <out-list> ]
      - the value of the specified input port bit

    [ "nport", <portname>, <bitindex>, <out-list> ]
      - the inverted value of the specified input port bit

    [ "and", <node-index>, <node-index>, <out-list> ]
      - the ANDed value of the specified nodes

    [ "nand", <node-index>, <node-index>, <out-list> ]
      - the inverted ANDed value of the specified nodes

    [ "true", <out-list> ]
      - the constant value 1

    [ "false", <out-list> ]
      - the constant value 0

All nodes appear in topological order. I.e. only nodes with smaller indices
are referenced by "and" and "nand" nodes.

The optional <out-list> at the end of a node specification is a list of
output portname and bitindex pairs, specifying the outputs driven by this node.

For example, the following is the model for a 3-input 3-output $reduce_and cell
inferred by the following code:

    module test(input [2:0] in, output [2:0] out);
      assign in = &out;
    endmodule

    "$reduce_and:3U:3": [
      /*   0 */ [ "port", "A", 0 ],
      /*   1 */ [ "port", "A", 1 ],
      /*   2 */ [ "and", 0, 1 ],
      /*   3 */ [ "port", "A", 2 ],
```

```
    /*   4 */ [ "and", 2, 3, "Y", 0 ],
    /*   5 */ [ "false", "Y", 1, "Y", 2 ]
  ]

Future version of Yosys might add support for additional fields in the JSON
format. A program processing this format must ignore all unknown fields.
```

C.227 write_rtlil – write design to RTLIL file

```
    write_rtlil [filename]

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

    -selected
        only write selected parts of the design.
```

C.228 write_simplec – convert design to simple C code

```
    write_simplec [options] [filename]

Write simple C code for simulating the design. The C code written can be used to
simulate the design in a C environment, but the purpose of this command is to
generate code that works well with C-based formal verification.

    -verbose
        this will print the recursive walk used to export the modules.

    -i8, -i16, -i32, -i64
        set the maximum integer bit width to use in the generated code.

THIS COMMAND IS UNDER CONSTRUCTION
```

C.229 write_smt2 – write design to SMT-LIBv2 file

```
    write_smt2 [options] [filename]

Write a SMT-LIBv2 [1] description of the current design. For a module with name
'<mod>' this will declare the sort '<mod>_s' (state of the module) and will
define and declare functions operating on that state.

The following SMT2 functions are generated for a module with name '<mod>'.
Some declarations/definitions are printed with a special comment. A prover
using the SMT2 files can use those comments to collect all relevant metadata
about the design.

    ; yosys-smt2-module <mod>
```

```
    (declare-sort |<mod>_s| 0)
        The sort representing a state of module <mod>.

    (define-fun |<mod>_h| ((state |<mod>_s|)) Bool (...))
        This function must be asserted for each state to establish the
        design hierarchy.

    ; yosys-smt2-input <wirename> <width>
    ; yosys-smt2-output <wirename> <width>
    ; yosys-smt2-register <wirename> <width>
    ; yosys-smt2-wire <wirename> <width>
    (define-fun |<mod>_n <wirename>| (|<mod>_s|) (_ BitVec <width>))
    (define-fun |<mod>_n <wirename>| (|<mod>_s|) Bool)
        For each port, register, and wire with the 'keep' attribute set an
        accessor function is generated. Single-bit wires are returned as Bool,
        multi-bit wires as BitVec.

    ; yosys-smt2-cell <submod> <instancename>
    (declare-fun |<mod>_h <instancename>| (|<mod>_s|) |<submod>_s|)
        There is a function like that for each hierarchical instance. It
        returns the sort that represents the state of the sub-module that
        implements the instance.

    (declare-fun |<mod>_is| (|<mod>_s|) Bool)
        This function must be asserted 'true' for initial states, and 'false'
        otherwise.

    (define-fun |<mod>_i| ((state |<mod>_s|)) Bool (...))
        This function must be asserted 'true' for initial states. For
        non-initial states it must be left unconstrained.

    (define-fun |<mod>_t| ((state |<mod>_s|) (next_state |<mod>_s|)) Bool (...))
        This function evaluates to 'true' if the states 'state' and
        'next_state' form a valid state transition.

    (define-fun |<mod>_a| ((state |<mod>_s|)) Bool (...))
        This function evaluates to 'true' if all assertions hold in the state.

