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 were 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

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

1 Introduction 12
   1.1 History of Yosys ................................................. 12
   1.2 Structure of this Document ........................................... 13

2 Basic Principles 14
   2.1 Levels of Abstraction ................................................. 14
      2.1.1 System Level .................................................. 15
      2.1.2 High Level ..................................................... 15
      2.1.3 Behavioural Level .............................................. 15
      2.1.4 Register-Transfer Level (RTL) ............................... 16
      2.1.5 Logical Gate Level ........................................... 16
      2.1.6 Physical Gate Level ........................................... 17
      2.1.7 Switch Level .................................................... 17
      2.1.8 Yosys .......................................................... 17
   2.2 Features of Synthesizable Verilog ................................... 17
      2.2.1 Structural Verilog .............................................. 18
      2.2.2 Expressions in Verilog ........................................ 18
      2.2.3 Behavioural Modelling ........................................ 18
      2.2.4 Functions and Tasks .......................................... 19
      2.2.5 Conditionals, Loops and Generate-Statements .................. 19
      2.2.6 Arrays and Memories .......................................... 20
   2.3 Challenges in Digital Circuit Synthesis ............................ 20
      2.3.1 Standards Compliance ........................................ 20
      2.3.2 Optimizations .................................................. 21
      2.3.3 Technology Mapping ........................................... 21
   2.4 Script-Based Synthesis Flows ....................................... 21
   2.5 Methods from Compiler Design ..................................... 22
      2.5.1 Lexing and Parsing ............................................ 22
      2.5.2 Multi-Pass Compilation ...................................... 24
3 Approach

3.1 Data- and Control-Flow .................................................. 25
3.2 Internal Formats in Yosys ................................................ 26
3.3 Typical Use Case ............................................................. 26

4 Implementation Overview

4.1 Simplified Data Flow ...................................................... 28
4.2 The RTL Intermediate Language ....................................... 29
  4.2.1 RTLIL Identifiers ..................................................... 30
  4.2.2 RTLIL::Design and RTLIL::Module .............................. 31
  4.2.3 RTLIL::Cell and RTLIL::Wire .................................... 31
  4.2.4 RTLIL::SigSpec ...................................................... 32
  4.2.5 RTLIL::Process .................................................... 32
  4.2.6 RTLIL::Memory ..................................................... 34
4.3 Command Interface and Synthesis Scripts ............................ 35
4.4 Source Tree and Build System ......................................... 35

5 Internal Cell Library

5.1 RTL Cells ................................................................. 37
  5.1.1 Unary Operators .................................................... 37
  5.1.2 Binary Operators .................................................. 38
  5.1.3 Multiplexers ........................................................ 38
  5.1.4 Registers ............................................................ 39
  5.1.5 Memories ............................................................ 39
  5.1.6 Finite State Machines ............................................. 42
5.2 Gates ................................................................. 42

6 Programming Yosys Extensions

6.1 The “CodingReadme” File ............................................. 44
6.2 The “stubsnets” Example Module ..................................... 49
7 The Verilog and AST Frontends

7.1 Transforming Verilog to AST
   7.1.1 The Verilog Preprocessor
   7.1.2 The Verilog Lexer
   7.1.3 The Verilog Parser

7.2 Transforming AST to RTLIL
   7.2.1 AST Simplification
   7.2.2 Generating RTLIL

7.3 Synthesizing Verilog always Blocks
   7.3.1 The ProcessGenerator Algorithm
   7.3.2 The proc pass

7.4 Synthesizing Verilog Arrays

7.5 Synthesizing Parametric Designs

8 Optimizations

8.1 Simple Optimizations
   8.1.1 The opt_expr pass
   8.1.2 The opt_muxtree pass
   8.1.3 The opt_reduce pass
   8.1.4 The opt_rmdff pass
   8.1.5 The opt_clean pass
   8.1.6 The opt_merge pass

8.2 FSM Extraction and Encoding
   8.2.1 FSM Detection
   8.2.2 FSM Extraction
   8.2.3 FSM Optimization
   8.2.4 FSM Recoding

8.3 Logic Optimization

9 Technology Mapping

9.1 Cell Substitution

9.2 Subcircuit Substitution

9.3 Gate-Level Technology Mapping

A Auxiliary Libraries

A.1 SHA1
A.2 BigInt
A.3 SubCircuit
A.4 ezSAT
CONTENTS

B Auxiliary Programs

B.1 yosys-config .................................................. 71
B.2 yosys-filterlib ............................................. 71
B.3 yosys-abc .................................................. 71

C Command Reference Manual

C.1 abc – use ABC for technology mapping ..................... 72
C.2 add – add objects to the design ............................ 75
C.3 aigmap – map logic to and-inverter-graph circuit ......... 75
C.4 aluacce – extract ALU and MACC cells .................... 76
C.5 assertpmux – convert internal signals to module ports .. 76
C.6 async2sync – convert async FF inputs to sync circuits ... 76
C.7 attrmap – renaming attributes ............................... 76
C.8 attmvep – move or copy attributes from wires to driving cells . 77
C.9 blackbox – change type of cells in the design ............. 78
C.10 cd – a shortcut for ‘select -module <name>’ ................ 78
C.11 check – check for obvious problems in the design ....... 78
C.12 chformal – change formal constraints of the design ...... 79
C.13 chparam – re-evaluate modules with new parameters ..... 79
C.14 chtype – change type of cells in the design .............. 79
C.15 clean – remove unused cells and wires ..................... 80
C.16 clk2fflogic – convert clocked FFs to generic $ff cells ... 80
C.17 connect – create or remove connections ................... 80
C.18 connwrappers – match width of input-output port pairs ... 81
C.19 coolrunner2_sop – break $sop cells into ANDTERM/ORTERM cells 81
C.20 copy – copy modules in the design ......................... 81
C.21 cover – print code coverage counters ....................... 82
C.22 delete – delete objects in the design ....................... 82
C.23 deminout – demote inout ports to input or output ...... 83
C.24 design – save, restore and reset current design ........... 83
C.25 dff2dffe – transform $dff cells to $dffe cells ............ 84
C.26 dff2dffs – process sync set/reset with SR over CE priority . 84
C.27 dffinit – set INIT param on FF cells ....................... 85
C.28 dfflibmap – technology mapping of flip-flops ............. 85
C.29 dffsr2dff – convert DFFSR cells to simpler FF cell types ... 85
C.30 dump – print parts of the design in ilang format ........ 85
C.31 echo – turning echoing back of commands on and off ...... 86
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.32</td>
<td>edgetypes – list all types of edges in selection</td>
<td>86</td>
</tr>
<tr>
<td>C.33</td>
<td>equiv_add – add a $equiv cell</td>
<td>86</td>
</tr>
<tr>
<td>C.34</td>
<td>equiv_induct – proving $equiv cells using temporal induction</td>
<td>87</td>
</tr>
<tr>
<td>C.35</td>
<td>equiv_make – prepare a circuit for equivalence checking</td>
<td>87</td>
</tr>
<tr>
<td>C.36</td>
<td>equiv_mark – mark equivalence checking regions</td>
<td>88</td>
</tr>
<tr>
<td>C.37</td>
<td>equiv_miter – extract miter from equiv circuit</td>
<td>88</td>
</tr>
<tr>
<td>C.38</td>
<td>equiv_purge – purge equivalence checking module</td>
<td>88</td>
</tr>
<tr>
<td>C.39</td>
<td>equiv_remove – remove $equiv cells</td>
<td>88</td>
</tr>
<tr>
<td>C.40</td>
<td>equiv_simple – try proving simple $equiv instances</td>
<td>89</td>
</tr>
<tr>
<td>C.41</td>
<td>equiv_status – print status of equivalent checking module</td>
<td>89</td>
</tr>
<tr>
<td>C.42</td>
<td>equiv_struct – structural equivalence checking</td>
<td>89</td>
</tr>
<tr>
<td>C.43</td>
<td>eval – evaluate the circuit given an input</td>
<td>90</td>
</tr>
<tr>
<td>C.44</td>
<td>expose – convert internal signals to module ports</td>
<td>90</td>
</tr>
<tr>
<td>C.45</td>
<td>extract – find subcircuits and replace them with cells</td>
<td>91</td>
</tr>
<tr>
<td>C.46</td>
<td>extract_counter – Extract GreenPak4 counter cells</td>
<td>92</td>
</tr>
<tr>
<td>C.47</td>
<td>extract_fa – find and extract full/half adders</td>
<td>93</td>
</tr>
<tr>
<td>C.48</td>
<td>extract_reduce – converts gate chains into $reduce_* cells</td>
<td>93</td>
</tr>
<tr>
<td>C.49</td>
<td>flatten – flatten design</td>
<td>94</td>
</tr>
<tr>
<td>C.50</td>
<td>freduce – perform functional reduction</td>
<td>94</td>
</tr>
<tr>
<td>C.51</td>
<td>fsm – extract and optimize finite state machines</td>
<td>94</td>
</tr>
<tr>
<td>C.52</td>
<td>fsm_detect – finding FSMs in design</td>
<td>95</td>
</tr>
<tr>
<td>C.53</td>
<td>fsm_expand – expand FSM cells by merging logic into it</td>
<td>95</td>
</tr>
<tr>
<td>C.54</td>
<td>fsm_export – exporting FSMs to KISS2 files</td>
<td>96</td>
</tr>
<tr>
<td>C.55</td>
<td>fsm_extract – extracting FSMs in design</td>
<td>96</td>
</tr>
<tr>
<td>C.56</td>
<td>fsm_info – print information on finite state machines</td>
<td>96</td>
</tr>
<tr>
<td>C.57</td>
<td>fsm_map – mapping FSMs to basic logic</td>
<td>97</td>
</tr>
<tr>
<td>C.58</td>
<td>fsm_opt – optimize finite state machines</td>
<td>97</td>
</tr>
<tr>
<td>C.59</td>
<td>fsm_recode – recoding finite state machines</td>
<td>97</td>
</tr>
<tr>
<td>C.60</td>
<td>greenpak4_dffinv – merge greenpak4 inverters and DFF/latches</td>
<td>97</td>
</tr>
<tr>
<td>C.61</td>
<td>help – display help messages</td>
<td>98</td>
</tr>
<tr>
<td>C.62</td>
<td>hierarchy – check, expand and clean up design hierarchy</td>
<td>98</td>
</tr>
<tr>
<td>C.63</td>
<td>hilomap – technology mapping of constant hi- and/or lo-drivers</td>
<td>99</td>
</tr>
<tr>
<td>C.64</td>
<td>history – show last interactive commands</td>
<td>99</td>
</tr>
<tr>
<td>C.65</td>
<td>ice40_ffinit – iCE40: handle FF init values</td>
<td>100</td>
</tr>
<tr>
<td>C.66</td>
<td>ice40_ffssr – iCE40: merge synchronous set/reset into FF cells</td>
<td>100</td>
</tr>
<tr>
<td>C.67</td>
<td>ice40_opt – iCE40: perform simple optimizations</td>
<td>100</td>
</tr>
</tbody>
</table>
CONTENTS

