University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Masters Theses | Dissertations and Theses

October 2022

Formally Verifiable Synthesis Flow In FPGAs

Anurag V. Muttur
University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/masters_theses_2

 Part of the VLSI and Circuits, Embedded and Hardware Systems Commons

Recommended Citation

Muttur, Anurag V., "Formally Verifiable Synthesis Flow In FPGAs" (2022). *Masters Theses*. 1237.
https://doi.org/10.7275/31043432 https://scholarworks.umass.edu/masters_theses_2/1237

This Open Access Thesis is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Masters Theses by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

FORMALLY VERIFIABLE SYNTHESIS FLOW IN FPGAs

A Thesis Presented

by

ANURAG MUTTUR

Submitted to the Graduate School of the

University of Massachusetts Amherst in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2022

© Copyright by Anurag Muttur 2022
All Rights Reserved

FORMALLY VERIFIABLE SYNTHESIS FLOW IN FPGAs

A Thesis Presented

by

ANURAG MUTTUR

Approved as to style and content by:

_____

Russell Tessier, Chair

_____

Wayne Burleson, Member

_____

Neal Anderson, Member

_____

Christopher V. Hollot, Department Head

Electrical and Computer Engineering

ACKNOWLEDGMENTS

I want to express my gratitude to my advisor Professor Russell Tessier, for his support, guidance, and mentorship throughout my thesis work. It was a great honor and privilege to work under his guidance. I am extremely grateful for what he has offered me.

I would also like to thank my colleagues and good friends Andrew Hartnett, Tien Li Shen, Dhruv Kansagara, and Shayan Moini for guiding me throughout my work. Their endless support throughout the research has been immense and I am thankful for that.

Finally, I would like to express my profound gratitude to my parents and brother for encouraging me along the way.

ABSTRACT

FORMALLY VERIFIABLE SYNTHESIS FLOW IN FPGAs

SEPTEMBER 2022

ANURAG MUTTUR
B.Tech., PES UNIVERSITY, BANGALORE
M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell Tessier

FPGAs are used in a wide variety of digital systems. Due to their ability to support parallelism and specialization, these devices are becoming more commonplace in fields such as machine learning. One of the biggest benefits of FPGAs, logic specialization, can lead to security risks. Prior research has shown that a large variety of malicious circuits can snoop on sensitive user data, induce circuit faults, or physically damage the FPGA. These Trojan circuits can easily be crafted and embedded in FPGA designs. Often, these Trojans are small, consume little power in comparison to the target circuit, and are hard to detect via simulation or physical inspection.

Computer-aided design (CAD) software in FPGAs has been the subject of extensive research and development of FPGAs for the past thirty-five years. The current FPGA software landscape includes vendors that provide widely used software flows to convert behavioral and register-transfer level (RTL) descriptions to bitstreams needed to program an FPGA device. Given the complexity of the algorithms needed to perform this translation, these CAD tool flows are generally structured as black boxes with limited transparency regarding design conversion steps or the logical equivalence of the generated design and initial design specification.

This work explores the enhancement of open-source FPGA software, SymbiFlow, that focuses on FPGA RTL synthesis, place and route and bitstream generation. SymbiFlow uses Yosys for synthesis, VPR for place and route, and Project X-Ray for bitstream generation. We focus on synthesis using Yosys and formal verification using the Cadence Conformal Logic Equivalence Checker (LEC) for Xilinx Artix-7 FPGAs. Yosys is used to synthesize 160 benchmarks written in Verilog. We implement required code modifications to Yosys for designs to pass the equivalence checker. For Conformal, this work involves processing 160 benchmark designs with the equivalence checker. Parameters can be toggled on or off to obtain results that indicates if a design has passed formal verification when comparing RTL and synthesized netlists.

TABLE OF CONTENTS

ACKNOWLEDGMENTS ..... **iv**

ABSTRACT ..... **v**

CHAPTER 1 ..... **1**

INTRODUCTION **1**

1.1 Yosys 2

1.2 Conformal Equivalence Checker (LEC) 2

1.3 Xilinx Vivado 3

1.4 The Setup 3

1.5 Thesis Outline 4

CHAPTER 2 ..... **5**

SYNTHESIS WITH YOSYS **5**

2.1 Internal Yosys Operation 5

2.2 Synthesis Flow 6

2.3 Source Code Changes in Yosys 9

2.3.1 Turning off Optimizations 9

2.3.2 Addition of Hint Files for FSM recoding 11

2.3.3 Hint Files for Flip Flop Merging 12

2.3.4 Renaming Registers 12

2.3.5 ABC Script File 12

CHAPTER 3 ..... **14**

FORMAL VERIFICATION USING CADENCE CONFORMAL LEC **14**

3.1 Conformal System Modes 15

3.2 Key Points and Mapping 16

3.3 Comparing and Diagnosing Key Points 16

3.4 Commands Used by Conformal 17

CHAPTER 4 ..... **20**

BENCHMARKS USED IN SYNTHESIS AND FORMAL VERIFICATION 20

4.1 Work in Design Verification For 8 Additional Designs 20

CHAPTER 5 26

FORMAL VERIFICATION IN YOSYS 26

5.1 An Open-Source Formal Verification Tool 26

5.2 Verifying Designs in Yosys-SMTBMC 27

5.2.1 Files used by Yosys-SMTBMC 28

5.3 Verifying Designs Using the Yosys Equivalence Checker 33

CHAPTER 6 36

BUGS AND BUG FIXES IN YOSYS 36

6.1 Bug Fixes in Yosys 36

6.1.1 Preset-Clear Swapping Bug 36

6.1.2 Synthesis Bug in boundtop 40

6.2 Modifying the VPR Library 41

6.2.1 Syntax Errors in VPR Library 43

7.2.2 Addition of Cells to VPR Library 44

7.2.3 Formally Verifiable Block RAMs 46

CHAPTER 7 51

RESULTS 51

CHAPTER 8 55

CONCLUSION 55

9.1 Future Work 55

9.1.1 Additional Benchmarks 55

9.1.2 Protection of Hints Files 56

9.1.3 Robust Yosys Equivalence Checker 56

APPENDIX 57

BIBLIOGRAPHY 81

LIST OF TABLES

Table **Page**

2. 1 Yosys Commands, Passes and Their Functions 7

2. 2. Optimizations Toggled in Yosys 10

3. 1. Commands in Conformal 17

4. 1. Additional designs tested 21

5. 1. Design tested in Yosys-SMTBMC 27

5. 2. Yosys Equivalence Check commands 33

5. 3. Yosys Equivalence Checker Statistics 35

7. 1. Logic Design Area Comparison (LUTs and Flip Flops) 51

7. 2. Logic Design Area Comparison (BRAMs and DSP Blocks) 53

7. 3. Design Performance Comparison 53

7. 4. Arithmetic and Geometric Means of Designs Synthesized in Yosys and Vivado 54

A. 1. All Verilog Benchmarks 57

A. 2. Initial 160 Designs Tested 63

A. 3. Benchmarks Used In Results Generation 69

A. 4. Benchmarks That Pass Equivalence Checking Through Yosys in SymbiFlow 71

A. 5. Designs that generate bitstreams in SymbiFlow 75

LIST OF FIGURES

Figure **Page**

1. 1. The Formal Verification Environment 4
2. 1. Data Flow of Yosys 6
3. 1. Conformal LEC Flow 15
3. 2. Design Fails Conformal 17
3. 3. Design Passes Conformal 17
5. 1. The fiedler-cooley design 29
5. 2. Condition to check reachability of count_out 29
5. 3. The fc.sby file used by fiedler-cooley 30
5. 4. The fielder-cooley design passes SymbiYosys analysis 31
5. 5. Visualization of BMC for fiedler-cooley 32
5. 6. Bash script used to execute the Yosys equivalence checker 33
6. 1. Design dffsr2_sub synthesized by Yosys 37
6. 2. Design dffsr2_sub synthesized by Conformal 37
6. 3. dffsr2_sub module in Verilog 37
6. 4. Hierarchy of the *if* and *else* statements 38
6. 5. *Many_async_rules_vector* created to keep track of the hierarchy 38
6. 6. Before bug fix in *proc_dff.cc* 39
6. 7. After bug fix in *proc_dff.cc* 39
6. 8. Synthesized dffsr2_sub with the bug 40
6. 9. Synthesized dffsr2_sub without the bug after bug fix 40
6. 10. Outputs of two flip flops drive the same wire 41
6. 11. Added default statement to Boundtop 41
6. 12. FPGA flow from RTL to bitstream generation in SymbiFlow 42
6. 13. Before the Verilog syntax change 44
6. 14. After the Verilog syntax change 44
6. 15. LDCE added to *cells_sim.v* 45

6. 16. LDCE added to *cells_map.v* 46
6. 17. RAMB18E1 Xilinx primitiveT 47
6. 18. BRAM36 module 48
6. 19. clkardclk wire depends on parameter values 49
6. 20. Direct connection of clkardclk wire to CLKARDCLK port 50

CHAPTER 1

INTRODUCTION

The verification of digital hardware correctness is becoming increasingly important due to the growing use of custom accelerators using FPGAs. A register transfer level (RTL) hardware design is generally written in a hardware description language (HDL). An HDL, such as Verilog or VHDL, is used to describe the structure and behavior of a circuit. A synthesis tool is a program that takes the specified HDL file and converts it into a netlist that implements the circuit. Several circuit synthesis tools are available, but most of them are commercial tools, such as Cadence Genus [30] and Mentor Precision [31]. These proprietary tools limit the transparency of design conversion steps.