    (define-fun |<mod>_u| ((state |<mod>_s|)) Bool (...))
        This function evaluates to 'true' if all assumptions hold in the state.

    ; yosys-smt2-assert <id> <filename:linenum>
    (define-fun |<mod>_a <id>| ((state |<mod>_s|)) Bool (...))
        Each $assert cell is converted into one of this functions. The function
        evaluates to 'true' if the assert statement holds in the state.

    ; yosys-smt2-assume <id> <filename:linenum>
    (define-fun |<mod>_u <id>| ((state |<mod>_s|)) Bool (...))
        Each $assume cell is converted into one of this functions. The function
        evaluates to 'true' if the assume statement holds in the state.

    ; yosys-smt2-cover <id> <filename:linenum>
    (define-fun |<mod>_c <id>| ((state |<mod>_s|)) Bool (...))
        Each $cover cell is converted into one of this functions. The function
```

```
        evaluates to 'true' if the cover statement is activated in the state.

Options:

    -verbose
        this will print the recursive walk used to export the modules.

    -stbv
        Use a BitVec sort to represent a state instead of an uninterpreted
        sort. As a side-effect this will prevent use of arrays to model
        memories.

    -stdt
        Use SMT-LIB 2.6 style datatypes to represent a state instead of an
        uninterpreted sort.

    -nobv
        disable support for BitVec (FixedSizeBitVectors theory). without this
        option multi-bit wires are represented using the BitVec sort and
        support for coarse grain cells (incl. arithmetic) is enabled.

    -nomem
        disable support for memories (via ArraysEx theory). this option is
        implied by -nobv. only $mem cells without merged registers in
        read ports are supported. call "memory" with -nordff to make sure
        that no registers are merged into $mem read ports. '<mod>_m' functions
        will be generated for accessing the arrays that are used to represent
        memories.

    -wires
        create '<mod>_n' functions for all public wires. by default only ports,
        registers, and wires with the 'keep' attribute are exported.

    -tpl <template_file>
        use the given template file. the line containing only the token '%%'
        is replaced with the regular output of this command.

    -solver-option <option> <value>
        emit a `; yosys-smt2-solver-option` directive for yosys-smtbmc to write
        the given option as a `(set-option ...)` command in the SMT-LIBv2.

[1] For more information on SMT-LIBv2 visit http://smt-lib.org/ or read David
R. Cok's tutorial: https://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf

---------------------------------------------------------------------------

Example:

Consider the following module (test.v). We want to prove that the output can
never transition from a non-zero value to a zero value.

        module test(input clk, output reg [3:0] y);
          always @(posedge clk)
            y <= (y << 1) | ^y;
```

```
        endmodule

For this proof we create the following template (test.tpl).

        ; we need QF_UFBV for this proof
        (set-logic QF_UFBV)

        ; insert the auto-generated code here
        %%

        ; declare two state variables s1 and s2
        (declare-fun s1 () test_s)
        (declare-fun s2 () test_s)

        ; state s2 is the successor of state s1
        (assert (test_t s1 s2))

        ; we are looking for a model with y non-zero in s1
        (assert (distinct (|test_n y| s1) #b0000))

        ; we are looking for a model with y zero in s2
        (assert (= (|test_n y| s2) #b0000))

        ; is there such a model?
        (check-sat)

The following yosys script will create a 'test.smt2' file for our proof:

        read_verilog test.v
        hierarchy -check; proc; opt; check -assert
        write_smt2 -bv -tpl test.tpl test.smt2

Running 'cvc4 test.smt2' will print 'unsat' because y can never transition
from non-zero to zero in the test design.
```

C.230 write_smv – write design to SMV file

```
    write_smv [options] [filename]

Write an SMV description of the current design.

    -verbose
        this will print the recursive walk used to export the modules.

    -tpl <template_file>
        use the given template file. the line containing only the token '%%'
        is replaced with the regular output of this command.

THIS COMMAND IS UNDER CONSTRUCTION
```

C.231 write_spice – write design to SPICE netlist file

```
    write_spice [options] [filename]

Write the current design to an SPICE netlist file.