C.68 insbuf – insert buffer cells for connected wires ............................................. 100
C.69 iopadmap – technology mapping of i/o pads (or buffers) ............................. 101
C.70 json – write design in JSON format ............................................................. 101
C.71 log – print text and log files ........................................................................ 102
C.72 ls – list modules or objects in modules ......................................................... 102
C.73 ltp – print longest topological path ............................................................... 102
C.74 lut2mux – convert $lut to $_MUX_ ............................................................... 103
C.75 maccmap – mapping macc cells ................................................................. 103
C.76 memory – translate memories to basic cells .................................................. 103
C.77 memory_bram – map memories to block rams ............................................. 103
C.78 memory_collect – creating multi-port memory cells .................................... 105
C.79 memory_dff – merge input/output DFFs into memories ............................... 105
C.80 memory_map – translate multiport memories to basic cells ......................... 105
C.81 memory_memx – emulate vlog sim behavior for mem ports ......................... 105
C.82 memory_nordff – extract read port FFs from memories .............................. 106
C.83 memory_share – consolidate memory ports .................................................. 106
C.84 memory_unpack – unpack multi-port memory cells ..................................... 106
C.85 miter – automatically create a miter circuit ................................................... 107
C.86 muxcover – cover trees of MUX cells with wider MUXes ............................ 107
C.87 nlutmap – map to LUTs of different sizes ...................................................... 108
C.88 opt – perform simple optimizations ............................................................. 108
C.89 opt_clean – remove unused cells and wires ............................................... 109
C.90 opt_demorgan – Optimize reductions with DeMorgan equivalents .............. 109
C.91 opt_expr – perform const folding and simple expression rewriting ............... 109
C.92 opt_merge – consolidate identical cells ....................................................... 110
C.93 opt_muxtree – eliminate dead trees in multiplexer trees .............................. 110
C.94 opt_reduce – simplify large MUXes and AND/OR gates .............................. 110
C.95 opt_rmdff – remove DFFs with constant inputs ........................................... 111
C.96 plugin – load and list loaded plugins .......................................................... 111
C.97 pmuxtree – transform $pmux cells to trees of $mux cells ............................. 111
C.98 prep – generic synthesis script .................................................................. 111
C.99 proc – translate processes to netlists ........................................................... 113
C.100 proc_arst – detect asynchronous resets ..................................................... 113
C.101 proc_clean – remove empty parts of processes .......................................... 113
C.102 proc_dff – extract flip-flops from processes ............................................... 114
C.103 proc_dlatch – extract latches from processes ............................................. 114
C.104 proc_init – convert initial block to init attributes . . . . . . . . . . . . . . . . . . . . . . . . 114
C.105 proc_mux – convert decision trees to multiplexers . . . . . . . . . . . . . . . . . . . . . . . . 114
C.106 proc_rmded – eliminate dead trees in decision trees . . . . . . . . . . . . . . . . . . . . . . . . 114
C.107 qwp – quadratic wirelength placer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
C.108 read – load HDL designs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
C.109 read_blif – read BLIF file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
C.110 read_ilang – read modules from ilang file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
C.111 read_json – read JSON file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
C.112 read_liberty – read cells from liberty file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
C.113 read_verilog – read modules from Verilog file . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
C.114 rename – rename object in the design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
C.115 rmports – remove module ports with no connections . . . . . . . . . . . . . . . . . . . . . . . 120
C.116 sat – solve a SAT problem in the circuit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
C.117 scatter – add additional intermediate nets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
C.118 sc – detect strongly connected components (logic loops) . . . . . . . . . . . . . . . . . . . . 123
C.119 script – execute commands from script file . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
C.120 select – modify and view the list of selected objects . . . . . . . . . . . . . . . . . . . . . . . . 124
C.121 setattr – set/unset attributes on objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
C.122 setparam – set/unset parameters on objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
C.123 setundef – replace undef values with defined constants . . . . . . . . . . . . . . . . . . . . . 129
C.124 share – perform sat-based resource sharing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
C.125 shell – enter interactive command mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
C.126 show – generate schematics using graphviz . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
C.127 shregmap – map shift registers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
C.128 sim – simulate the circuit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
C.129 simplemap – mapping simple coarse-grain cells . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
C.130 splice – create explicit splicing cells . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
C.131 splitnets – split up multi-bit nets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
C.132 stat – print some statistics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
C.133 submod – moving part of a module to a new submodule . . . . . . . . . . . . . . . . . . . . . 136
C.134 synth – generic synthesis script . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
C.135 synth_acronix – synthesis for Achronix Speedster22i FPGAs . . . . . . . . . . . . . . . . . . 138
C.136 synth_coolrunner2 – synthesis for Xilinx Coolrunner-II CPLDs . . . . . . . . . . . . . . . . . 139
C.137 synth_easic – synthesis for eASIC platform . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
C.138 synth_ecp5 – synthesis for ECP5 FPGAs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142
C.139 synth_gowin – synthesis for Gowin FPGAs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
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 VHDL1 [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 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 implementation of the synthesizable HDL subset.

Due to the author’s preference for Verilog over VHDL it was decided early on to go for Verilog instead of VHDL2. 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.

1VHDL is an acronym for “VHSIC hardware description language” and VHSIC is an acronym for “Very-High-Speed Integrated Circuits”.

2A 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 10 covers the evaluation of the performance (correctness and quality) of Yosys on real-world input data. The chapter concludes the main part of this document with conclusions and outlook to future work.

Various appendices, including a command reference manual (App. C) and an evaluation of pre-existing FOSS Verilog synthesis tools (App. ??) 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.](image-url)
CHAPTER 2. BASIC PRINCIPLES

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 \( y \leftarrow a + b \) 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.
CHAPTER 2. BASIC PRINCIPLES

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:

```verilog
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
CHAPTER 2. BASIC PRINCIPLES

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:

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

// latched
always @* begin
  if (!hold)
    y <= a + b;
end
```
CHAPTER 2. BASIC PRINCIPLES

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:

```verilog
// 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:

```verilog
// 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:

```verilog
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:
CHAPTER 2. BASIC PRINCIPLES

- 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.
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 behavioral 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 behavioral 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.
CHAPTER 2. BASIC PRINCIPLES

<table>
<thead>
<tr>
<th>Token-Type</th>
<th>Token-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOK_ASSIGN</td>
<td></td>
</tr>
<tr>
<td>TOK_IDENTIFIER</td>
<td>“foo”</td>
</tr>
<tr>
<td>TOK_EQ</td>
<td></td>
</tr>
<tr>
<td>TOK_IDENTIFIER</td>
<td>“bar”</td>
</tr>
<tr>
<td>TOK_PLUS</td>
<td></td>
</tr>
<tr>
<td>TOK_NUMBER</td>
<td>42</td>
</tr>
<tr>
<td>TOK_SEMICOLON</td>
<td></td>
</tr>
</tbody>
</table>

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):

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

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

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

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

Usually the AST is then converted into yet another representation that is more suitable for further processing. In compilers this is often an assembler-like three-address-code intermediate representation. [ASU86]
CHAPTER 2. BASIC PRINCIPLES

![Example abstract syntax tree for the Verilog expression “assign foo = bar + 42;”](image)

**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 an 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](image_url)

25
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 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.
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 I2C 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 ILANG Frontend.

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

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

With the exception of the AST Frontend, 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).
CHAPTER 4. IMPLEMENTATION OVERVIEW

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\textsuperscript{1} representation. The only exception are the high-level frontends that use the AST representation as an intermediate step before generating RTLIL data.

In order to avoid reinventing names for the RTLIL classes, they are simply referred to by their full C++ name, i.e. including the \texttt{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 \texttt{RTLIL::Design} contains $N$ (zero to many) instances of \texttt{RTLIL::Module}. A two-pointed arrow indicates a $1 : 1$ relationship.

The \texttt{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 \texttt{RTLIL::Design} objects. For example when a pass is reading an auxiliary Verilog file such as a cell library, it might create an additional \texttt{RTLIL::Design} object and call the Verilog frontend with this other object to parse the cell library.

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

\textsuperscript{1}The \textit{Language} in \textit{RTL Intermediate Language} refers to the fact, that RTLIL also has a text representation, usually referred to as \textit{Intermediate Language} (ILANG).

29
CHAPTER 4. IMPLEMENTATION OVERVIEW

Figure 4.2: Simplified RTLIL Entity-Relationship Diagram

- RTLIL::Cell and RTLIL::Wire objects represent classical netlist data.
- 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 a central location. Internally auto-generated names that may hold important information for Yosys developers can be used without disturbing external tools. For example the Verilog backend assigns names in the form `_integer_`.

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

All RTLIL identifiers are case sensitive.

4.2.2 RTLIL::Design and RTLIL::Module

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

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

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

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

Verilog and VHDL both support parametric modules (known as “generic entities” in VHDL). The RTLIL format does not support parametric modules itself. Instead each module contains a callback function into the AST frontend to generate a parametrized variation of the RTLIL::Module as needed. This callback then returns the auto-generated name of the parametrized variation of the module. (A hash over the parameters and the module name is used to prohibit the same parametrized variation 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)
CHAPTER 4. IMPLEMENTATION OVERVIEW

- 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.

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:

```verbatim
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
```
CHAPTER 4. IMPLEMENTATION OVERVIEW

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

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

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

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

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

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

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

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

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

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

\(^2\)The syntax 1’1 in the ILANG code specifies a constant with a length of one bit (the first “1”), and this bit is a one (the second “1”).
CHAPTER 4. IMPLEMENTATION OVERVIEW

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:

```vhdl
process $proc$ff_with_en_and_async_reset.v:4$1
  assign $0\ll[0:0]\r
  switch $\text{\texttt{enable}}$
  case $\text{1'1}$
    assign $0\ll[0:0]\r
  case
  end
  sync posedge $\text{\texttt{clock}}$
  update $\ll[0:0]$\r
  sync high $\text{\texttt{reset}}$
  update $\text{1'0}$
end
```

Different combinations of passes may yield different results. Note that $\text{adff}$ and $\text{mux}$ 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 $\text{proc}$ 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 $\text{memrd}$ cells and all write accesses to $\text{memwr}$ cells by the language frontend. These cells consist of independent read- and write-ports to the memory. The \MEMID parameter on these cells is used to link them together and to the RTLIL::Memory object they belong to.

The rationale behind using separate cells for the individual ports versus creating a large multiport memory cell right in the language frontend is that the separate $\text{memrd}$ and $\text{memwr}$ 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 $\text{memrd}$ and $\text{memwr}$ 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 $\text{mem}$ 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:

- \( A_{\text{SIGNED}} \)
  
  Set to a non-zero value if the input \( A \) is signed and therefore should be sign-extended when needed.

- \( A_{\text{WIDTH}} \)
  
  The width of the input port \( A \).

- \( Y_{\text{WIDTH}} \)
  
  The width of the output port \( Y \).

Table 5.1 lists all cells for unary RTL operators.

Note that $\text{reduce_or}$ and $\text{reduce_bool}$ actually represent the same logic function. But the HDL frontends generate them in different situations. A $\text{reduce_or}$ cell is generated when the prefix \( | \) operator is being used. A $\text{reduce_bool}$ cell is generated when a bit vector is used as a condition in an if-statement or ?:-expression.
CHAPTER 5. INTERNAL CELL LIBRARY

<table>
<thead>
<tr>
<th>Verilog</th>
<th>Cell Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y = ~A</td>
<td>$not</td>
</tr>
<tr>
<td>Y = +A</td>
<td>$pos</td>
</tr>
<tr>
<td>Y = ~A</td>
<td>$neg</td>
</tr>
<tr>
<td>Y = &amp;A</td>
<td>$reduce_and</td>
</tr>
<tr>
<td>Y =</td>
<td>A</td>
</tr>
<tr>
<td>Y = ^A</td>
<td>$reduce_xor</td>
</tr>
<tr>
<td>Y = ~^A</td>
<td>$reduce_xnor</td>
</tr>
<tr>
<td>Y =</td>
<td>A</td>
</tr>
<tr>
<td>Y = !A</td>
<td>$logic_not</td>
</tr>
</tbody>
</table>

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

5.1.2 Binary Operators

All binary RTL cells have two input ports \( \text{\texttt{A}} \) and \( \text{\texttt{B}} \) and one output port \( \text{\texttt{Y}} \). They also have the following parameters:

- \( \text{\texttt{A\_SIGNED}} \)
  Set to a non-zero value if the input \( \text{\texttt{A}} \) is signed and therefore should be sign-extended when needed.

- \( \text{\texttt{A\_WIDTH}} \)
  The width of the input port \( \text{\texttt{A}} \).

- \( \text{\texttt{B\_SIGNED}} \)
  Set to a non-zero value if the input \( \text{\texttt{B}} \) is signed and therefore should be sign-extended when needed.

- \( \text{\texttt{B\_WIDTH}} \)
  The width of the input port \( \text{\texttt{B}} \).

- \( \text{\texttt{Y\_WIDTH}} \)
  The width of the output port \( \text{\texttt{Y}} \).

Table 5.2 lists all cells for binary RTL operators.

5.1.3 Multiplexers

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

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

The \texttt{$\text{\texttt{pmux}}$} cell is used to multiplex between many inputs using a one-hot select signal. Cells of this type have a \( \text{\texttt{WIDTH}} \) and a \( \text{\texttt{S\_WIDTH}} \) parameter and inputs \( \text{\texttt{A}} \), \( \text{\texttt{B}} \), and \( \text{\texttt{S}} \) and an output \( \text{\texttt{Y}} \). The \( \text{\texttt{S}} \) input is \( \text{\texttt{S\_WIDTH}} \) bits wide. The \( \text{\texttt{A}} \) input and the output are both \( \text{\texttt{WIDTH}} \) bits wide and the \( \text{\texttt{B}} \) input is \( \text{\texttt{WIDTH}} \times \text{\texttt{S\_WIDTH}} \) bits wide. When all bits of \( \text{\texttt{S}} \) are zero, the value from \( \text{\texttt{A}} \) input is sent to the output, if the \( n \)’th bit from \( \text{\texttt{S}} \) is set, the value \( n \)’th \( \text{\texttt{WIDTH}} \) bits wide slice of the \( \text{\texttt{B}} \) input is sent to the output. When more than one bit from \( \text{\texttt{S}} \) is set the output is undefined. Cells of this type are used to model “parallel cases” (defined by using the \texttt{parallel\_case} attribute or detected by an optimization).

Behavioural code with cascaded \texttt{if-then-else} and \texttt{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 \texttt{$\text{\texttt{and}}$} gate).
5.1.4 Registers

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

- $\text{WIDTH}$
  The width of input $\text{D}$ and output $\text{Q}$.

- $\text{CLK\_POLARITY}$
  Clock is active on the positive edge if this parameter has the value $1'b1$ and on the negative edge if this parameter is $1'b0$.

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

- $\text{ARST\_POLARITY}$
  The asynchronous reset is high-active if this parameter has the value $1'b1$ and low-active if this parameter is $1'b0$.

- $\text{ARST\_VALUE}$
  The state of $\text{Q}$ will be set to this value when the reset is active.

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

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

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

5.1.5 Memories

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

In the first alternative the RTLIL::Memory objects hold the general metadata for the memory (bit width, size in number of words, etc.) and for each port a $memrd$ (read port) or $memwr$ (write port) cell is created.
CHAPTER 5. INTERNAL CELL LIBRARY

Having individual cells for read and write ports has the advantage that they can be consolidated using resource sharing passes. In some cases this drastically reduces the number of required ports on the memory cell.

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

- $\MEMID$
  The name of the RTLIL::Memory object that is associated with this read port.

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

- $\WIDTH$
  The number of data bits (width of the $\DATA$ output port).

- $\CLK_ENABLE$
  When this parameter is non-zero, the clock is used. Otherwise this read port is asynchronous and the $\CLK$ input is not used.

- $\CLK_POLARITY$
  Clock is active on the positive edge if this parameter has the value $1'b1$ and on the negative edge if this parameter is $1'b0$.

- $\TRANSPARENT$
  If this parameter is set to $1'b1$, a read and write to the same address in the same cycle will return the new value. Otherwise the old value is returned.

The $memwr$ 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 read port.

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

- $\WIDTH$
  The number of data bits (width of the $\DATA$ output port).

- $\CLK_ENABLE$
  When this parameter is non-zero, the clock is used. Otherwise this read port is asynchronous and the $\CLK$ input is not used.

- $\CLK_POLARITY$
  Clock is active on positive edge if this parameter has the value $1'b1$ and on the negative edge if this parameter is $1'b0$.

- $\PRIORITY$
  The cell with the higher integer value in this parameter wins a write conflict.

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

The $mem$ cell provides the following parameters:
CHAPTER 5. INTERNAL CELL LIBRARY

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

- **\$SIZE**
  The number of words in the memory.

- **\$ABITS**
  The number of address bits.

- **\$WIDTH**
  The number of data bits per word.

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

- **\$RD_CLK_ENABLE**
  This parameter is $\$RD_PORTS$ bits wide, containing a clock enable bit for each read port.

- **\$RD_CLK_POLARITY**
  This parameter is $\$RD_PORTS$ bits wide, containing a clock polarity bit for each read port.

- **\$RD_TRANSPARENT**
  This parameter is $\$RD_PORTS$ bits wide, containing a transparent bit for each read port.

- **\$WR_PORTS**
  The number of write ports on this memory cell.

- **\$WR_CLK_ENABLE**
  This parameter is $\$WR_PORTS$ bits wide, containing a clock enable bit for each write port.

- **\$WR_CLK_POLARITY**
  This parameter is $\$WR_PORTS$ bits wide, containing a clock polarity bit for each write port.

The $mem cell has the following ports:

- **\$RD_CLK**
  This input is $\$RD_PORTS$ bits wide, containing all clock signals for the read ports.

- **\$RD_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.

- **\$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 techmap pass can be used to manually map $mem cells to specialized memory cells on the target architecture, such as block ram resources on an FPGA.
5.1.6 Finite State Machines

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

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.

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

The cell types $_DFF_NNO_, $_DFF_NN1_, $_DFF_NPO_, $_DFF_NP0_, $_DFF_NP1_, $_DFF_PP0_ and $_DFF_PP1_ implement d-type flip-flops with asynchronous resets. The values in the table for these cell types relate to the following Verilog code template, where RstEdge is posedge if RstLvl is 1, and needge otherwise.

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

In most cases gate level logic networks are created from RTL networks using the techmap pass. The flip-flop cells from the gate level logic network can be mapped to physical flip-flop cells from a Liberty file using the dfflibmap pass. The combinatorial logic cells can be mapped to physical cells from a Liberty file via ABC [27] using the abc pass.

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

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.
FIXME:
Add information about $ff$ and $_FF_ cells.

FIXME:
Add information about $dffe$, $dffsr$, $dlatch$, and $dlatchsr$ cells.

FIXME:
Add information about $_DFFE_???_, $_DFFSR_???_, $_DLATCH_?_, and $_DLATCHSR_???_ cells.

FIXME:
Add information about $_NAND_$, $_NOR_$, $_XNOR_$, $_ANDNOT_$, $_ORNOT_$, $_AOI3_$, $_OAI3_$, $_AOI4_$, and $_OAI4_$ 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 Yosys Subreddit.

6.1 The “CodingReadme” File

The following is an excerpt of the CodingReadme file from the Yosys source tree.

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

```c++
#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");
    }
};

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:

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

And then executed using the following command:

```bash
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:

```c++
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).
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
```

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.

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 contain 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.