Yosys [1][2][21] is a free and open-source synthesis tool that uses ABC [10] as a backend for optimizations and technology mapping. The open-source tool allows the user to make code modifications for extra functionality. This work aims to improve the quality of synthesis tools and verify the correctness of the design through formal verification. The synthesis of an RTL design constructs a gate-level netlist by mapping it to a technology library. Conformal LEC [3], a tool developed by Cadence Design Systems [4][5], is a formal verification tool that can check whether a design written in RTL (the golden netlist) and a synthesized circuit (the revised netlist) have equivalent functionality. For this work, 160 benchmarks have been synthesized by Yosys and used by Cadence Conformal to perform formal verification. These designs of varying complexity are from the Yosys-simple [14], Yosys-bigsim [15], and VTR-benchmarks [16] benchmark suites. The designs vary from a simple full-adder to complex designs such as microprocessors and circuits used in ray tracing. The main goal for our work is to verify that synthesized designs pass equivalence checking [20]. A failed verification can lead to Trojan detection [48].

Rathmair et al. [48] apply formal methods to circuits to detect hardware Trojans. Equivalence checking, property checking, and reachability analysis are used to detect Trojans. Designs are converted into Reduced Ordinary Binary Decision Diagrams (ROBDDs) in a canonical form, a unique representation of digital logic. During

equivalence checking, the functions are compared at different abstraction levels. Negative equivalence checking indicates that the abstraction level has changed by the insertion of malicious hardware in the design.

Previous synthesis work on RTL designs [17] examined Yosys, Xilinx Vivado [7][8], and Intel Quartus Prime [38]. An open-source tool named Verismith [32] generates pseudo-random, valid, deterministic Verilog designs and feeds each design to a synthesis tool. For formal verification, Verismith uses an SMT solver or the ABC circuit verification tool to check that the output is logically equivalent to the input. Bugs were found in multiple synthesis tools, except Quartus Prime [38]. Our work focuses on Yosys and performs formal verification using a commercial equivalence checker. The designs tested in this thesis were created for real-world deployment and were not randomly generated.

Shah et al. [19] showed how Yosys can be used for design synthesis and nextpnr can be used for place-and-route. This flow targets iCE40 and ECP5 FPGAs from Lattice [33]. In contrast, our work targets Xilinx Artix-7 FPGAs [34]. Our modified Yosys synthesis tool does not aim to be competitive with commercial synthesis flows in terms of area or performance. Our focus is formal verification of synthesized designs.

1.1 Yosys

Yosys is a free and open-source tool that performs logic synthesis. Commands in Yosys are used to perform RTL and logic synthesis. In this work, Yosys generates a synthesized structural netlist that has Verilog syntax and contains FPGA cells. These designs can be mapped to a commercial FPGA board [7][8] including an Artix-7 series FPGA [34]. Logic cells include look up tables (LUTs), flip flops, block RAMs, and digital signal processing (DSP) blocks [12].

1.2 Conformal Equivalence Checker (LEC)

Simulation can be used to verify a synthesized design's functionality. However, this solution may not be feasible for large designs as it requires sizable sets of input vectors to

be used with the design, and simulation may be time-consuming [18]. A potentially faster and more thorough verification approach is formal verification [22][23]. This approach establishes the functional equivalence of two designs generally represented as models without performing simulation. Cadence Conformal LEC is a logic equivalence tool that verifies if the RTL design is equivalent to the synthesized design. A '.dofile' command script contains the parameters required to direct Cadence Conformal LEC operation.

1.3 Xilinx Vivado

Vivado is a comprehensive FPGA design suite that includes synthesis, place-and-route, and bitstream generation. Vivado can read a post-synthesis netlist (e.g., one created by Yosys) for subsequent physical design. The tool includes a simulator that can be used to verify an RTL or post-synthesis netlist [8]. In this thesis, we contrast results generated with Vivado for an Artix-7 FPGA against those generated by Yosys. Designs from both flows are mapped to an Arty-35 board [11].

1.4 The Setup

Figure 1.1 demonstrates the setup of the formal verification environment used in this work. Synthesis quality can be evaluated in lookup table (LUT) and flip flop count, number of block RAMs (BRAMs) and number of DSP blocks. Vivado performs place-and-route without optimizations using the Yosys-synthesized netlist. Post place-and-route performance and area can be determined by iteratively changing the clock period for each design until no timing violations are detected. During formal verification, the equivalence checker (e.g., LEC) should produce a 'PASS'. Incomplete or non-equivalent results can be considered a failure [5]. The input library, xeclib [27], is used to support verification.

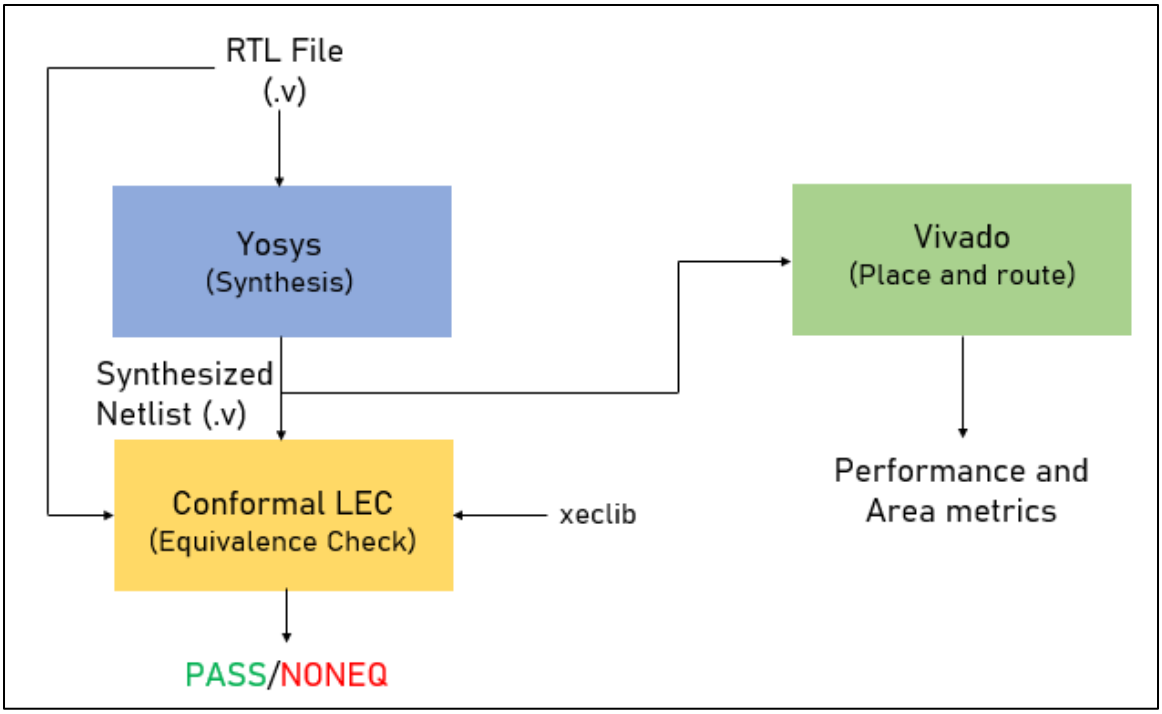

Figure 1. 1. The Formal Verification Environment

1.5 Thesis Outline

This thesis document is divided into the following chapters. Chapter 2 discusses how Yosys is used and includes changes made to the tool to allow optimization shut off and the generation of equivalence checking hints. This chapter also discusses Vivado place and route usage. Chapter 3 discusses Cadence Conformal LEC, and the various switches supported by the tool. Chapter 4 describes the 160 benchmarks used, of which 144 passed, and details why some designs may not have passed. Chapter 5 describes the results obtained when these designs are synthesized with Yosys and placed-and-routed by Vivado. Chapter 7 discusses formal verification with Yosys-SMTBMC [43] and the Yosys equivalence checker [2]. Chapter 8 discusses bug fixes made in Yosys and support for SymbiFlow. Chapter 9 concludes the thesis document and offers directions for future work.