    -big_endian
        generate multi-bit ports in MSB first order
        (default is LSB first)

    -neg net_name
        set the net name for constant 0 (default: Vss)

    -pos net_name
        set the net name for constant 1 (default: Vdd)

    -buf DC|subckt_name
        set the name for jumper element (default: DC)
        (used to connect different nets)

    -nc_prefix
        prefix for not-connected nets (default: _NC)

    -inames
        include names of internal ($-prefixed) nets in outputs
        (default is to use net numbers instead)

    -top top_module
        set the specified module as design top module
```

C.232 write_table – write design as connectivity table

```
    write_table [options] [filename]

Write the current design as connectivity table. The output is a tab-separated
ASCII table with the following columns:

  module name
  cell name
  cell type
  cell port
  direction
  signal

module inputs and outputs are output using cell type and port '-' and with
'pi' (primary input) or 'po' (primary output) or 'pio' as direction.
```

C.233 write_verilog – write design to Verilog file

```
    write_verilog [options] [filename]

Write the current design to a Verilog file.

    -sv
        with this option, SystemVerilog constructs like always_comb are used

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

    -renameprefix <prefix>
        insert this prefix in front of auto-generated instance names

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

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

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

    -siminit
        add initial statements with hierarchical refs to initialize FFs when
        in -noexpr mode.

    -nodec
        32-bit constant values are by default dumped as decimal numbers,
        not bit pattern. This option deactivates this feature and instead
        will write out all constants in binary.

    -decimal
        dump 32-bit constants in decimal and without size and radix

    -nohex
        constant values that are compatible with hex output are usually
        dumped as hex values. This option deactivates this feature and
        instead will write out all constants in binary.

    -nostr
        Parameters and attributes that are specified as strings in the
        original input will be output as strings by this back-end. This
        deactivates this feature and instead will write string constants
        as binary numbers.

    -simple-lhs
        Connection assignments with simple left hand side without concatenations.

    -extmem
        instead of initializing memories using assignments to individual
        elements, use the '$readmemh' function to read initialization data
```

```
        from a file. This data is written to a file named by appending
        a sequential index to the Verilog filename and replacing the extension
        with '.mem', e.g. 'write_verilog -extmem foo.v' writes 'foo-1.mem',
        'foo-2.mem' and so on.

    -defparam
        use 'defparam' statements instead of the Verilog-2001 syntax for
        cell parameters.

    -blackboxes
        usually modules with the 'blackbox' attribute are ignored. with
        this option set only the modules with the 'blackbox' attribute
        are written to the output file.

    -selected
        only write selected modules. modules must be selected entirely or
        not at all.

    -v
        verbose output (print new names of all renamed wires and cells)

Note that RTLIL processes can't always be mapped directly to Verilog
always blocks. This frontend should only be used to export an RTLIL
netlist, i.e. after the "proc" pass has been used to convert all
processes to logic networks and registers. A warning is generated when
this command is called on a design with RTLIL processes.
```

C.234 write_xaiger – write design to XAIGER file

```
    write_xaiger [options] [filename]

Write the top module (according to the (* top *) attribute or if only one module
is currently selected) to an XAIGER file. Any non $_NOT_, $_AND_, (optionally
$_DFF_N_, $_DFF_P_), or non (* abc9_box *) cells will be converted into psuedo-
inputs and pseudo-outputs. Whitebox contents will be taken from the equivalent
module in the '$abc9_holes' design, if it exists.

    -ascii
        write ASCII version of AIGER format

    -map <filename>
        write an extra file with port and box symbols

    -dff
        write $_DFF_[NP]_ cells
```

C.235 xilinx_dffopt – Xilinx: optimize FF control signal usage

```
    xilinx_dffopt [options] [selection]

```

```
Converts hardware clock enable and set/reset signals on FFs to emulation
using LUTs, if doing so would improve area.  Operates on post-techmap Xilinx
cells (LUT*, FD*).

    -lut4
        Assume a LUT4-based device (instead of a LUT6-based device).
```

C.236 xilinx_dsp – Xilinx: pack resources into DSPs

```
    xilinx_dsp [options] [selection]

Pack input registers (A2, A1, B2, B1, C, D, AD; with optional enable/reset),
pipeline registers (M; with optional enable/reset), output registers (P; with
optional enable/reset), pre-adder and/or post-adder into Xilinx DSP resources.