```
#include "kernel/yosys.h"
#include "kernel/sigtools.h"
#include <string>
#include <map>
#include <set>

USING_YOSYS_NAMESPACE
PRIVATE_NAMESPACE_BEGIN

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
            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_)
                {  
```

CHAPTER 6. PROGRAMMING YOSYS EXTENSIONS

```
```
CHAPTER 6. PROGRAMMING YOSYS EXTENSIONS

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\n", 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) YS_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);
enmodule
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](image-url)
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/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 Synopsys 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/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:
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 `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:

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

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
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 \(~, \&, |, ^\)
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 \(<\ll, \gg, \ll\ll\), and \(\gg\)
AST_LT, AST_LE, AST_EQ, AST_NE, AST_GE, AST_GT | The relational operators \(<, \leq, =\), \(\neq\), \(\geq\), and \(>\)
AST_ADD, AST_SUB, AST_MUL, AST_DIV, AST_MOD, AST_POW | The binary operators \(+, -, \times, /, \%\), 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 (\(\ll\)) 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 $<number>\langleoriginal_name>$ for each of the assigned signals.

• Then an RTLIL::Process is created that assigns all intermediate values for each left-hand-side signal to the temporary signal in its RTLIL::CaseRule/RTLIL::SwitchRule tree.

• Finally a RTLIL::SyncRule is created for the RTLIL::Process that assigns the temporary signals for the final values to the actual signals.

• Calls to AST::AstNode::genRTLIL() are generated for right hand sides as needed. When blocking assignments are used, AST::AstNode::genRTLIL() is configured using global variables to use the temporary signals that hold the correct intermediate values whenever one of the previously assigned signals is used in an expression.

Unfortunately the generation of a correct RTLIL::CaseRule/RTLIL::SwitchRule tree for behavioural code is a non-trivial task. The AST frontend solves the problem using the approach described on the following pages. The following example illustrates what the algorithm is supposed to do. Consider the following Verilog code:
always @(posedge clock) begin
    out1 = in1;
    if (in2)
        out1 = !out1;
    out2 <= out1;
    if (in3)
        out2 <= out2;
    if (in4)
        if (in5)
            out3 <= in6;
        else
            out3 <= in7;
        out1 = out1 ^ out2;
end

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

cell $logic_not $logic_not$<input>:4$2
    connect A \in1
    connect Y $logic_not$<input>:4$2_Y
end

cell $xor $xor$<input>:13$3
    connect A \in1\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
                end
        case
        end
    sync posedge \clock
    update \out1 $0\out1[0:0]
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:

- `substitution_rvalue_from` and `substitution_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.

- `substitution_lvalue_from` and `substitution_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`. 
CHAPTER 7. THE VERILOG AND AST FRONTENDS

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.
CHAPTER 7. THE VERILOG AND AST FRONTENDS

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

7.3.2 The proc pass

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

- **proc_clean and proc_rmdead**
  These two passes just clean up the RTLIL::Process structure. The proc_clean pass removes empty parts (eg. empty assignments) from the process and proc_rmdead detects and removes unreachable branches from the process's decision trees.

- **proc_arst**
  This pass detects processes that describe d-type flip-flops with asynchronous resets and rewrites the process to better reflect what they are modelling: Before this pass, an asynchronous reset has two edge-sensitive sync rules and one top-level RTLIL::SwitchRule for the reset path. After this pass the sync rule for the reset is level-sensitive and the top-level RTLIL::SwitchRule has been removed.

- **proc_mux**
  This pass converts the RTLIL::CaseRule/RTLIL::SwitchRule-tree to a tree of multiplexers per written signal. After this, the RTLIL::Process structure only contains the RTLIL::SyncRules that describe the output registers.

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

- **proc_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].

62
CHAPTER 8. OPTIMIZATIONS

<table>
<thead>
<tr>
<th>A-Input</th>
<th>B-Input</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>any</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X/Z</td>
<td>X/Z</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X/Z</td>
<td>X</td>
</tr>
<tr>
<td>X/Z</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>any</td>
<td>X/Z</td>
<td>0</td>
</tr>
<tr>
<td>X/Z</td>
<td>any</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

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 $seq 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:

```verbatim
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 ($\text{DFF}_*$, $\text{dff}$, and $\text{adff}$ 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 $\text{unused_bits}$ 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 $\text{-nomux}$ can be used to disable resource sharing for multiplexer cells ($\text{mux}$ and $\text{pmux}$. This can be useful as it prevents multiplexer trees to be merged, which might prevent $\text{opt_muxtree}$ to identify possible optimizations.

8.2 FSM Extraction and Encoding

The $\text{fsm}$ pass performs finite-state-machine (FSM) extraction and recoding. The $\text{fsm}$ pass simply executes the following other passes:

- Identify and extract FSMs:
  - $\text{fsm_detect}$
  - $\text{fsm_extract}$
- Basic optimizations:
  - $\text{fsm_opt}$
  - $\text{opt_clean}$
  - $\text{fsm_opt}$
- Expanding to nearby gate-logic (if called with $\text{-expand}$):
  - $\text{fsm_expand}$
  - $\text{opt_clean}$
  - $\text{fsm_opt}$
- Re-code FSM states (unless called with $\text{-norecode}$):
  - $\text{fsm_recode}$
- Print information about FSMs:
  - $\text{fsm_info}$
- Export FSMs in KISS2 file format (if called with $\text{-export}$):
  - $\text{fsm_export}$
CHAPTER 8. OPTIMIZATIONS

- 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 $seq 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
CHAPTER 8. OPTIMIZATIONS

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 \texttt{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 \texttt{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 \texttt{abc} pass extracts the combinational gate-level parts of the design, passes it through ABC, and re-integrates the results. The \texttt{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.145). 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 Clifford Wolf and based on the Ullmann Subgraph Isomorphism Algorithm [Ull76]. It is used by the `extract` pass (see `help extract` or Sec. C.45).

A.4 ezSAT

The files in `libs/ezsat` provide a library for simplifying generating CNF formulas for SAT solvers. It also contains bindings of MiniSAT. The ezSAT library is written by Clifford Wolf. It is used by the `sat` pass (see `help sat` or Sec. C.116).
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 unmodified copy of ABC [27]. 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 without -constr:
strash; ifraig; scorr; dc2; dretime; strash; &get -n; &dch -f;
&nf (D); &put

for -liberty 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:
```

72
APPENDIX C. COMMAND REFERENCE MANUAL

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

(replaces \{P\} in the default scripts above)

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

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

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

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

-\texttt{sop}
  map to sum-of-product cells and inverters

-\texttt{g} \texttt{type1,type2,...}
  Map to the specified list of gate types. Supported gates types are:
  AND, NAND, OR, NOR, XOR, XNOR, ANDNOT, ORNOT, MUX, 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:
  \texttt{simple}: AND OR XOR MUX
  \texttt{cmos2}: NAND NOR
  \texttt{cmos3}: NAND NOR AOI3 OAI3
  \texttt{cmos4}: NAND NOR AOI3 AOI4 OAI4
  \texttt{gates}: AND NAND OR NOR XOR XNOR ANDNOT ORNOT
  \texttt{aig}: AND NAND OR NOR ANDNOT ORNOT

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

-\texttt{dff}
  also pass \$_\texttt{DFF}_?$_ and \$_\texttt{DFFE}_??$_ cells through ABC. modules with many
  clock domains are automatically partitioned in clock domains and each
  domain is passed through ABC independently.

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

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

-\texttt{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.

When neither -liberty nor -lut is used, 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 the load that into ABC externally if
you want to use ABC to convert your design into another format.


C.2 add – add objects to the design

```plaintext
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.

```plaintext
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.

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

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

C.3 aigmap – map logic to and-inverter-graph circuit

```plaintext
aigmap [options] [selection]
```

Replace all logic cells with circuits made of only $\text{AND}_$ and
$\text{NOT}_$ cells.

```plaintext
-nand
```
Enable creation of $_NAND_ cells

C.4 aluacc – extract ALU and MACC cells

```plaintext
aluacc [selection]
```

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

C.5 assertpmux – convert internal signals to module ports

```plaintext
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 be the mux tree it drives. this option will deactivate this additional constrained and check the $pmux$ condition always.

C.6 async2sync – convert async FF inputs to sync circuits

```plaintext
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.

Currently only $adff$ cells are supported by this pass.

C.7 attrmap – renaming attributes

```plaintext
attrmap [options] [selection]
```

This command renames attributes and/or mapps key/value pairs to other key/value pairs.
APPENDIX C. COMMAND REFERENCE MANUAL

- 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.8 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, attributes 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 <attribname>**
  Move or copy this attribute. This option can be used multiple times.
C.9  blackbox – change type of cells in the design

blackbox [options] [selection]
Convert modules into blackbox modules (remove contents and set the blackbox module attribute).

C.10  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.11  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

When called with -noinit then this command also checks for wires which have the 'init' attribute set.

When called with -initdrv then this command also checks for wires which have the 'init' attribute set and aren't driven by a FF cell type.

When called with -assert then the command will produce an error if any problems are found in the current design.
C.12 chformal – change formal constraints of the design

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

Make changes to the formal constraints of the design. The [types] options determine the type of constraint to operate on. If none of the following options is 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 change the roles of cells as indicated. this options can be combined
- assume2assert
- live2fair
- fair2live

C.13 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.14 chtype – change type of cells in the design
APPENDIX C. COMMAND REFERENCE MANUAL

CHTYPE

```plaintext
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.15 clean – remove unused cells and wires

```plaintext
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.16 clk2fflogic – convert clocked FFs to generic $ff cells

```plaintext
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.17 connect – create or remove connections

```plaintext
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] -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 to 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.

This command does not operate on module with processes.

C.18 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.19 coolrunner2_sop – break $sop cells into ANDTERM/ORTERM cells

coolrunner2_sop [options] [selection]

Break $sop cells into ANDTERM/ORTERM cells.

C.20 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.21 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.22 delete – delete objects in the design

```
delete [selection]
```

Deletes the selected objects. This will also remove entire modules, if the whole module is selected.
APPENDIX C. COMMAND REFERENCE MANUAL

|   | delete {-input|-output|-port} [selection] |
|---|------------------------------------------|
| 8 | Does not delete any object but removes the input and/or output flag on the selected wires, thus 'deleting' module ports. |

**C.23 deminout – demote inout ports to input or output**

<table>
<thead>
<tr>
<th></th>
<th>deminout [options] [selection]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&quot;Demote&quot; inout ports to input or output ports, if possible.</td>
</tr>
</tbody>
</table>

**C.24 design – save, restore and reset current design**

<table>
<thead>
<tr>
<th></th>
<th>design -reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Clear the current design.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>design -save &lt;name&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Save the current design under the given name.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>design -stash &lt;name&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Save the current design under the given name and then clear the current design.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>design -push</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Push the current design to the stack and then clear the current design.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>design -pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Reset the current design and pop the last design from the stack.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>design -load &lt;name&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Reset the current design and load the design previously saved under the given name.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>design -copy-from &lt;name&gt; [-as &lt;new_mod_name&gt;] &lt;selection&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>Copy modules from the specified design into the current one. The selection is evaluated in the other design.</td>
</tr>
</tbody>
</table>
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.

---

C.25  dff2dffe – transform $dff cells to $dffe cells

dff2dffe [options] [selection]
This pass transforms $dff cells driven by a tree of multiplexers with one or
more feedback paths to $dffe cells. It also works on gate-level cells such as
$_DFF_P_$, $_DFF_N_ and $_MUX_.$.

- unmap
  operate in the opposite direction: replace $dffe$ cells with combinations
  of $dff$ and $mux$ cells. the options below are ignore in unmap mode.

- direct <internal_gate_type> <external_gate_type>
  map directly to external gate type. <internal_gate_type> can
  be any internal gate-level FF cell (except $_DFFE_??_). the
  <external_gate_type> is the cell type name for a cell with an
  identical interface to the <internal_gate_type>, except it
  also has an high-active enable port 'E'.
  Usually <external_gate_type> is an intermediate cell type
  that is then translated to the final type using 'techmap'.

- direct-match <pattern>
  like -direct for all DFF cell types matching the expression.
  this will use $__DFFE_* as <external_gate_type> matching the
  internal gate type $_DFF_*$, and $__DFFSE_* for those matching
  $_DFFS_*$, except for $_DFF_[NP]*$, which is converted to
  $_DFFE_[NP]*$.

C.26  dff2dffs – process sync set/reset with SR over CE priority

dff2dffs [options] [selection]
Merge synchronous set/reset \_MUX\_ cells to create \$__DFFS\_[NP][NP][01]\_, to be run before dff2dffe for SR over CE priority.

C.27 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.)

C.28 dfflibmap – technology mapping of flip-flops

```
dfflibmap [-prepare] -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.