CHAPTER 2

SYNTHESIS WITH YOSYS

Logic synthesis converts a design specified in an HDL, such as Verilog or VHDL, into a gate-level netlist. Optimizations can be used by the synthesis tool to simplify a netlist. Our benchmarks are synthesized by Yosys using a synthesis library provided by Xilinx referred to as xeclib [27]. Our work uses a modified copy of Yosys version 0.9+4249 and 160 RTL benchmarks. Commands are used to read the design written in RTL, synthesize and technology map the design to the Xilinx library, and write out the synthesized netlist. This chapter provides information on how Yosys works and our modifications.

2.1 Internal Yosys Operation

Yosys synthesis steps [24] include the use of a lexer and a parser. The input design is converted into an Abstract Syntax Tree (AST) [25] and then into Register-to-Transfer Level Intermediate Language (RTLIL) format. The lexer parses the input Verilog files, identifies Verilog keywords and tokenizes them. These tokens are used as nodes in an AST. The Verilog parser then generates an AST data structure using the information about nodes provided by the lexer. The data structure is then passed directly to the AST frontend where it is simplified and converted into an RTLIL netlist - an internal format used by Yosys. This conversion from AST to RTLIL can be done in two steps: simplification and RTLIL generation. During simplification, keywords present in the AST data structure are converted to simpler tokens. Once simplified, RTLIL generation is performed by a recursive process that generates equivalent RTLIL data for the AST data. Finally, the RTLIL representation is technology mapped using a library. The synthesized netlist is then written in the file format specified by the user.

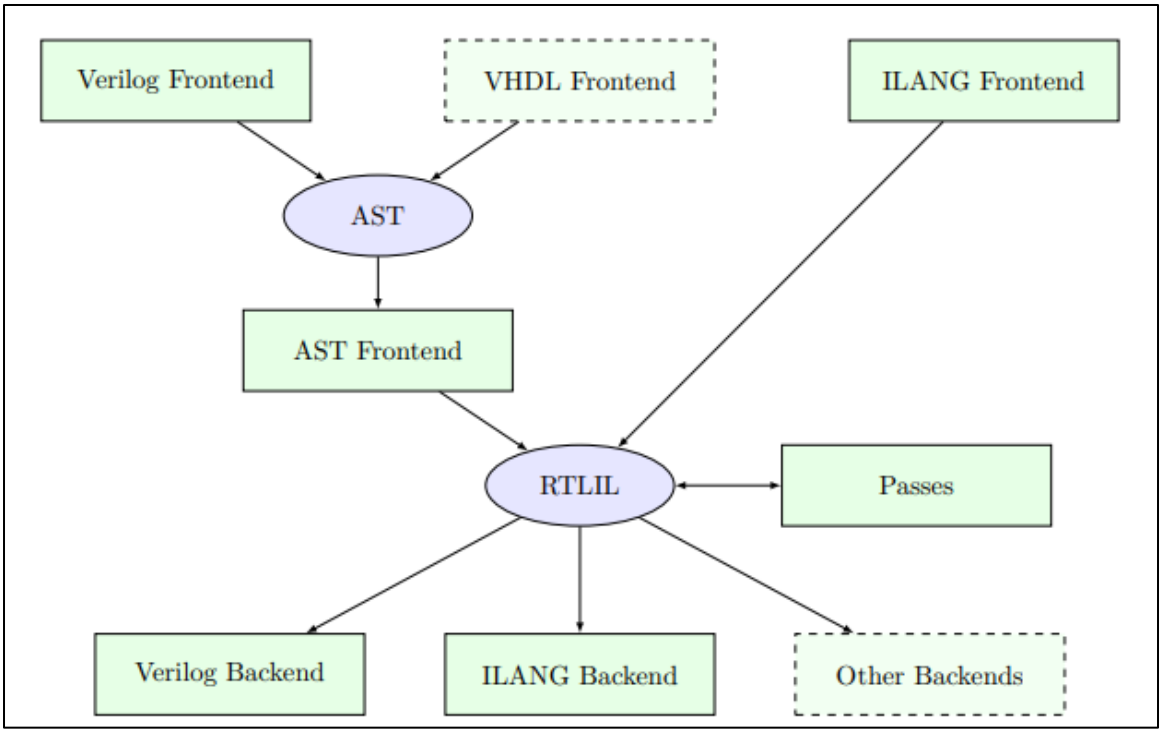

Figure 2. 1. Data Flow of Yosys [2]

Figure 2.1 shows the simplified data flow in Yosys. The rectangles in the diagram are program modules and the ellipses are data structures generated by the program modules. Optimizations and technology mapping are examples of passes used in the Passes module. The ILANG representation shown in Figure 2.1 is a text representation of the RTLIL format that can directly be converted into RTLIL without conversion to AST. The 'other backends' module includes netlist output into different formats, such as edif or blif. The conversion of RTLIL to Verilog is performed when the netlist is written out.

2.2 Synthesis Flow

Our Yosys synthesis flow includes the following commands:

1. read_verilog <design_file.v>

   This command parses and reads the Verilog file. The lexer and parser are used to read the input RTL file provided and converts it into an AST representation.
2. synth_xilinx

This command synthesizes the design and performs technology mapping to the Xilinx library input into Yosys. Synthesis uses the steps noted in Table 2.1.

Table 2. 1 Yosys Commands, Passes and Their Functions

	Command under 'synth_xilinx'	Sub-commands/Passes	Function
1	begin	• read_verilog -lib +/xilinx/cells_sim.v • read_verilog -lib +/xilinx/cells_xtra.v • read_verilog -lib +/xilinx/brams_bb.v • read_verilog -lib +/xilinx/drams_bb.v • hierarchy -check -top <top>	Read the Xilinx libraries in Yosys, checks the hierarchy of the design and sets the top module accordingly
2	synth -run coarse	• proc • opt_expr • opt_clean • check • opt • wreduce • alumacc • share • opt • fsm • opt -fast • memory -nomap • opt_clean	Run coarse optimizations
3	bram	• memory_bram -rules +/xilinx/brams.txt • techmap -map +/xilinx/brams_map.v	Technology mapping for block RAMs.

4	dram	• memory_bram -rules +/xilinx/drams.txt • techmap -map +/xilinx/drams_map.v	Technology mapping for memory (DRAMs).
5	fine	• opt -fast -full • memory_map • dffsr2dff • dff2dffe • opt -full • techmap -map +/techmap.v -map +/xilinx/arith_map.v • opt -fast	Perform fine grain optimizations.
6	map_luts	• abc -luts 2:2,3,6:5,10,20 [-dff] • clean	Map LUTs to design.
7	map_cells	• techmap -map +/xilinx/cells_map.v • dffinit -ff FDRE Q INIT -ff FDCE Q INIT -ff FDPE Q INIT • clean	Map cells to design.
8	check	• hierarchy -check • stat • check -noinit	Check and display hierarchy of the design.

The commands shown in Table 2.1 are used on the design RTLIL representation.

3. write_verilog <design_file_synth.v>

   The synthesized netlist is written to the current Yosys execution directory unless the directory of the synthesized file is specified. The synthesized file is usually appended with a ‘synth’ suffix at the end of the file name, before the ‘.v’ extension.

These steps can be automated by creating a text file with the commands and making a single call to Yosys to call the script. This call will run all steps in one command. For example, if the commands are stored in a text file named *yosys_run.txt*, this file can be executed by calling Yosys along with the switch *-s* that makes use of the specified script. The command used would be *yosys -s yosys_run.txt* to run all the steps.

2.3 Source Code Changes in Yosys

Initially, over 100 of the 160 designs synthesized by Yosys for a Xilinx Artix-7 FPGA did not pass LEC equivalence checking with default Yosys optimizations. In some cases, the optimizations obscured the design function (e.g., removed registers, renamed registers, recoded state machines) making it impossible for the verification tool to work properly. Initially, to allow designs with these optimizations to pass equivalence checking, the optimizations had to be turned off. Yosys provide some switches to disable optimizations, such as *-nobram* which forbids the use of block RAMs [36]. However, there were optimizations (e.g. register removal) that could not be deactivated using standard Yosys switches. Yosys source code modifications allowed us to add command line switches and functionality to suppress these optimizations. These additions are noted below.

2.3.1 Turning off Optimizations

The switches used to suppress optimizations are detailed in Table 2.2. These switches are used during the *synth_xilinx* step.