Multiply-accumulate operations using the post-adder with feedback on the 'C'
input will be folded into the DSP. In this scenario only, the 'C' input can be
used to override the current accumulation result with a new value, which will
be added to the multiplier result to form the next accumulation result.

Use of the dedicated 'PCOUT' -> 'PCIN' cascade path is detected for 'P' -> 'C'
connections (optionally, where 'P' is right-shifted by 17-bits and used as an
input to the post-adder -- a pattern common for summing partial products to
implement wide multipliers). Limited support also exists for similar cascading
for A and B using '[AB]COUT' -> '[AB]CIN'. Currently, cascade chains are limited
to a maximum length of 20 cells, corresponding to the smallest Xilinx 7 Series
device.

This pass is a no-op if the scratchpad variable 'xilinx_dsp.multonly' is set
to 1.


Experimental feature: addition/subtractions less than 12 or 24 bits with the
'(* use_dsp="simd" *)' attribute attached to the output wire or attached to
the add/subtract operator will cause those operations to be implemented using
the 'SIMD' feature of DSPs.

Experimental feature: the presence of a `$ge' cell attached to the registered
P output implementing the operation "(P >= <power-of-2>)" will be transformed
into using the DSP48E1's pattern detector feature for overflow detection.

    -family {xcup|xcu|xc7|xc6v|xc5v|xc4v|xc6s|xc3sda}
        select the family to target
        default: xc7
```

C.237 xilinx_srl – Xilinx shift register extraction

```
    xilinx_srl [options] [selection]

This pass converts chains of built-in flops (bit-level: $_DFF_[NP]_, $_DFFE_*
```

```
and word-level: $dff, $dffe) as well as Xilinx flops (FDRE, FDRE_1) into a
$__XILINX_SHREG cell. Chains must be of the same cell type, clock, clock polarity,
enable, and enable polarity (where relevant).
Flops with resets cannot be mapped to Xilinx devices and will not be inferred.
    -minlen N
        min length of shift register (default = 3)

    -fixed
        infer fixed-length shift registers.

    -variable
        infer variable-length shift registers (i.e. fixed-length shifts where
        each element also fans-out to a $shiftx cell).
```

C.238 zinit – add inverters so all FF are zero-initialized

```
    zinit [options] [selection]

Add inverters as needed to make all FFs zero-initialized.

    -all
        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:

```
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:

```
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:

```
read_verilog softusb_navre.v
hierarchy -check -top softusb_navre
proc; opt; memory; opt; fsm; opt; techmap; opt
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`

```
reg [7:0] a = 13, b;
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`

```c
#include <stdint.h>
#include <stdbool.h>

#define BITMAP_SIZE 64
#define OUTPORT 0x10000000

static uint32_t bitmap[BITMAP_SIZE/32];

static void bitmap_set(uint32_t idx) { bitmap[idx/32] |= 1 << (idx % 32); }
static bool bitmap_get(uint32_t idx) { return (bitmap[idx/32] & (1 << (idx % 32))) != 0; }
static void output(uint32_t val) { *((volatile uint32_t*)OUTPORT) = val; }

int main() {
	uint32_t i, j, k;
	output(2);
	for (i = 0; i < BITMAP_SIZE; i++) {
		if (bitmap_get(i)) continue;
		output(3+2*i);
		for (j = 2*(3+2*i);; j += 3+2*i) {
			if (j%2 == 0) continue;
			k = (j-3)/2;
			if (k >= BITMAP_SIZE) break;
			bitmap_set(k);
		}
	}
	output(0);
	return 0;
}
```