C.29 dffsr2dff – convert DFFSR cells to simpler FF cell types

```
dffsr2dff [options] [selection]
```

This pass converts DFFSR cells ($dffsr, \$_DFFSR_???\_) and ADFF cells ($adff, \$_DFF_???\_) to simpler FF cell types when any of the set/reset inputs is unused.

C.30 dump – print parts of the design in ilang format
dump [options] [selection]
Write the selected parts of the design to the console or specified file in
ilang format.

-\(m\)
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.31  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.32  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.33  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.34 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.35 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.
APPENDIX C. COMMAND REFERENCE MANUAL

C.36  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 subcircuits. The integer attribute 'equiv_region' is set on all
wires and cells.

C.37  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.38  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.39  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.40  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.41  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.42  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
had $\equiv$ 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 performs 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.43 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.44 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.45 extract – find subcircuits and replace them with cells

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

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

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

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

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

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

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

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

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

C.46  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

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

C.47  **extract_fa** – find and extract full/half adders

```bash
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.48  **extract_reduce** – converts gate chains into $reduce_* cells

```bash
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.49 flatten – flatten design

```
flatten [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.

C.50 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.51 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`
C.52 fsm_detect – finding FSMs in design

```bash
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.53 fsm_expand – expand FSM cells by merging logic into it

```bash
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.54  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.55  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.56  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.57  fsm_map – mapping FSMs to basic logic

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

C.58  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.59  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.60  greenpak4_dffinv – merge greenpak4 inverters and DF-F/latches

greenpak4_dffinv [options] [selection]
Merge GP_INV cells with GP_DFF* and GP_DLATCH* cells.
C.61 help – display help messages

<table>
<thead>
<tr>
<th>Help options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help ..........</td>
<td>list all commands</td>
</tr>
<tr>
<td>help &lt;command&gt;</td>
<td>print help message for given command</td>
</tr>
<tr>
<td>help -all . . .</td>
<td>print complete command reference</td>
</tr>
<tr>
<td>help -cells ...</td>
<td>list all cell types</td>
</tr>
<tr>
<td>help &lt;celltype&gt; ..</td>
<td>print help message for given cell type</td>
</tr>
<tr>
<td>help &lt;celltype&gt;+</td>
<td>print verilog code for given cell type</td>
</tr>
</tbody>
</table>

C.62 hierarchy – check, expand and clean up design hierarchy

In parametric designs, a module might exist 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.

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

- **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.

- **nokeep_asserts**
  - per default this pass sets the “keep” attribute on all modules that directly or indirectly contain one or more $assert cells. This option disables this behavior.

- **top <module>**
use the specified top module to build a 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.

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.63  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.64  history – show last interactive commands

history

This command prints all commands in the shell history buffer. This are
APPENDIX C. COMMAND REFERENCE MANUAL

all commands executed in an interactive session, but not the commands
from executed scripts.

C.65 ice40_ffinit – iCE40: handle FF init values

```
  ice40_ffinit [options] [selection]
```

Remove zero init values for FF output signals. Add inverters to implement
nonzero init values.

C.66 ice40_ffssr – iCE40: merge synchronous set/reset into FF cells

```
  ice40_ffssr [options] [selection]
```

Merge synchronous set/reset $MUX_ cells into iCE40 FFs.

C.67 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_rmff
    opt_clean
  while <changed design>
```

When called with the option -unlut, this command will transform all already
mapped SB_LUT4 cells back to logic.

C.68 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.69  iopadmap – technology mapping of i/o pads (or buffers)

```
iopadmap [options] [selection]

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

-`inpad <celltype> <portname>:<portname>`
   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> <portname>:<portname>`
-`inoutpad <celltype> <portname>:<portname>`
   Similar to -inpad, but for output and inout ports.

-`toutpad <celltype> <portname>:<portname>`
   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.

-`tinoutpad <celltype> <portname>:<portname>:<portname>`
   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.

-`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.)
```

C.70  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
```
C.71 log – print text and log files

```plaintext
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.72 ls – list modules or objects in modules

```plaintext
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.73 ltp – print longest topological path

```plaintext
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.74 **lut2mux** – convert $lut$ to $\_MUX\_$

```
lut2mux [options] [selection]
This pass converts $lut$ cells to $\_MUX\_$ gates.
```

C.75 **macmap** – 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.76 **memory** – translate memories to basic cells

```
memory [-nomap] [-nordff] [-memx] [-bram <bram_rules>] [selection]
This pass calls all the other memory_* passes in a useful order:
memory_dff [-nordff] (-memx implies -nordff)
opt_clean
memory_share
opt_clean
memory_memx (when called with -memx)
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.77 **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 a set of block ram description and a sequence of match rules. 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
```
```
APPENDIX C. COMMAND REFERENCE MANUAL

15 ports 1 1  # number of ports in each group
16 wrmode 1 0  # set to '1' if this groups is write ports
17 enable 4 1  # number of enable bits
18 transp 0 2  # transparent (for read ports)
19 clocks 1 2  # clock configuration
20 clkpol 2 2  # clock polarity configuration
21 endbram
22
23 For the option 'transp' the value 0 means non-transparent, 1 means transparent
24 and a value greater than 1 means configurable. All groups with the same
25 value greater than 1 share the same configuration bit.
26
27 For the option 'clocks' the value 0 means non-clocked, and a value greater
28 than 0 means clocked. All groups with the same value share the same clock
29 signal.
30
31 For the option 'clkpol' the value 0 means negative edge, 1 means positive edge
32 and a value greater than 1 means configurable. All groups with the same value
33 greater than 1 share the same configuration bit.
34
35 Using the same bram name in different bram blocks will create different variants
36 of the bram. Verilog configuration parameters for the bram are created as needed.
37
38 It is also possible to create variants by repeating statements in the bram block
39 and appending '@<label>' to the individual statements.
40
41 A match rule looks like this:
42
43 match RAMB1024X32
44    max waste 16384  # only use this bram if <= 16k ram bits are unused
45    min efficiency 80  # only use this bram if efficiency is at least 80%
46 endmatch
47
48 It is possible to match against the following values with min/max rules:
49
50 words ........ number of words in memory in design
51 abits ........ number of address bits on memory in design
52 dbits ........ number of data bits on memory in design
53 wports ........ number of write ports on memory in design
54 rports ........ number of read ports on memory in design
55 ports ........ number of ports on memory in design
56 bits ........ number of bits in memory in design
57 dups ........ number of duplications for more read ports
58
59 awaste ....... number of unused address slots for this match
60 dwaste ....... number of unused data bits for this match
61 bwaste ....... number of unused bram bits for this match
62 waste ........ total number of unused bram bits (bwaste*dups)
63 efficiency ... total percentage of used and non-duplicated bits
64
65 acells ....... number of cells in 'address-direction'
66 dcells ....... number of cells in 'data-direction'
67 cells ........ total number of cells (acells+dcells+dups)
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.78 memory_collect – creating multi-port memory cells

memory_collect [selection]

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

C.79 memory_dff – merge input/output DFFs into memories

memory_dff [options] [selection]

This pass detects DFFs at memory 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.

-nordfff
do not merge registers on read ports

C.80 memory_map – translate multiport memories to basic cells

memory_map [selection]

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

C.81 memory_memx – emulate vlog sim behavior for mem ports
APPENDIX C. COMMAND REFERENCE MANUAL

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

C.82 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 calling memory_dff with -nordff.

C.83 memory_share – consolidate memory ports

memory_share [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 write ports are connected to async read ports accessing the same address, then this feedback path is converted to a write port with byte/part enable signals.
- 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 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.

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.84 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.85 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; 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; opt_expr -keepdc -undriven;;' on the miter circuit.

C.86 muxcover – cover trees of MUX cells with wider MUXes

`muxcover [options] [selection]`

Cover trees of $MUX_ cells with $MUX\{4,8,16\}_ cells

- `-mux4, -mux8, -mux16`
  Use the specified types of MUXes. If none of those options are used, the effect is the same as if all of them where used.

- `-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.

C.87 nlutmap – map to LUTs of different sizes

```bash
nlutmap [options] [selection]
```

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

```bash
-luts N_1,N_2,N_3,...
```

The number of LUTs with 1, 2, 3, ... inputs that are available in the target architecture.