Table 2. 2. Optimizations Toggled in Yosys

Sl. No.	Switch that turns off the optimization	Function
1	-nofsm	Do not extract and optimize finite state machines in input code
2	-nowreduce	Do not attempt word size reduction of arithmetic operations. For example, if both input values are 32 bits but only an 8-bit output is needed, the adder size can be reduced to 8 bits.
3	-noopt	Prevent simple logic optimizations that may result in the elimination of flip flops.
4	-noopt_dff	Do not perform DFF optimization. This includes the removal of D flip flops that are driven by constant values and the merging of flip flops that are logically equivalent.
5	-noopt_muxtree	Do not eliminate dead branches in multiplexer trees.
6	-noopt_reduce	Do not simplify large MUXes and AND/OR gates.
7	-noopt_merge	Do not merge logically identical cells, including flip flops. This optimization is separate from the optimizations affected by -noopt_dff
8	-noopt_clean	Do not remove unused cells and wires
9	-noopt_expr	Do not perform const folding and simple expression rewriting
10	-noopt_mem	Do not perform various optimizations on memories in the design

11	-keepff	Keep flip-flop output wires even if the wires do not affect primary output values.
12	-abc_script <file>	Use the specified ABC script file instead of the default script for logic optimization with ABC. More information on this optimization is detailed in Section 2.3.5.

2.3.2 Addition of Hint Files for FSM recoding

During Yosys execution, state machines can be automatically recoded [26]. For example, binary coded state may be recorded to a one-hot encoding (e.g., a three-bit code representing states 0….7 would be converted to an 8-bit code with only one bit active at a time). By default, Conformal expects binary state machine encoding, so Yosys recoding leads to design mismatches and formal verification failure. To address this issue, Yosys was modified to write out FSM recoding information in a "hint" file that could be used by Conformal. For example, the following FSM that uses state registers of 32-bits is converted to one-hot encoding of 5-bits. The contents of the file is shown below.

.fromstates state_reg[31] state_reg[30] state_reg[29] state_reg[28] state_reg[27] state_reg[26] state_reg[25] state_reg[24] state_reg[23] state_reg[22] state_reg[21] state_reg[20] state_reg[19] state_reg[18] state_reg[17] state_reg[16] state_reg[15] state_reg[14] state_reg[13] state_reg[12] state_reg[11] state_reg[10] state_reg[9] state_reg[8] state_reg[7] state_reg[6] state_reg[5] state_reg[4] state_reg[3] state_reg[2] state_reg[1] state_reg[0]

.tostates state_reg[4] state_reg[3] state_reg[2] state_reg[1] state_reg[0]

.begin

00000000000000000000000011001000 00010

00000000000000000000000001100100 00001

00000000000000000000000100101100 00100

00000000000000000000000011010010 01000

00000000000000000000000100110110 10000

.end

The switch *-fm_set_fsm_file fsm-file-conformal* is set during the *synth_xilinx* step in Yosys to dump out recoding information into the fsm-file-conformal file. The line *read fsm encoding fsm-file-conformal -Golden* is added to the LEC "dofile" command file so that the file produced by Yosys is read by LEC and used during formal verification.

2.3.3 Hint Files for Flip Flop Merging

During Yosys execution, design flip flops that are logically equivalent (e.g., same input signals) are identified. Redundant flip flops can be removed. If flip flops named FF1 and FF2 have the same functionality, the line *add instance equivalence FF1 FF2* is written to a hints file and FF2 is removed from the design. Yosys outputs a file *FF_renaming.log* during this optimization which lists the name of the remaining flip flop for removed flip flops. This hints file can be used by Conformal to aid in formal verification. The line *dofile FF_renaming.log* is added to the Conformal LEC "dofile".

2.3.4 Renaming Registers

The Yosys source code was modified to rename registers so that the names in the synthesized output file would be consistent with those expected by Conformal. By default, Conformal expects registers to have a '_reg' suffix to the name of the flip flop or latch output signal. For example, a DFF with an output signal *outputval* is renamed to *outputval_reg*. Output signals with an index append the suffix before the index. For example, *outputval[0]* becomes *outputval_reg[0]*. This naming convention matches the flip flop and latch naming used by LEC.

2.3.5 ABC Script File

The addition of an ABC script to perform logic optimization provides more options for synthesis improvement in Yosys. ABC performs standard optimizations by default. A command file *abc.script* that optimizes the design for binary decision diagrams (BDDs) and performs structural hashing (strash) has been added. BDD optimization identifies XOR gates for more efficient implementation of arithmetic operations (for example, wide addition). Structural hashing is performed to minimize logic across RTL logic structures

such as chained multiplexers.

CHAPTER 3

FORMAL VERIFICATION USING CADENCE CONFORMAL LEC

Formal verification uses mathematics to prove that two representations of a design possess the same behavior [28]. Equivalence checking is a type of formal verification that takes two designs that may be at the same or different levels of abstraction and determines if they are logically equivalent. Equivalence checking (e.g., Conformal LEC) can verify that a synthesized design's function is the same as an original RTL version [28]. Conformal uses combinational equivalence to determine if a design matches an original RTL or gate-level description [29] (e.g. the *golden netlist*). Verified logic includes complex arithmetic circuits, datapath circuits, memories, and custom logic. A Conformal 'PASS' result indicates that the synthesized design has passed the equivalence check [5]. Other results ('NONEQ', 'INCOMPLETE' or 'ABORT') indicate that the design did not pass the equivalence check successfully. For a design that fails, Conformal provides the number of non-equivalence points (input or output pins and-flip flops or latches). An INCOMPLETE status indicates that there are some points in the synthesized netlist that do not exist in the golden file. An ABORT occurs when some portions of the design are too complex to verify. In some cases, this issue can be overcome by setting the netlist compare effort to a higher setting [4]. Designs that do not pass the equivalence check can be assessed by observing the circuit schematic and noting differences between the RTL and synthesized netlists.

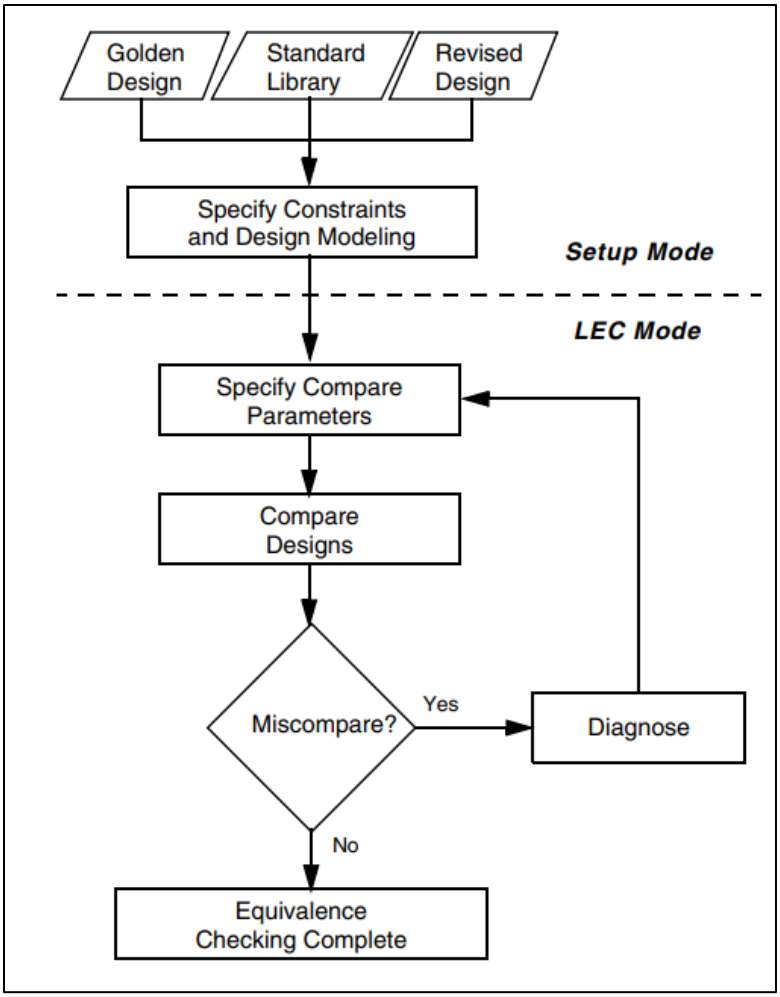

Figure 3. 1. Conformal LEC Flow [5]

3.1 Conformal System Modes

Conformal LEC operates in two modes – Setup and LEC [5]. In Setup mode, the RTL design, the synthesized design and libraries are read into Conformal LEC. The designs are set as golden and revised respectively. Generally, the revised design is the synthesized design. Constraints are set in Setup mode, such as whether the design should be flattened. Once constraints are set, Conformal switches to LEC mode. This mode uses options to perform equivalence comparison and can be used to report non-equivalent points. A variety of comparison effort settings can be used. The equivalence result is displayed at the end of LEC mode. Figure 3.1 gives an overview of the operations that occur in Setup and LEC modes.