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).

```
$ cat example.ys
read_verilog example.v
show -pause
proc
show -pause
opt
show -pause

$ cat example.v
module example(input clk, a, b, c,
               output reg [1:0] y);
    always @(posedge clk)
        if (c)
            y <= c ? a + b : 2'd0;
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.

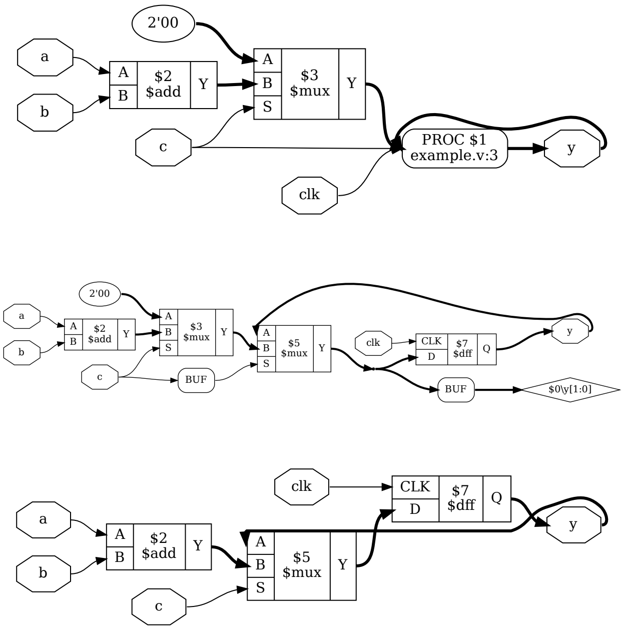


Figure 2. Output of the three show commands from Fig. 1

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.

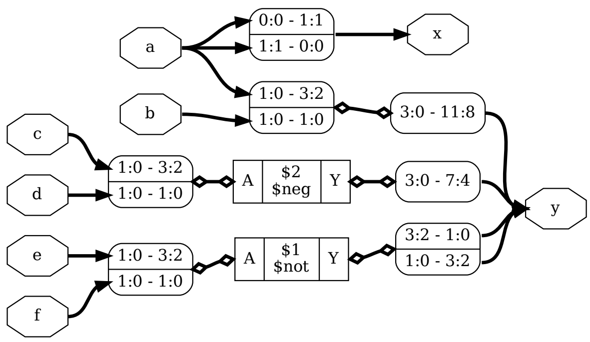


Figure 3. Output of yosys -p 'proc; opt; show' splice.v

```
module splice_demo(a, b, c, d, e, f, x, y);

input [1:0] a, b, c, d, e, f;
output [1:0] x = {a[0], a[1]};

output [11:0] y;
assign {y[11:4], y[1:0], y[3:2]} =
                {a, b, -{c, d}, ~{e, f}};

endmodule
```

Figure 4. splice.v

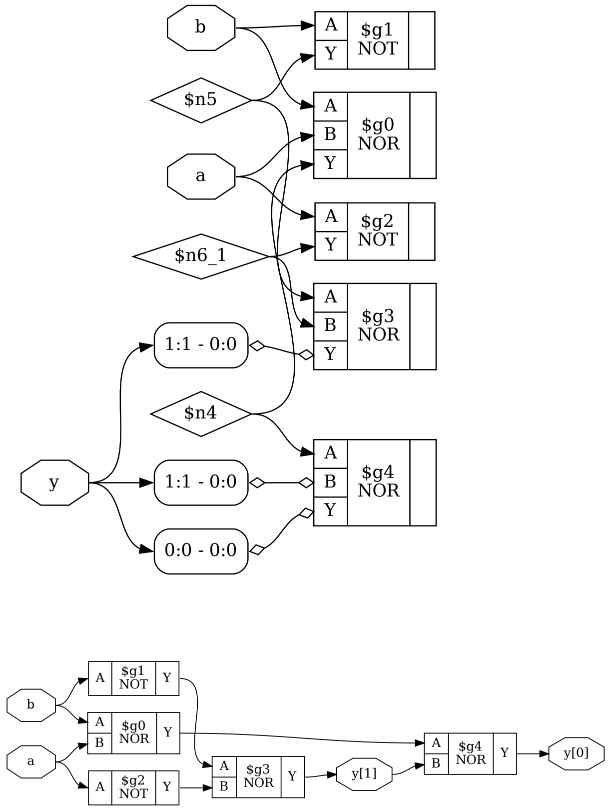


Figure 5. Effects of splitnets command and of providing a cell library. (The circuit is a half-adder built from simple CMOS gates.)

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.

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 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.