```bash
-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.88 opt – perform simple optimizations

```bash
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:

```bash
opt_merge [-share_all] -nomux
```

```bash
do
```

```bash
opt_muxtree
```

```bash
opt_reduce [-fine] [-full]
```

```bash
opt_merge [-share_all]
```

```bash
opt_rmddf [-keepdc]
```

```bash
opt_clean [-purge]
```

```bash
```

```bash
while <changed design>
```

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

```bash
do
```

```bash
```

```bash
opt_merge [-share_all]
```

```bash
opt_rmddf [-keepdc]
```

```bash
opt_clean [-purge]
```

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

C.89 opt_clean – remove unused cells and wires

```sh
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.90 opt_demorgan – Optimize reductions with DeMorgan equivalents

```sh
opt_demorgan [selection]
```

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

C.91 opt_expr – perform const folding and simple expression rewriting

```sh
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

- clkinv
  optimize clock inverters by changing FF types

- fine
C.92  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.93  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.94  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
C.95  opt_rmdff – remove DFFs with constant inputs

```
opt_rmdff [-keepdc] [selection]
```

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

C.96  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.97  pmuxtree – transform $pmux cells to trees of $mux cells

```
pmuxtree [options] [selection]
```

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

C.98  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
    do not pass -nordff to '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 -keepdc
    wreduce [-memx]
    memory_dff [-nordff]
    memory_memx (if -memx)
    opt_clean
    memory_collect
    opt -keepdc -fast

check:
    stat
    check
C.99 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_init
proc_arst
proc_mux
proc_dlatch
procdff
proc_clean

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

The following options are supported:

- `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.
```

C.100 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.101 proc_clean – remove empty parts of processes

```
proc_clean [selection]

This pass removes empty parts of processes and ultimately removes a process if it contains only empty structures.
C.102 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.103 proc_dlatch – extract latches from processes

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

C.104 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.105 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.106 proc_rmdead – eliminate dead trees in decision trees

```
proc_rmdead [selection]
This pass identifies unreachable branches in decision trees and removes them.
```
C.107 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.108 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 {-vhdl87|-vhdl93|-vhdl2k|-vhdl2008|-vhdl} <vhdl-file>..
```

Load the specified VHDL files. (Requires Verific.)

```
read -define <macro>[=<value>]..
```

Set global Verilog/SystemVerilog defines.

```
read -undef <macro>..
```

Unset global Verilog/SystemVerilog defines.
APPENDIX C. COMMAND REFERENCE MANUAL

24
25    read -incdir <directory>
26
27 Add directory to global Verilog/SystemVerilog include directories.

C.109 read_blif – read BLIF file

1    read_blif [filename]
2
3 Load modules from a BLIF file into the current design.
4
5      -sop
6      Create $sop cells instead of $lut cells
7
8      -wideports
9      Merge ports that match the pattern 'name[int]' into a single
10     multi-bit port 'name'.

C.110 read_ilang – read modules from ilang file

1    read_ilang [filename]
2
3 Load modules from an ilang file to the current design. (ilang is a text
4 representation of a design in yosys’s internal format.)

C.111 read_json – read JSON file

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

C.112 read_liberty – read cells from liberty file

1    read_liberty [filename]
2
3 Read cells from liberty file as modules into current design.
4
5      -lib
6      only create empty blackbox modules
7
8      -nooverwrite
9      ignore re-definitions of modules. (the default behavior is to
10     create an error message if the existing module is not a blackbox
11     module, and overwrite the existing module if it is a blackbox module.)
APPENDIX C. COMMAND REFERENCE MANUAL

-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.113 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

-norestrict
  ignore restrict() assertions

-assume-asserts
  treat all assert() statements like assume() statements

-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_vlog
  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

-lib
  only create empty blackbox modules. This implies -DBLACKBOX.

-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
APPENDIX C. COMMAND REFERENCE MANUAL

-`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. 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.114  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 -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 an 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.115 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.116 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>
-set-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.

-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.

-steplsize <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.117 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.118 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> SSCs. A different number of SSCs 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

C.119 script – execute commands from script file

script <filename> [<from_label>:<to_label>]

This command executes the yosys commands in the specified file.
The 2nd argument can be used to only execute the section of the
file between the specified labels. An empty from label is synonymous
for the beginning of the file and an empty to label is synonymous
for 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.

C.120 select – modify and view the list of selected objects

select [ -add | -del | -set <name> ] {-read <filename> | <selection>}
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 YS_OVERRIDE the global selection for the command. The syntax of this
optional argument is identical to the syntax of the <selection> argument
-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> %"

-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 command has been executed, the union of all remaining sets

on the stack is computed and used as selection for the command.

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

<mod_pattern>

select the specified module(s)

<mod_pattern>/<obj_pattern>

select the specified object(s) from the module(s)

Pushing (selecting) object when in -module mode:

<obj_pattern>

select the specified object(s) from the current module

A <mod_pattern> can be a module name, 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

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

%<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.

%e
  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.121 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.122 setparam – set/unset parameters on objects
APPENDIX C. COMMAND REFERENCE MANUAL

```
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.123 setundef – replace undefined values with defined constants

```
setundef [options] [selection]
```

This command replaces undefined (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 undefined (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

### C.124 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.125 shell – enter interactive command mode

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

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 busses 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.
```
C.127 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.128 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

-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)
C.129 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, $dffsr, $adff, $dlatch

C.130 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.131 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.132 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
APPENDIX C. COMMAND REFERENCE MANUAL

- width
  annotate internal cell types with their word width.
  e.g. $add_8 for an 8 bit wide $add cell.

C.133 submod – moving part of a module to a new submodule

submod [-copy] [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.

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

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.

C.134 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'

-nofsm
APPENDIX C. COMMAND REFERENCE MANUAL

20 do not run FSM optimization
21
22 -noabc
23 do not run abc (as if yosys was compiled without ABC support)
24
25 -noalumacc
26 do not run 'alumacc' pass. i.e. keep arithmetic operators in
27 their direct form ($add, $sub, etc.).
28
29 -nordff
30 passed to 'memory'. prohibits merging of FFs into memory read ports
31
32 -noshare
33 do not run SAT-based resource sharing
34
35 -run <from_label>[:<to_label>]
36 only run the commands between the labels (see below). an empty
37 from label is synonymous to 'begin', and empty to label is
38 synonymous to the end of the command list.
39
40 The following commands are executed by this synthesis command:
41
42 begin:
43 hierarchy -check [-top <top> | -auto-top]
44
45 coarse:
46 proc
47 flatten (if -flatten)
48 opt_expr
49 opt_clean
50 check
51 opt
52 wreduce
53 alumacc
54 share
55 opt
56 fsm
57 opt -fast
58 memory -nomap
59 opt_clean
60
61 fine:
62 opt -fast -full
63 memory_map
64 opt -full
65 techmap
66 opt -fast
67 abc -fast
68 opt -fast
69
70 check:
71 hierarchy -check
72 stat
73
137
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 option

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
  dffsr2dff
  dff2dffe -direct-match $_DFF_#
  opt -fine
  techmap -map +/techmap.v
  opt -full
```
APPENDIX C. COMMAND REFERENCE MANUAL

48  clean -purge
49  setundef -undriven -zero
50  abc -markgroups -dff (only if -retime)

52  map_luts:
53    abc -lut 4
54    clean

56  map_cells:
57    iopadmap -bits -outpad $__outpad I:O -inpad $__inpad 0:I
58    techmap -map +/achronix/speedster22i/cells_map.v
59    clean -purge

61  check:
62    hierarchy -check
63    stat
64    check -noinit

66  vout:
70    write_verilog -nodec -attr2comment -defparam -renameprefix syn_ <file-name>

C.136 synth_coolrunner2 – synthesis for Xilinx Coolrunner-II CPLDs

1  synth_coolrunner2 [options]

3  This command runs synthesis for Coolrunner-II CPLDs. This work is experimental.
4  It is intended to be used with https://github.com/azonenberg/openfpga as the
5  place-and-route.

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

10 -json <file>
11    write the design to the specified JSON file. Writing of an output file
12    is omitted if this parameter is not specified.

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

19 -noflatten
20    do not flatten design before synthesis

23 -retime
24    run 'abc' with -dff option

26 The following commands are executed by this synthesis command:
27  begin:
APPENDIX C. COMMAND REFERENCE MANUAL

```plaintext
read_verilog -lib +/coolrunner2/cells_sim.v
hierarchy -check -top <top>

flattens: (unless -noflatten)
  proc
  flatten
  tribuf -logic

coarse:
  synth -run coarse

fine:
  opt -fast -full
  techmap
  techmap -map +/coolrunner2/cells_latch.v
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
iopadmap -bits -inpad IBUF 0:I -outpad IOBUF 0:IO -inooutpad IOBUFE 0:IO -outpad IOBUF 0:IO
attrmvcp -attr src -attr LOC t:IOBUF n:*
attrmvcp -attr src -attr LOC -driven t:IBUF n:
splitnets
  clean

check:
  hierarchy -check
  stat
  check -noinit

json:
  write_json <file-name>
```

C.137 synth__easic – synthesis for eASIC platform

```plaintext
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 option

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 (only if -retime)
  opt_clean (only if -retime)

map:
  dfflibmap -liberty <etools_phys_clk_lib>
  abc -liberty <etools_logic_lut_lib>
  opt_clean

check:
```
C.138 synth_ecp5 – synthesis for ECP5 FPGAs

```bash
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

- retime
  run 'abc' with -dff option

- noccu2
  do not use CCU2 cells in output netlist

- nodffe
  do not use flipflops with CE in output netlist

- nobram
  do not use BRAM cells in output netlist

- nodram
  do not use distributed RAM cells in output netlist

- nomux
```
do not use PFU muxes to implement LUTs larger than LUT4s

-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)

The following commands are executed by this synthesis command:

begin:
  read_verilog -lib +/ecp5/cells_sim.v
  hierarchy -check -top <top>
flattens: (unless -noflatten)
  proc
  flatten
  tribuf -logic
  deminout
coarse:
  synth -run coarse
bram:  (skip if -nobram)
dram:  (skip if -nodram)
  memory_bram -rules +/ecp5/dram.txt
  techmap -map +/ecp5/drams_map.v
fine:
  opt -fast -mux_undef -undriven -fine
  memory_map
  opt -undriven -fine
  techmap -map +/techmap.v -map +/ecp5/arith_map.v
  abc -dff  (only if -retim)
map_ffs:
  dffsr2dff
  dff2dffs
  opt_clean
  dff2dffe -direct-match $_DFF_* -direct-match $_DFFS_*
  techmap -D NO_LUT -map +/ecp5/cells_map.v
  opt_expr -mux_undef
  simplemap
map_luts:
  abc  (only if -abc2)
  abc -lut 4:7
  clean
map_cells:
  techmap -map +/ecp5/cells_map.v  (with -D NO_LUT in vpr mode)
C.139 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.

- **-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.

- **-retime**
  
  run 'abc' with -dff option

The following commands are executed by this synthesis command:

- **begin:**
  
  read_verilog -lib +/gowin/cells_sim.v
  hierarchy -check -top <top>

- **flatten:**
  
  proc
  flatten
  tribuf -logic
  deminout
```
coarse:
    synth -run coarse

fine:
    opt -fast -mux_undef -undriven -fine
    memory_map
    opt -undriven -fine
    techmap
    clean -purge
    splitnets -ports
    setundef -undriven -zero
    abc -dff (only if -ret ime)

map_luts:
    abc -lut 4
    clean

map_cells:
    techmap -map +/gowin/cells_map.v
    hilomap -hicell VCC V -locell GND G
    iopadmap -ipad IBUF 0:I -opad OBUF I:O
    clean -purge

check:
    hierarchy -check
    stat
    check -noinit

vout:
    write_verilog -nodec -attr2comment -defparam -renameprefix gen <file-name>