3.2 Key Points and Mapping

In Conformal, multiple key points in the golden and revised netlists are compared. Key points include primary input and primary output pins, D flip flops, D latches, blackboxes, TIE-Z gates (high impedance signals), TIE-E gates (gates created when a don't care x-assignment exists in the revised design) and cut gates (artificial gates that break combinational loops) [5].

By default, Conformal maps key points by matching names once Setup mode exits. Name-based mapping is most useful when small changes are made to the logic [4]. A no-name mapping method for mapping key points is used when designs have completely different names. Any key point that is not mapped by Conformal is an unmapped point.

Unmapped points are divided into three categories: extra, unreachable and not-mapped [5]. Extra points in Conformal include the key points that are present in only one of the designs – either the golden or revised. Unreachable points are key points that do not have an observable point, such as a primary input. These points can usually be ignored during debugging of unmapped points. Not-mapped points are key points that are reachable but do not have a corresponding point in the logic of the corresponding design.

3.3 Comparing and Diagnosing Key Points

Once Conformal has finished executing, it will display a 'PASS' or 'NONEQ' result. A 'PASS' indicates that a design has passed Conformal LEC equivalence checking successfully. A 'NONEQ' indicates that in a design, there exists a cell or cells in the synthesized netlist that does not correspond to the golden netlist. Examples are shown in Figures 3.1 and 3.2.

```
// Command: add compared points -all
// 2 compared points added to compare list
// Command: compare
================================================================================
Compared points      PO     DFF       Total
--------------------------------------------------------------------------------
Equivalent           1      0         1
--------------------------------------------------------------------------------
Non-equivalent       0      1         1
================================================================================
// Command: report verification -compare_result
Compare Results:                                                  FAIL:NONEQ
// Command: exit -f
```

Figure 3. 2. Design Fails Conformal

```
Undriven key points                          0              176
   Unmapped                                  0              176
      Unreachable                            0              176

Primary outputs                            128              128
   Mapped                                  128              128
      Equivalent              128

State key points                           896              896
   Mapped                                  128              128
      Equivalent              128
   Unmapped                                768              768
      Unreachable                          768              768
================================================================================
// Command: usage
CPU time     : 4084.24 seconds
Memory usage : 332.07  M bytes
// Command: report verification -compare_result
Compare Results:                                                        PASS
// Command: set log file
// Command: exit -f
```

Figure 3. 3. Design Passes Conformal

3.4 Commands Used by Conformal

A ‘dofile’ with the extension ‘.do’ includes the commands executed by Conformal LEC to perform equivalence checking. Representative commands used by the dofile are summarized in Table 3.1.

Table 3. 1. Commands in Conformal

Command in Conformal	Function of command
read library -Both -Replace -sensitive -Verilog2k ~/FVSF-Benchmark/xeclib/*.v –nooptimize	Reads in the verification library files for Xilinx 7-series FPGAs named xeclib [27]. The files are provided by Xilinx with the

	Vivado software distribution for formal verification.
read design <design.v> -Verilog2k -Golden -Replace -sensitive -continuousassignment Bidirectional -nokeep_unreach –nosupply read design <design_synth.v> -Verilog2k -Revised -Replace -sensitive -continuousassignment Bidirectional -nokeep_unreach –nosupply	Reads in the golden RTL and synthesized design files based on "-Golden" and "-Revised" switches, respectively
set root module <top_module> –revised	Sets the root module name of the synthesized Verilog design for analysis
set analyze option -auto	Allows LEC to automatically perform advanced equivalence checking operations automatically
set flatten model -seq_constant	Converts flip flops driven by a constant data input to a constant value, eliminating the flip flops
set flatten model -all_seq_merge -golden	Optional line. Merges flip flops that have the same D and clock inputs into one flip flop
set system mode lec	Switches LEC from Setup to logic equivalence (LEC) mode
remodel -seq_constant	"Try harder" mode for LEC design mapping with flip flops replaced by constants
add compared points -all	Includes all compared points in the analysis
compare	Performs the logic equivalence comparison

diagnose -noneq	Print out information about flip flops and primary I/Os that did not pass logic equivalence in the design
analyze abort -compare	Optional line to have LEC perform additional steps to resolve aborts which occur for logic structures that have high combinational complexity, such as in combinational multipliers.
report verification -compare_result	Report the results as PASS, FAIL, or ABORT
exit -f	Exits Conformal

CHAPTER 4

BENCHMARKS USED IN SYNTHESIS AND FORMAL VERIFICATION

Benchmarks ranging from 134 simple to 26 complex designs are used in this work. Out of 160 designs tested, 144 designs passed Cadence Conformal evaluation using synthesized netlists created by Yosys. The other 16 designs have limitations that will be explained in this chapter. The 160 designs tested are shown in Table A.2. These designs were taken from yosys-simple [14], yosys-bigsim [15], and VTR [16] benchmark suites. Their complexities range from simple to complex. There are 119 yosys-simple designs, 8 yosys-bigsim designs, 15 VTR designs and 2 unit tests which pass equivalence checking. The designs are written in standard Verilog-2005 format [49], which is supported by synthesis tools used by all major FPGA vendors (e.g. Xilinx, Intel, and Microsemi). All designs that pass is displayed in Table A.1. Apart from those designs, additional designs were included and is discussed in the following subsection.

4.1 Work in Design Verification For 8 Additional Designs

Several additional designs were evaluated with Yosys synthesis for Vivado followed by LEC logic equivalence. Among them was *PicoRV32*, a CPU core that implements the RISC-V RV32IMC instruction set [39]. This design is included in *PicoSoC*, an SoC, along with small modules that include memory and a UART interface. The comments about each design as well as speculation as to their LEC failure are outlined in Table 4.1. The optimization switch column indicates the optimizations that were turned off to allow the design to pass equivalence checking. The other additional designs in addition to *PicoSoC* includes three designs from yosys-bigsim and four from vtr-benchmarks. The 16 designs that initially failed in yosys-simple and single-design from 160 benchmarks are revaluated.

In Table 4.1, ‘All’ optimizations include the switches: -nobram, -nolutram, -nosrl, -nocarry, -nowidelut, -noiopad, -nofsm, -nowreduce, -noopt, -noopt_dff, -noopt_muxtree, -

noopt_reduce, -noopt_merge, -noopt_clean, -noopt_expr, and -noopt_mem. 'None' indicates that no switches are used and the design passes with all optimizations turned on. 'Stall' indicates that the design does not produce an output and is stuck in Conformal LEC evaluation.

Table 4. 1. Designs tested with optimization switches

Sl. No.	Benchmark	Benchmark Suite	Pass/Fail	Optimization Switches	Reason For Failure
1	PicoSoC	-	Pass	-nobram, -nolutram -nosrl -nofsm -noopt_merge	
2	mkSMAdapter	vtr-benchmarks	Pass	None	
3	mkDelayWorker	vtr-benchmarks	Pass	None	
4	LU32PEeng	vtr-benchmarks	Pass	All	
5	mcml	vtr-benchmarks	Fail	Stall	Takes a significant amount of time in Conformal without producing a result.
6	amber23	yosys-bigsim	Fail	Stall	Takes a significant amount of time in Conformal without producing a result.
7	lm32	yosys-bigsim	Pass	All	
8	bch_verilog	yosys-bigsim	Fail	-	Verilog construct unreadable by

					Conformal. The line *poly[(nk+1)*M+:M] = 1;* is unreadable by Conformal.
9	forgen01	simple	Pass	None	
10	const_branch_finish	simple	Fail	-	No input or output
11	hierdefparam	simple	Pass	-	Conformal cannot handle generate and defparam statements. Passes when defparam statements are absent.
12	localparam_attr	simple	Pass	-	Passes when default value is more than 1. The design only consists of input and output wires.
13	operators	simple	Fail	-	Unsigned binary division is often difficult for synthesis tools, issue likely stems from there
14	param_attr	simple	Pass	-	With default value, input and output are set to 0. Passes otherwise.
15	string_format	simple	Fail	-	Only contains $display statements

16	task_func	simple	Fail	-	*add(w, <input>)* task is called three times for separate inputs, none of which work as desired. Conformal and Yosys both tie the w[7:0] outputs to GND
17	undef_eqx_nex		Fail	-	Invalid construct. Outputs are assigned to invalid mathematical operations (eg. 0/0)
18	wandwor		Invalid	-	FPGAs generally do not support wired and (wand)/wired or (wor). Conformal cannot handle a mix of modules and assign statements for the same wand or wor
19	top_bram_n1	-	Fail	-	21 registers in the modules under uart fail. All the *send_divcnt_reg* registers are non-equivalent, with other registers such as *send_pattern_reg*

					failing as it is dependent on *send_divcnt_reg* as a corresponding support point.
20	top_bram_n2	-	Fail	-	21 registers in the modules under uart fail. All the *send_divcnt_reg* registers are non-equivalent, with other registers such as *send_pattern_reg* failing as it is dependent on *send_divcnt_reg* as a corresponding support point.
21	top_bram_n3	-	Fail	-	21 registers in the modules under uart fail. All the *send_divcnt_reg* registers are non-equivalent, with other registers such as *send_pattern_reg* failing as it is dependent on *send_divcnt_reg* as a corresponding support point.