```
yosys> ls

1 modules:
  example

yosys> cd example

yosys [example]> ls

7 wires:
  $0\y[1:0]
  $add$example.v:5$2_Y
  a
  b
  c
  clk
  y

3 cells:
  $add$example.v:5$2
  $procdff$7
  $procmux$5
```

Figure 6. Demonstration of ls and cd using example.v from Fig. 1

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

```
  attribute \src "example.v:5"
  cell $add $add$example.v:5$2
    parameter \A_SIGNED 0
    parameter \A_WIDTH 1
    parameter \B_SIGNED 0
    parameter \B_WIDTH 1
    parameter \Y_WIDTH 2
    connect \A \a
    connect \B \b
    connect \Y $add$example.v:5$2_Y
  end
```

Figure 7. Output of `dump $2` using the design from Fig. 1 and Fig. 2

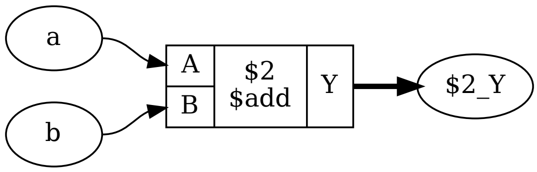


Figure 8. Output of show after `select $2` or `select t:$add` (see also Fig. 2)

```
module sumprod(a, b, c, sum, prod);

  input [7:0] a, b, c;
  output [7:0] sum, prod;

  {* sumstuff *}
  assign sum = a + b + c;
  {* *}

  assign prod = a * b * c;

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.

```
module foobaraddsub(a, b, c, d, fa, fs, ba, bs);
  input [7:0] a, b, c, d;
  output [7:0] fa, fs, ba, bs;
  assign fa = a + (* foo *) b;
  assign fs = a - (* foo *) b;
  assign ba = c + (* bar *) d;
  assign bs = c - (* bar *) d;
endmodule
```

Figure 9. Test module for operations on selections

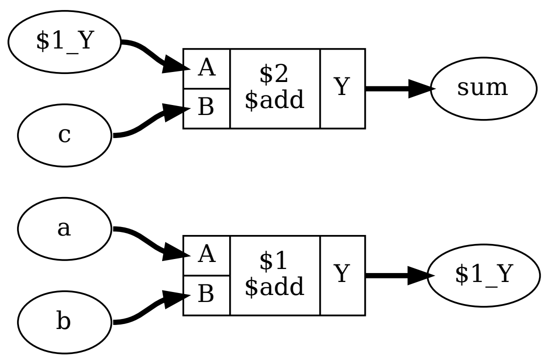


Figure 11. Output of `show a:sumstuff` on Fig. 10

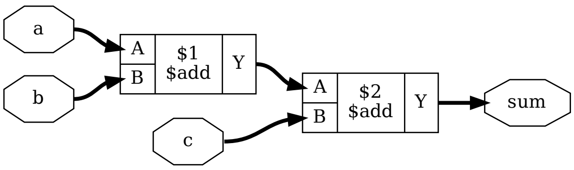


Figure 12. Output of `show a:sumstuff %x` on Fig. 10

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:

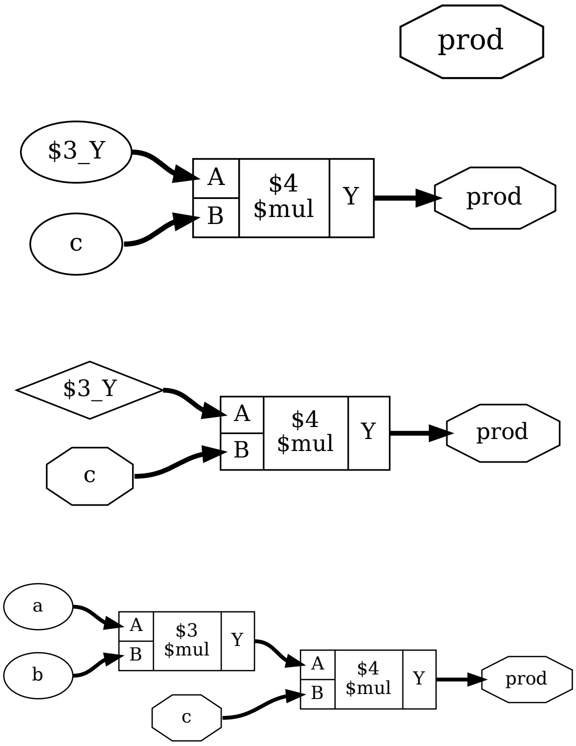


Figure 13. Objects selected by `select prod %ci...`

```
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]
```

```
module memdemo(clk, d, y);

input clk;
input [3:0] d;
output reg [3:0] y;

integer i;
reg [1:0] s1, s2;
reg [3:0] mem [0:3];

always @(posedge clk) begin
    for (i = 0; i < 4; i = i+1)
        mem[i] <= mem[(i+1) % 4] + mem[(i+2) % 4];
    { s2, s1 } = d ? { s1, s2 } ^ d : 4'b0;
    mem[s1] <= d;
    y <= mem[s2];
end

endmodule
```