---

C.140 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 option

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
    techmap -map +/greenpak4/cells_latch.v
    dfflibmap -prepare -liberty +/greenpak4/gp_dff.lib
    opt -fast
    abc -dff  (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
    greenpak4_dffinv
APPENDIX C. COMMAND REFERENCE MANUAL

74    clean
75
76    check:
77    hierarchy -check
78    stat
79    check -noinit
80
81    json:
82    write_json <file-name>

C.141 synth_ice40 – synthesis for iCE40 FPGAs

synth_ice40 [options]
This command runs synthesis for iCE40 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

- retime
  run 'abc' with -dff option

- nocarry
  do not use SB_CARRY cells in output netlist

- nodffe
  do not use SB_DFFE* cells in output netlist

- nobram
  do not use SB_RAM40_4K* cells in output netlist

- abc2
APPENDIX C. COMMAND REFERENCE MANUAL

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)

The following commands are executed by this synthesis command:

begin:
  read_verilog -lib +/ice40/cells_sim.v
  hierarchy -check -top <top>

flatten: (unless -noflatten)
  proc
  flatten
  tribuf -logic
  deminout

coarse:
  synth -run coarse

bram: (skip if -nobram)
  memory_bram -rules +/ice40/brams.txt
  techmap -map +/ice40/brams_map.v

fine:
  opt -fast -mux_undef -undriven -fine
  memory_map
  opt -undriven -fine
  techmap -map +/techmap.v -map +/ice40/arith_map.v
  abc -dff (only if -retime)
  ice40_opt

map_ffs:
  dffsr2dff
  dff2dffe -direct-match $_DFF_*
  techmap -D NO_LUT -map +/ice40/cells_map.v
  opt_expr -mux_undef
  simplemap
  ice40_ffinit
  ice40_ffssr
  ice40_opt -full

map_luts:
  abc (only if -abc2)
  ice40_opt (only if -abc2)
  techmap -map +/ice40/latches_map.v
  abc -lut 4
  clean

map_cells:
  techmap -map +/ice40/cells_map.v (with -D NO_LUT in vpr mode)
  clean
check:
  hierarchy -check
  stat
  check -noinit

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.142 synth_intel – synthesis for Intel (Altera) FPGAs.

```bash
synth_intel [options]

This command runs synthesis for Intel FPGAs.

-famly < max10 | a10gx | cyclone10 | cyclonev | cycloneiv | cycloneive>
generate the synthesis netlist for the specified family.
MAX10 is the default target if not family argument specified.
For Cyclone GX devices, use cycloneiv argument; For Cyclone E, use cycloneive.
Cyclone V and Arria 10 GX devices are experimental, use it with a10gx argument.

-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.

-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.

-noiopads
do not use altsyncram cells in output netlist

-nobram
do not use altsyncram cells in output netlist
```
APPENDIX C. COMMAND REFERENCE MANUAL

-noflatten
  do not flatten design before synthesis

-retime
  run 'abc' with -dff option

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

course:
  synth -run coarse

bram: (skip if -nobram)
  memory_bram -rules +/intel/common/brams.txt
  techmap -map +/intel/common/brams_map.v

fine:
  opt -fast -mux_undef -undriven -fine -full
  memory_map
  opt -undriven -fine
  dffsr2dff
  dff2dffe -direct-match $DFF_*
  opt -fine
  techmap -map +/techmap.v
  opt -full
  clean -purge
  setundef -undriven -zero
  abc -markgroups -dff (only if -ret ime)

map_luts:
  abc -lut 4
  clean

map_cells:
  iopadmap -bits -outpad $__outpad I:O -inpad $__inpad 0:I (unless -noiopads)
  techmap -map +/intel/max10/cells_map.v
  dffinit -highlow -ff dffeas q power_up
  clean -purge

check:
  hierarchy -check
C.143  \texttt{synth\_xilinx} – synthesis for Xilinx FPGAs

\begin{verbatim}
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.

-\texttt{-top <module>}
  use the specified module as top module

-\texttt{-edif <file>}
  write the design to the specified edif file. Writing of an output file is omitted if this parameter is not specified.

-\texttt{-blif <file>}
  write the design to the specified BLIF file. Writing of an output file is omitted if this parameter is not specified.

-\texttt{-vpr}
  generate an output netlist (and BLIF file) suitable for VPR (this feature is experimental and incomplete)

-\texttt{-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.

-\texttt{-flatten}
  flatten design before synthesis

-\texttt{-retime}
  run 'abc' with -dff option

The following commands are executed by this synthesis command:

-\texttt{begin:}
  -\texttt{read_verilog -lib +/xilinx/cells_sim.v}
  -\texttt{read_verilog -lib +/xilinx/cells_xtra.v}
  -\texttt{read_verilog -lib +/xilinx/brams_bb.v}
  -\texttt{hierarchy -check -top <top>}
\end{verbatim}
flatten: (only if -flatten)
   proc
   flatten

coarse:
   synth -run coarse

bram:
   memory_bram -rules +/xilinx/brams.txt
   techmap -map +/xilinx/brams_map.v

dram:
   memory_bram -rules +/xilinx/drams.txt
   techmap -map +/xilinx/drams_map.v

fine:
   opt -fast -full
   memory_map
dffsr2dff
dff2dffe
   opt -full
   techmap -map +/techmap.v -map +/xilinx/arith_map.v
   opt -fast

map_luts:
   abc -luts 2:2,3,6:5,10,20 [-dff]
clean

map_cells:
   techmap -map +/xilinx/cells_map.v (with -D NO_LUT in vpr mode)
dffinit -ff FDRE Q INIT -ff FDCE Q INIT -ff FDPE Q INIT
clean

check:
   hierarchy -check
   stat
   check -noinit

edif: (only if -edif)
   write_edif <file-name>

blif: (only if -blif)
   write_blif <file-name>

---

C.144 tcl – execute a TCL script file

tcl <filename>

This command executes the tcl commands in the specified file.
Use 'yosys cmd' to run the yosys command 'cmd' from tcl.
The tcl command 'yosys -import' can be used to import all yosys commands directly as tcl commands to the tcl shell. 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.

C.145 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 ilang source file.

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

- **-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.

- **-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.

- **-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.

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.

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:

_INDICATEFAIL

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
_INDICATE_* commands have been executed, an error is generated.

_INDICATE_*

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; '.

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.
APPENDIX C. COMMAND REFERENCE MANUAL

_**TECHMAP_CONSTMSK_<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_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.

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.146 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.147 test_abcloop – automatically test handling of loops in abc command**
APPENDIX C. COMMAND REFERENCE MANUAL

C.148 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.

```
-n <int>
number of iterations the test bench should run (default = 1000)
```

C.149 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).
```
C.150  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.151 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.152 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.153 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.154 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 -Dmacro[=<value>] options may be added after the option indicating
the language version (and before file names) to set additional verilog defines.
The macros SYNTHESE and VERIFIC are defined implicitly.

```
verific -formal <verilog-file>..
```

Like -sv, but define FORMAL instead of SYNTHESE.

```
verific {-vhdl87|-vhdl93|-vhdl2k|-vhdl2008|-vhdl} <vhdl-file>..
```

Load the specified VHDL files into Verific.

```
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 -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

-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.

Visit http://verific.com/ for more information on Verific.
C.155  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.156  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'
```

C.157  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.
```
C.158  write_aiger – write design to AIGER file

```plaintext
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
```

C.159  write_blif – write design to BLIF file

```plaintext
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.160  write_btor – write design to BTOR file

write_btor [options] [filename]

Write a BTOR description of the current design.

-v
APPENDIX C. COMMAND REFERENCE MANUAL

Add comments and indentation to BTOR output file

Output only a single bad property for all asserts

C.161 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)

-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.162 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.163 write_firrtl – write design to a FIRRTL file
APPENDIX C. COMMAND REFERENCE MANUAL

write_firrtl [options] [filename]
Write a FIRRTL netlist of the current design.

C.164 write_ilang – write design to ilang file

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

-selected
  only write selected parts of the design.

C.165 write_intersynth – write design to InterSynth netlist file

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

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

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

-selected
  only write selected modules. modules must be selected entirely or not at all.

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

C.166 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
The general syntax of the JSON output created by this command is as follows:

```
{
  "modules": {
    <module_name>: {
      "ports": {
        <port_name>: <port_details>,
        ...
      },
      "cells": {
        <cell_name>: <cell_details>,
        ...
      },
      "netnames": {
        <net_name>: <net_details>,
        ...
      }
    },
  },
  "models": {
    ...
  }
}
```

Where `<port_details>` is:

```
{
  "direction": <"input" | "output" | "inout">,
  "bits": <bit_vector>
}
```

And `<cell_details>` is:

```
{
  "hide_name": <1 | 0>,
  "type": <cell_type>,
  "parameters": {
    <parameter_name>: <parameter_value>,
    ...
  },
  "attributes": {
    <attribute_name>: <attribute_value>,
    ...
  },
  "port_directions": {
    <port_name>: <"input" | "output" | "inout">,
    ...
  },
  "connections": {
    <port_name>: <bit_vector>,
    ...
  }
}
```
And `<net_details>` is:

```json
{
    "hide_name": <1 | 0>,
    "bits": <bit_vector>
}
```

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" or "1" instead of a number.

Numeric parameter and attribute values up to 32 bits are written as decimal values. Numbers larger than that are written as string holding the binary representation of the value.

For example the following Verilog code:

```verilog
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:

```json
{
    "modules": {
        "test": {
            "ports": {
                "x": {
                    "direction": "input",
                    "bits": [ 2 ]
                },
                "y": {
                    "direction": "input",
                    "bits": [ 3 ]
                }
            },
            "cells": {
                "foo_inst": {
                    "hide_name": 0,
                    "type": "foo",
                    "parameters": {
                        "Q": 1337,
                        "P": 42
                    },
                    "attributes": {
```
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> ]
```

"keep": 1,
"src": "test.v:2"
},
"connections": {
    "C": [ 2, 2, 2, 2, "0", "1", "0", "1" ],
    "B": [ 2, 3 ],
    "A": [ 3, 2 ]
}
},

"netnames": {
    "y": {
        "hide_name": 0,
        "bits": [ 3 ],
        "attributes": {
            "src": "test.v:1"
        }
    },
    "x": {
        "hide_name": 0,
        "bits": [ 2 ],
        "attributes": {
            "src": "test.v:1"
        }
    }
}
```

The models are given as And-Inverter-Graphs (AIGs) in the following form:
- 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.167 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.168 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>
Each $assert cell is converted into one of this functions. The function evaluates to 'true' if the assert statement holds in the state.