Nine out of 21 designs passed Conformal LEC equivalence checking. Of the 12 benchmarks that did not pass, seven were found to be invalid. All designs under test are shown in Table A.1.

CHAPTER 5

FORMAL VERIFICATION IN YOSYS

In this chapter we consider the use of an open-source logic equivalence and a model checker, Yosys-SMTBMC [35], that are embedded in Yosys.

5.1 An Open-Source Formal Verification Tool

Yosys includes a in-built formal verification tool, Yosys-SMTBMC [35], also called SymbiYosys [43]. This tool uses a Boolean satisfiability (SAT) solver and assert statements in Verilog to check if specified conditions are met. Yosys-SMTBMC uses an external satisfiability modulo theory (SMT) [41] solving tool, either the Z3 Theorem Prover [40] or Yices [42], to solve SMTs and perform bounded model checking (BMC). SMT determines if a mathematical formula is satisfiable. SAT solvers determine Boolean satisfiability. SMT solvers use complex formulae that involve real numbers, integers, or various data structures such as lists, arrays, bit vectors, and strings [41] to address model checking. The SMT solver can check if a condition is reachable for a design, including one written in Verilog. For example, if a Verilog design uses a counter that counts to 10 and reachability is checked for an output value of 25, a result indicating that the reachability has failed will occur. For the same design, an output value of 8 is determined to be reachable.

Unlike Conformal LEC, Yosys-SMTBMC only checks reachability for pre-specified states rather than full equivalence checking. Yosys does provides a way to perform equivalence checking using *equiv* statements [2]. These statements can be used for equivalence checking to compare a design written in RTL against a corresponding synthesized netlist. In this case, *equiv_make* statements convert the cells from RTL and synthesized netlists into corresponding *$equiv* cells. The command breaks the golden and synthesized netlists down to cells (eg. Flip flops, LUTs), which constitutes the *$equiv* cells. The *equiv* statements used in Yosys compares the *$equiv* cells in the RTL and synthesized netlists and produces an output indicating a pass or failure. Yosys commands that perform synthesis and technology mapping can be implemented in a script similar to a *dofile* in Conformal and can be used as an equivalence checker for smaller designs. For example, a

simple gate-level netlist can be produced by Yosys and can be compared to the synthesized netlist that contains cells using the Xilinx library [27].

5.2 Verifying Designs in Yosys-SMTBMC

Yosys-SMTBMC (also called SymbiYosys) was tested on the following benchmarks:

- 9 demo designs under the *smtbmc* directory in Yosys
- 10 Yosys-simple designs
- 1 Yosys-bigsim design
- 1 vtr-benchmark
- PicoRV32 CPU Core [39]

These designs use *assert* statements for to assess reachability. An *assert* statement in Verilog checks if a condition is true. An 'assertion error' is produced if a condition is false. For example, the statement *assert (a == b)* will execute normally if the values of a and b are equal but will throw an assertion error if they are not equal. All designs in Table 5.1 pass reachability analysis using *assert* statements in Verilog. In the following subsubsection, we assess the *fiedler-cooley* design from yosys-simple with Yosys-SMTBMC.

Table 5. 1. Design tested in Yosys-SMTBMC

Sl. No.	Benchmark	Benchmark Suite	Pass/Fail
1	demo1	demo	Pass
2	demo2	demo	Pass
3	demo3	demo	Pass
4	demo4	demo	Pass
5	demo5	demo	Pass
6	demo6	demo	Pass
7	demo7	demo	Pass
8	demo8	demo	Pass
9	demo9	demo	Pass

10	fielder-cooley	simple	Pass
11	aes_kexp128	simple	Pass
12	carryadd	simple	Pass
13	always01	simple	Pass
14	subbytes	simple	Pass
15	rotate	simple	Pass
16	sincos	simple	Pass
17	memory	simple	Pass
18	values	simple	Pass
19	retime	simple	Pass
20	softusb_navre	yosys-bigsim	Pass
21	diffeq1	vtr-benchmarks	Pass
22	PicoRV32	single-design	Pass

5.2.1 Files used by Yosys-SMTBMC

The *fiedler-cooley* design is a nine-bit counter that increments by 3 and decrements by 5. The design can store an input data value, count up, count down, remain the same, or set the output to a don't care value 'X'. Operation is dependent on the received 'up' and 'down' inputs. The *fiedler-cooley* design is shown in Figure 5.1.

```
// borrowed with some modifications from
// http://www.ee.ed.ac.uk/~gerard/Teach/Verilog/manual/Example/lrgeEx2/cooley.html
module up3down5(clock, data_in, up, down, carry_out, borrow_out, count_out, parity_out);

input [8:0] data_in;
input clock, up, down;

output reg [8:0] count_out;
output reg carry_out, borrow_out, parity_out;

reg [9:0] cnt_up, cnt_dn;
reg [8:0] count_nxt;

always @(posedge clock)
begin
        cnt_dn = count_out - 3'b 101;
        cnt_up = count_out + 2'b 11;

        case ({up,down})
                2'b 00 : count_nxt = data_in;
                2'b 01 : count_nxt = cnt_dn;
                2'b 10 : count_nxt = cnt_up;
                2'b 11 : count_nxt = count_out;
                default : count_nxt = 9'bX;
        endcase

        parity_out  <= ^count_nxt;
        carry_out   <= up & cnt_up[9];
        borrow_out  <= down & cnt_dn[9];
        count_out   <= count_nxt;
end

endmodule
```

Figure 5. 1. The fiedler-cooley design

This design can be modified to check reachability. An *`ifdef FORMAL* statement followed by an *assert* statement is used to check if the output port *count_out* can hold a value lower than 512 (Figure 5.2).

```
        parity_out  <= ^count_nxt;
        carry_out   <= up & cnt_up[9];
        borrow_out  <= down & cnt_dn[9];
        count_out   <= count_nxt;
end


`ifdef FORMAL
  always @(posedge clock) begin
    assert (count_out < 512);
  end
`endif
endmodule
```

Figure 5. 2. Condition to check reachability of count_out

The file must be saved as a SystemVerilog file (*fiedler-cooley.sv*) as Yosys *assert* statements are only supported only in SystemVerilog [47].

SymbiYosys files (extension *.sby*) define all parameters required to execute the design with constraints applied. The *.sby* file used to execute fiedler-cooley is shown in Figure 5.3.

```
[options]
mode prove

[engines]
smtbmc z3

[script]
read -formal fiedler-cooley.sv
prep -top up3down5

[files]
fiedler-cooley.sv
```

Figure 5. 3. The fc.sby file used by fiedler-cooley

The *.sby* file consists of the ‘option’, the ‘engine’, the ‘scripts’ to be read and the design SystemVerilog ‘files’.

The ‘options’ section allows the user to perform bounded model checking for a specified number of cycles or to prove a condition using a k-induction proof. For bounded model checking, a number of cycles is set for the condition to be proven and reachability is checked in sequence starting from a reachable state only for the number of steps provided. K-induction proves that once the design is in a running state, it cannot reach a failed state. In other words, it starts from an unreachable state and proves that every point from that state should not be a failed state. The ‘mode prove’ option checks if the condition uses k-induction with 20 cycles by default. The number of cycles to be provided is not explicit, and this option proves that the safety properties can be satisfied forever.

The ‘engines’ section specifies the engine to be used. The engines perform the mathematical proof for the condition specified in the design. If only ‘smtbmc’ is provided under the section, the tool uses the Yices SMT solver, which is the default. Figure 6.3 shows the use of the Z3 solver.

The ‘script’ section uses Yosys commands to read the input file(s), with the “–formal” switch enabling support for SystemVerilog assertions. The “prep” command performs a low-level synthesis of the design. This command is mainly used in the preparation stage of a verification flow. This command can be used interchangeably with a “synth” command,

which performs more optimizations.