Figure 14. Demo circuit for demonstrating some advanced Yosys features

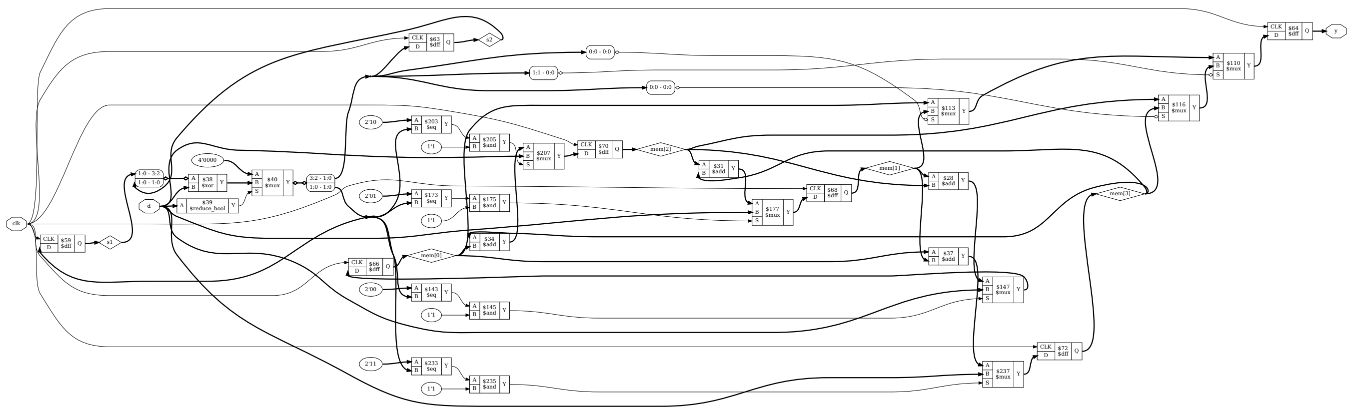


Figure 15. Complete circuit diagram for the design shown in Fig. 14

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.

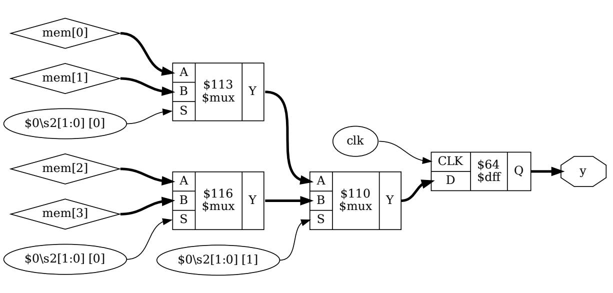


Figure 16. Output of `show y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff`

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.

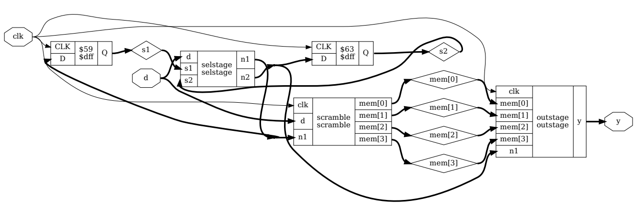
A. Evaluation of combinatorial circuits

The `eval` command can be used to evaluate combinatorial circuits. For example (see Fig. 17 for the circuit diagram of `selstage`):

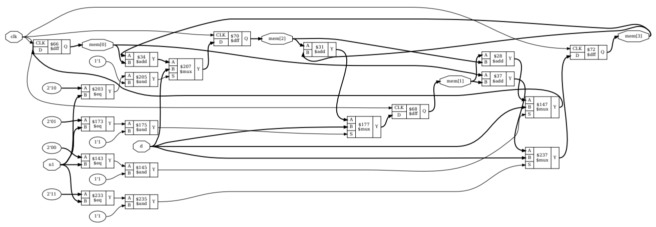
```
yosys [selstage]> eval -set s2,s1 4'b1001 -set d 4'hc -show n2 -show n1

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.
```