Each $assume cell is converted into one of this functions. The function evaluates to 'true' if the assume statement holds in the state.

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.

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.

```verilog
define 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).

```verilog
; 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:

```bash
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.169 write_smv – write design to SMV file

```bash
write_smv [options] [filename]
```

Write an SMV description of the current design.

- `-verbose`
  - this will print the recursive walk used to export the modules.
C.170 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)

-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.171 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.172  write_verilog – write design to Verilog file

```
write_verilog [options] [filename]

Write the current design to a Verilog file.

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

- 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.

- 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.

- 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.173  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

Application Notes

This appendix contains copies of the Yosys application notes.

- Yosys AppNote 010: Converting Verilog to BLIF .........................Page 177
- Yosys AppNote 011: Interactive Design Investigation ..................Page 180
- Yosys AppNote 012: Converting Verilog to BTOR ......................Page 189
Yosys Application Note 010:
Converting Verilog to BLIF
Clifford 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. The Berkeley Logic Interchange Format (BLIF) [6] is a simple file format for exchanging sequential logic between programs. It is easy to generate and easy to parse and is therefore the preferred method of design entry for many authors of logic synthesis tools.

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

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

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

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

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

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

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

1 $ yosys -o softusb_navre.blif -S softusb_navre.v

Listing 1. Calling Yosys without script file

Behind the scenes Yosys is controlled by synthesis scripts that execute commands that operate on Yosys’ internal state. For example, the -o softusb_navre.blif option just adds the command write_blif softusb_navre.blif to the end of the script. Likewise a file on the command line – softusb_navre.v in this case – adds the command read_verilog softusb_navre.v to the beginning of the synthesis script. In both cases the file type is detected from the file extension.

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 parameterized 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 -nonap is used, in which case the multi-port block memories stay in the design and can then be mapped to architecture-specific memory primitives using other commands.

• The command techmap turns a high-level circuit with coarse grain cells such as wide adders and multipliers to a fine-grain circuit of simple logic primitives and single-bit storage elements. The command does that by substituting the complex cells by circuits of simpler cells. It is possible to provide a custom set of rules for this process in the form of a Verilog source file, as we will see in the next section.

Now Yosys can be run with the filename of the synthesis script as argument:

1 $ yosys softusb_navre.ys

Listing 3. Calling Yosys with script file

Now that we are using a synthesis script we can easily modify how Yosys synthesizes the design. The first thing we should customize is the call to the hierarchy command:

177
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:

```verilog
module adff2dff (CLK, ARST, D, Q);
parameter WIDTH = 1;
parameter ARST_VALUE = 0;
parameter ARST_POLARITY = 1;
parameter CLK_POLARITY = 1;
parameter ARST_VALUE = 0;
input CLK, ARST;
input [WIDTH-1:0] D;
output reg [WIDTH-1:0] Q;
wire [1023:0] _TECHMAP_DO_ = "proc";
wire _TECHMAP_FAIL_ = !CLK_POLARITY || !ARST_POLARITY;
always @posedge CLK)
  if (ARST)
    Q <= ARST_VALUE;
  else
    Q <= D;
endmodule
```

Listing 7. Implicit coding of global asynchronous resets

(Not 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-bigism, was verified using the test-bench from yosys-bigism. It successfully executed the program shown in Listing 8 in the test-bench.
#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;
}

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

I. INSTALLATION AND PREREQUISITES

This Application Note is based on the Yosys [1] GIT Rev. 2b90a1 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 so forth. When called without any special options, it saves the circuit diagram in a temporary file and launches xdot.

Ordinary 32-bit constants are written using decimal numbers. Constants are shown as ellipses with the constant value as label. 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 thin 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 $mux cell but the multiplexer from the if-statement is yet still hidden within the process.

The syntax block in the 2nd line.

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 $bit is used 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.

Figure 1. Yosys script with show commands and example design

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-flop, 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 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.

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 to 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 sincells.v from the Yosys data directory (as printed by yosys-config --datadir).

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 be 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

```
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

the part after the last dollar-sign. Most auto-generated names (the ones starting with a dollar sign) are rather long and contains some additional information on the origin of the named object. But in most cases those names can simply be abbreviated using the last part.

Usually all interactive work is done with one module selected using the cd command. But it is also possible to work from the design-context (cd ..). In this case all object names must be prefixed with <module_name>/. For example a*/b* would refer to all objects whose names start with b from all modules whose names start with a.

The dump command can be used to print all information about an object. For example dump $2 will print Fig. 7. This can for example be useful to determine the names of nets connected to cells, as the net-names are usually suppressed in the circuit diagram if they are auto-generated.

For the remainder of this document we will assume that the commands are run from module-context and not design-context.

B. Working with selections

When a module is selected using the cd command, all commands (with a few exceptions, such as the read_* and write_* commands) operate only on the selected module. This can also be useful for

```
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
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 and 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 disjoint 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.
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 command 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:

```plaintext
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 $%ci^3$ 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.

Let's 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:

```plaintext
show y $%ci^2$:
```

To add a pattern we add a colon followed by the pattern to the $%ci$ action. The pattern itself 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 $\text{dff}$ cell, we can write:

```plaintext
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.
cell, we could as well only specify the port names:

```
show y %ci2:+[Q,D]
```

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 %ci to the command:

```
show y %ci2:=[Q,D] %ci+:=$mux[S]:=$dff
```

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:=[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 % action mentioned previously also accepts this advanced syntax.

This actions for traversing the circuit graph, combined with the actions for boolean operations such as intersection (%i) and difference (%d) are powerful tools for extracting the relevant portions of the circuit under investigation.

See help select for a complete list of actions available in selections.

E. Storing and recalling selections

The current selection can be stored in memory with the command:

```
select -set <name>
```

It can later be recalled using select @<name>. In fact, the @<name> expression pushes the stored selection on the stack maintained by the select command. So for example:

```
select @foo @bar %i
```

will select the intersection between the stored selections foo and bar.

In larger investigation efforts it is highly recommended to maintain a script that sets up relevant selections, so they can easily be recalled, for example when Yosys needs to be re-run after a design or source code change.

The history command can be used to list all recent interactive commands. This feature can be useful for creating such a script from the commands used in an interactive session.

V. ADVANCED INVESTIGATION TECHNIQUES

When working with very large modules, it is often not enough to just select the interesting part of the module. Instead it can be useful to extract the interesting part of the circuit into a separate module. This can for example be useful if one wants to run a series of synthesis commands on the critical part of the module and wants to carefully read all the debug output created by the commands in order to spot a problem. This kind of troubleshooting is much easier if the circuit under investigation is encapsulated in a separate module.

Fig. 17 shows how the submod command can be used to split the circuit from Fig. 14 and 15 into its components. The -name option is used to specify the name of the new module and also the name of the new cell in the current module.

A. Evaluation of combinatorial circuits

The eval command can be used to evaluate combinatorial circuits. For example (see Fig. 17 for the circuit diagram of selstage):

```
yosys [selstage]> eval -set s2,s1 4'b1001 -set d 4'hc -show s2 -show n1
```
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]

<table>
<thead>
<tr>
<th>$s1 &amp; [0]</th>
<th>$n2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2'00 1'0</td>
<td>2'00 1'0</td>
</tr>
<tr>
<td>2'00 1'0</td>
<td>2'00 2'00</td>
</tr>
</tbody>
</table>

Assumed undef (x) value for the following signals: \$n2

Note that the eval command (as well as the sat command discussed in the next sections) does only operate on flattened modules. It cannot 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 1 -set n2,n1 4'b1001
```

II. Executing SAT pass (solving SAT problems in the circuit).
   Full command line: sat -show s1,s2,d -set s1 1 -set n2,n1 4'b1001

Setting up SAT problem:
   Import set-constraint: \s1 = \s2
   Final constraint equation: \{ \n2 \& \n1 \} = \{ 4'1001 \& \s2 \}

Solving problem with 81 variables and 207 clauses...
SAT solving finished - model found:

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Dec</th>
<th>Hex</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>\n1</td>
<td>0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\n2</td>
<td>0 0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note that the sat command supports signal names in both arguments to the -set option. In the above example we used -set s1 1 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.)

Figure 18. A simple miter circuit for testing if a number is prime. But it has a problem (see main text and Fig. 19).
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 $p \neq a*b$ in the miter with $p \neq {16’d}0,a*b$, or by using a temporary variable for the 32 bit product a*b. But as 31 fits well into 8 bits (and as the purpose of this document is to show off Yosys 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

---

**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.
satisfy the other constraints.

It might be worth noting, that SAT solvers are not particularly efficient at factorizing large numbers. But if a small factorization problem occurs as part of a larger circuit problem, the Yosys SAT solver is perfectly capable of solving it.

C. Solving sequential SAT problems

The SAT solver functionality in Yosys can not only be used to solve combinatorial problems, but can also solve sequential problems. Let’s consider the entire mndemo 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 running the following command:

```bash
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

yosys/

188
Yosys Application Note 012:
Converting Verilog to BTOR
Ahmed Irfan and Clifford 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) begin
            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 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

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.
Listing 5 shows the Yosys commands that are executed by verilog2btor.sh.

```yaml
read_verilog -sv $1;
hierarchy -top $3; hierarchy -libdir $DIR;
hierarchy -check;
proc; opt;
opt_expr -mux_undef; opt;
rename -hide;;;
splice; opt;
memory_dff -wr_only; memory_collect;;
flatten;;
memory_unpack;
splitnets -driver;
setundef -zero -undriven;
opt;;;
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:

```yaml
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

Here is an example Verilog design that we want to convert to BTOR:

```verilog
module array(input clk);
    reg [7:0] counter;
    reg [7:0] mem [7:0];
    always @(
       posedge clk)
    begin
        counter <= counter + 8'd1;
        mem[counter] <= counter;
    end
    assert property (!(counter > 8'd0) ||
        mem[counter - 8'd1] == counter - 8'd1);
endmodule
```

Listing 7. Example - Verilog Design

The generated BTOR file that contain memories, using the script shown in Listing 5:

```text
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:
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

Bibliography


Internet References