The ‘files’ section includes all input design files to be used in Yosys-SMTBMC. In this case, only the fiedler-cooley.sv file is provided, but if a design uses more than one SystemVerilog file, all files should be provided.

The design is run using ‘sby’ followed by the *.sby* file to be used. For example, if the sby file is named *fc.sby* and has all file parameters set, the *fiedler-cooley* design is run by using the command ‘*sby fc.sby’*. This will generate an output indicating if the design has passed or failed, as shown in Figure 5.4.

```
SBY  9:50:56 [fc] engine_0.basecase: ##   0:00:00  Checking assumptions in step 16..
SBY  9:50:56 [fc] engine_0.basecase: ##   0:00:00  Checking assertions in step 16..
SBY  9:50:56 [fc] engine_0.basecase: ##   0:00:00  Checking assumptions in step 17..
SBY  9:50:56 [fc] engine_0.basecase: ##   0:00:00  Checking assertions in step 17..
SBY  9:50:56 [fc] engine_0.basecase: ##   0:00:00  Checking assumptions in step 18..
SBY  9:50:56 [fc] engine_0.basecase: ##   0:00:00  Checking assertions in step 18..
SBY  9:50:56 [fc] engine_0.basecase: ##   0:00:00  Checking assumptions in step 19..
SBY  9:50:56 [fc] engine_0.basecase: ##   0:00:00  Checking assertions in step 19..
SBY  9:50:56 [fc] engine_0.basecase: ##   0:00:00  Status: passed
SBY  9:50:56 [fc] engine_0.basecase: finished (returncode=0)
SBY  9:50:56 [fc] engine_0: Status returned by engine for basecase: pass
SBY  9:50:56 [fc] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY  9:50:56 [fc] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY  9:50:56 [fc] summary: engine_0 (smtbmc z3) returned pass for induction
SBY  9:50:56 [fc] summary: engine_0 (smtbmc z3) returned pass for basecase
SBY  9:50:56 [fc] summary: successful proof by k-induction.
SBY  9:50:56 [fc] DONE (PASS, rc=0)
```

Figure 5. 4. The fielder-cooley design passes SymbiYosys analysis

Yosys-SMTBMC generates a directory with the name of the *.sby* file used. This directory contains the log files produced during execution.

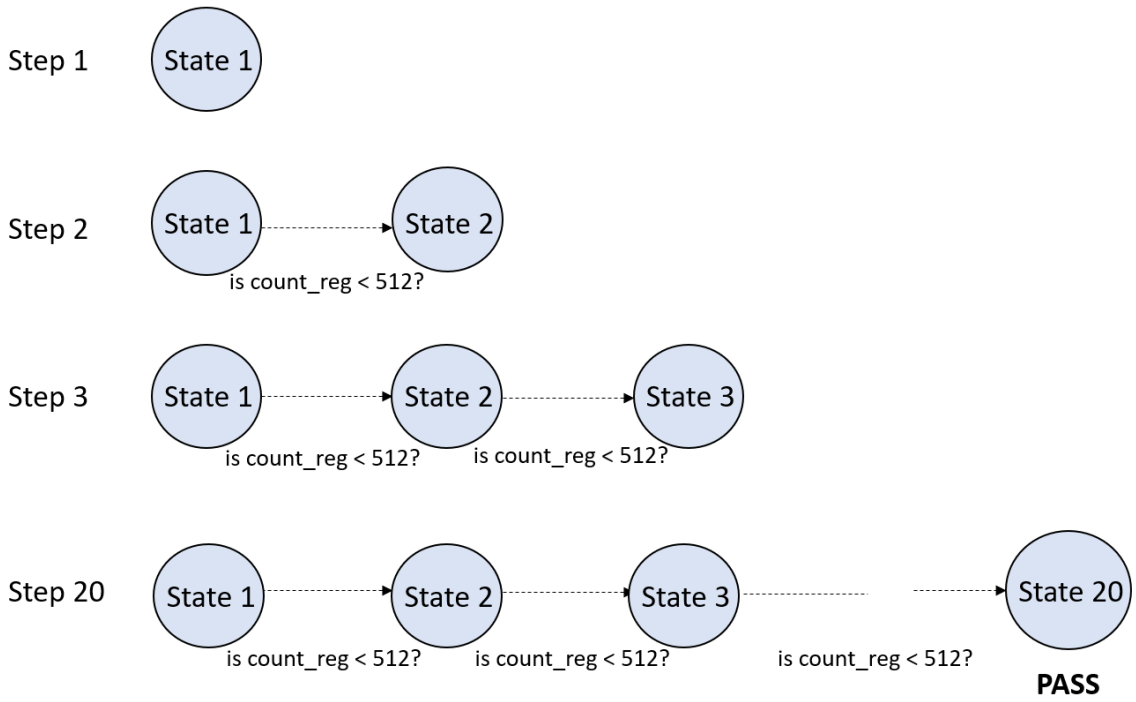

Figure 5. 5. Visualization of BMC for fiedler-cooley

If the design fails, an output indicating the design has failed is produced, along with logs to show at which trace the design failed. In the case of *fiedler-cooley*, the design would fail if a value greater than 511 is generated. This result occurs because *count_out* only includes nine bits.

Yosys-SMTBMC is helpful in locating design bugs. For example, if memories are set to initial values in a design, they can be verified by using assert statements to check if the stored memory block values match with the initial values (e.g, there has been no accidental memory change during design execution).

All 22 designs (which include 9 demo examples, 10 yosys-simple examples, 1 yosys-bigsim, 1 vtr-benchmark and the PicoRV32 CPU) have been evaluated for reachability using Yosys-SMTBMC. All designs passed this check. The simple designs tested include *aes_kexp128*, *carryadd*, *always01*, *subbytes*, *rotate*, *sincos*, *memory*, *values*, and *retime*. The yosys-bigsim design tested is *softusb_navre* and the vtr benchmark tested is *diffeq1*.

5.3 Verifying Designs Using the Yosys Equivalence Checker

A script (Figure 5.6) was used to perform equivalence checking flow, similar to the use of a *dofile* in Conformal LEC. The file runs executes Yosys commands to read the design, perform synthesis, and perform logic equivalence comparisons using *equiv* statements. An example bash script for *aes_key_expand128* is shown in Figure 5.6.

```
~/FVSF/yosys-0.9+4249/yosys -l check_3.log -p 'read_verilog aes_kexp128.v
synth -flatten -top aes_key_expand_128
splitnets -ports;;
design -stash gold

read_verilog aes_kexp128_yosys_synth.v
techmap -autoproc -map +/xilinx/cells_sim.v
splitnets -ports;;
design -stash gate

design -copy-from gold -as gold aes_key_expand_128
design -copy-from gate -as gate aes_key_expand_128
equiv_make gold gate equiv
hierarchy -top equiv
clean -purge; show
equiv_induct -seq 5
equiv_status -assert
'
```

Figure 5. 6. Bash script used to execute the Yosys equivalence checker

The commands used and their function are described in Table 5.2.

Table 5. 2. Yosys Equivalence Check commands

Sl. No.	Command Used in Yosys	Function of Command
1	read_verilog <design.v>	Reads the Verilog file of the design. Can be golden (RTL netlist) or gate (synthesized netlist)
2	synth -flatten -top <design_top>	Performs synthesis of the RTL design. Several optimizations are performed and a netlist with basic cells is produced (AND, OR, DFF, MUX etc.)
3	design -stash gold	Save the current design under the given

	design -stash gate	name and clear the current design. In this case, the RTL design is saved as **gold** and the synthesized netlist is saved as **gate**.
4	equiv_make	Creates modules with *equiv* cells from two presumably equivalent modules for the golden and gate netlists.
5	equiv_induct -seq <N>	Uses temporal induction to prove that the *equiv* cells are equal. This command is effective in proving complex sequential circuits. By default, 4 times steps are used for this proof, but the number of time steps can be increased by using the *-seq* switch.
6	equiv_status -assert	Prints status information for all selected *equiv* cells and produces an error if any unproven cell is found.
7	equiv_miter	Creates a miter module for further analysis of *equiv* cells.
8	splitnets -ports;;	Splits multi-bit nets and ports into single-bit nets and ports.
9	techmap -autoproc -map +/xilinx/cells_sim.v	Perform technology mapping on the Xilinx library.

The Yosys logic equivalence checker was evaluated with 144 benchmarks from the yosys-simple, yosys-bigsim and vtr-benchmark suites. As seen in Table 5.3, 58% of the benchmarks (84 out of 144) passed logic equivalence using the Yosys equivalence checker. Designs failed equivalence checking as a result of memory constructs that cannot be handled properly by the equivalence checker and complex finite state machine usage in designs.