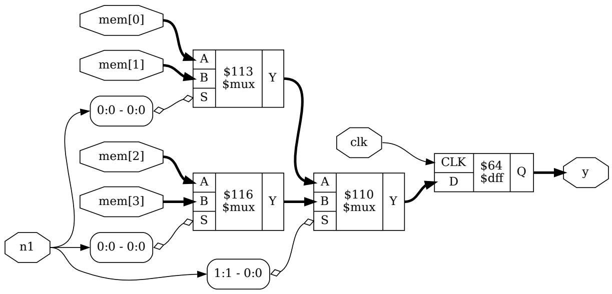
So the `-set` option is used to set input values and the `-show` option is used to specify the nets to evaluate. If no `-show` option is specified, all selected output ports are used per default.



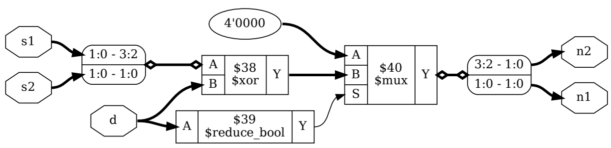
memdemo



scramble



outstage



selstage

```
select -set outstage y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff
select -set selstage y %ci2:+$dff[Q,D] %ci*:-$dff @outstage %d
select -set scramble mem* %ci2 %ci*:-$dff mem* %d @selstage %d
submod -name scramble @scramble
submod -name outstage @outstage
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

```
module primetest(p, a, b, ok);
input [15:0] p, a, b;
output ok = p != a*b || a == 1 || b == 1;
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.

Yosys 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.

```
yosys [memdemo]> 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

6. Executing SAT pass (solving SAT problems in the circuit).
Full command line: 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

Setting up time step 1:
Final constraint equation: { } = { }
Imported 29 cells to SAT database.

Setting up time step 2:
Final constraint equation: { } = { }
Imported 29 cells to SAT database.

Setting up time step 3:
Final constraint equation: { } = { }
Imported 29 cells to SAT database.

Setting up time step 4:
Import set-constraint for timestep: \y = 4'0001
Final constraint equation: \y = 4'0001
Imported 29 cells to SAT database.

Setting up time step 5:
Import set-constraint for timestep: \y = 4'0010
Final constraint equation: \y = 4'0010
Imported 29 cells to SAT database.

Setting up time step 6:
Import set-constraint for timestep: \y = 4'0011
Final constraint equation: \y = 4'0011
Imported 29 cells to SAT database.

Setting up initial state:
Final constraint equation: { \y \s2 \s1 \mem[3] \mem[2] \mem[1]
                             \mem[0] } = 24'xxxxxxxxxxxxxxxxxxxxxxxx

Import show expression: \y
Import show expression: \d

Solving problem with 10322 variables and 27881 clauses..
SAT model found. maximizing number of undefs.
SAT solving finished - model found:

  Time Signal Name                 Dec        Hex             Bin
  ---- -------------------- ---------- ---------- ---------------
  init \mem[0]                      --         --            xxxx
  init \mem[1]                      --         --            xxxx
  init \mem[2]                      --         --            xxxx
  init \mem[3]                      --         --            xxxx
  init \s1                          --         --              xx
  init \s2                          --         --              xx
  init \y                           --         --            xxxx
  ---- -------------------- ---------- ---------- ---------------
     1 \d                            0          0            0000
     1 \y                           --         --            xxxx
  ---- -------------------- ---------- ---------- ---------------
     2 \d                            1          1            0001
     2 \y                           --         --            xxxx
  ---- -------------------- ---------- ---------- ---------------
     3 \d                            2          2            0010
     3 \y                            0          0            0000
  ---- -------------------- ---------- ---------- ---------------
     4 \d                            3          3            0011
     4 \y                            1          1            0001
  ---- -------------------- ---------- ---------- ---------------
     5 \d                           --         --            001x
     5 \y                            2          2            0010
  ---- -------------------- ---------- ---------- ---------------
     6 \d                           --         --            xxxx
     6 \y                            3          3            0011
```

Figure 20. Solving a sequential SAT problem in the memdemo module from Fig. 14.

Let's 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`

[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.

script. Listing 5 shows the Yosys commands that are executed by `verilog2btor.sh`.

```
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:

```
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^2C 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/.