A total of ~70% of yosys-simple designs pass equivalence checking. None of the more complex designs from the yosys-bigsim or vtr-benchmark suites passed.

Table 5. 3. Yosys Equivalence Checker Statistics

Design Suite	**Total Number of Designs / Designs That Pass Conformal**	**Number of Designs That Pass the Yosys Equivalence Checker**	**Percentage of Designs That Pass**
simple	121	84	69.4%
yosys-bigsim	8	0	0%
vtr-benchmarks	15	0	0%
Total	144	84	58.3%

CHAPTER 6

BUGS AND BUG FIXES IN YOSYS

This chapter examines issues encountered with using Yosys, and with using Yosys as part of the SymbiFlow tool chain. This work required bug fixes and the creation of a new synthesis library to allow for logic equivalence checking by LEC.

Initially, 127 designs out of 144 tested when compiled by Yosys as part of the SymbiFlow tool chain did not generate any errors. The errors are produced due to missing cells in a "VPR library". The VPR library includes cells used by Yosys to synthesize a register-transfer level RTL design that is compatible with VPR in SymbiFlow. It constitutes a modified xeclib [27] that makes minor changes to cells in SymbiFlow, for example, setting initial values of BRAMs to zeros instead of an unknown 'x' value. The issues arise because of Yosys bugs, or because the target Artix 7 FPGA on the Arty-35 board has insufficient logic or I/O resources. With appropriate fixes made, this number was increased to 132 designs. One Yosys bug that is independent of SymbiFlow is the swapping of 'preset' and 'clear' signals on a D flip flop during synthesis. Other bugs observed during Yosys synthesis for SymbiFlow are related to the *boundtop* and *bgm* designs. As noted subsequently, these bugs have been fixed.

The section describes modifications to the Yosys synthesis library for 7-series FPGAs in SymbiFlow to allow the library to also be used by Conformal LEC. Missing cells and poor Verilog syntax resulted in Conformal LEC not being able to read designs and perform equivalence checking. We also discuss techniques to support block RAM verification in LEC.

6.1 Bug Fixes in Yosys

6.1.1 Preset-Clear Swapping Bug

The module *dffsr2_sub* of design *dff_different_styles* in the Yosys-simple suite exposes a Yosys bug. The preset and clear wires are implemented incorrectly in the synthesized

design. Figures 6.1 and 6.2 show the incorrect and correct circuit schematics generated by Yosys and Conformal respectively.

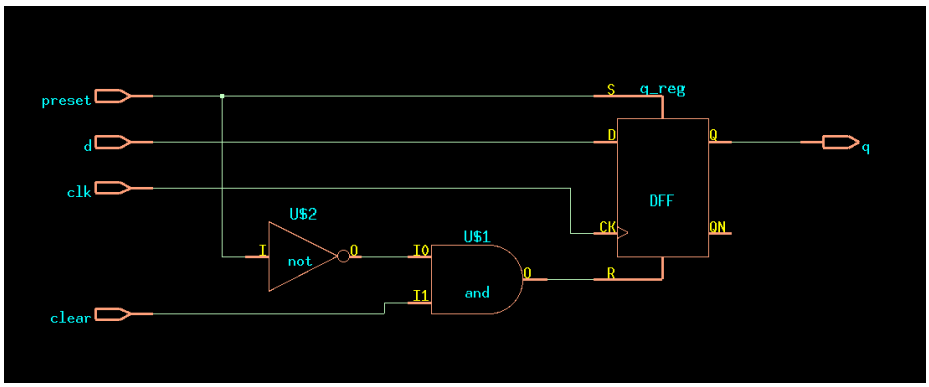

Figure 6. 1. Design dffsr2_sub synthesized by Yosys

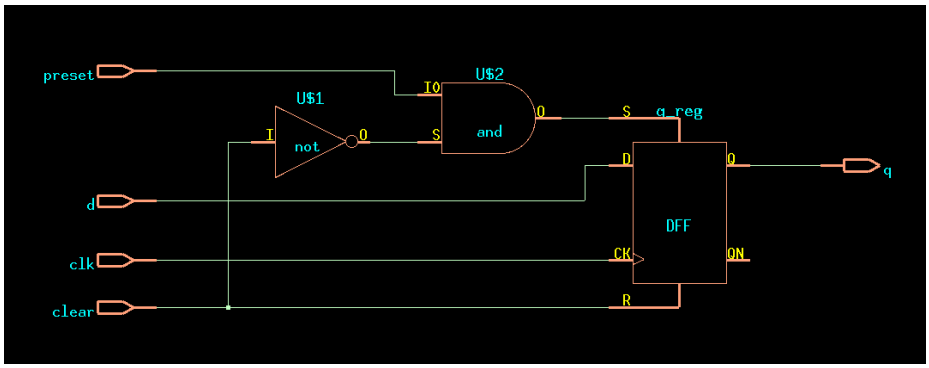

Figure 6. 2. Design dffsr2_sub synthesized by Conformal

The issue was resolved by making changes to the *proc_dff.cc* file in Yosys. The *proc_dff.cc* code identifies and processes D flip flops.

```
module dffsr2_sub(clk, preset, clear, d, q);
input clk, preset, clear, d;
output reg q;
always @(posedge clk, posedge preset, posedge clear) begin
        if (preset)
                q <= 1;
        else if (clear)
                q <= 0;
        else
                q <= d;
end
endmodule
```

Figure 6. 3. dffsr2_sub module in Verilog

Figure 6.3 shows the *dffsr2_sub* design in Verilog. The bug occurs during the 'always @' stage with the clk, preset, and clear signals in the sensitivity list. The hierarchy of the RTL design depends on the *if* and *else* statements. Thus, the hierarchy should be in the order of preset, clear, and then the default condition. Figure 6.4 shows the hierarchy changed with the bug and the hierarchy preserved with the bug fix.

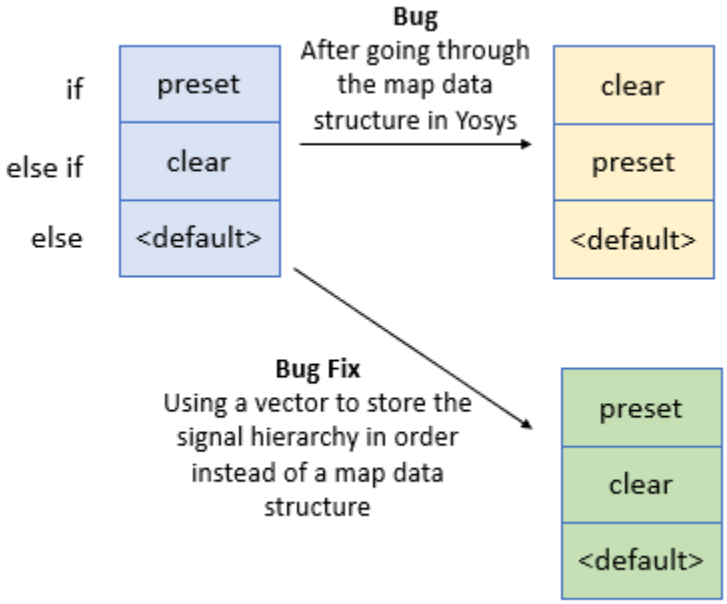

Figure 6. 4. Hierarchy of the *if* and *else* statements

Yosys prioritizes the clear synchronization signal regardless of how the Verilog RTL is written because Yosys stores the synchronization rules in a sorted C++ standard map data structure. This approach gives precedence to the clear signal over the preset during a sorting operation that occurs in the map structure.

A change is required to make the C++ standard map data structure retain the synchronization rule hierarchy. A newly created C++ standard vector titled *many_async_rules_vector* was declared alongside the already existing *many_sync_rules* as shown in Figure 6.5.

```
std::map<RTLIL::SigSpec, std::set<RTLIL::SyncRule*>> many_async_rules;
std::vector<RTLIL::SigSpec> many_async_rules_vector; // created to keep track of case rule within async resets
```

Figure 6. 5. *Many_async_rules_vector* created to keep track of the hierarchy

The *many_async_rules_vector* value receives the same insertion operations as *many_async_rules*. It is passed into the *gen_dffsr_complex()* function in *proc_dff.cc* that is used to iterate through the sensitivity list. The newly created vector *many_async_rules_vector* is used in *gen_dffsr_complex()* in place of *many_async_rules*. This approach preserves the hierarchy specified in the input RTL design and Yosys does not give any signal precedence. The code changes in the *proc_dff.cc* file in the function *proc_dff* are shown in Figures 6.6 and 6.7.

```
if (sync->type == RTLIL::SyncType::ST0 || sync->type == RTLIL::SyncType::ST1) {
        if (sync_level != NULL && sync_level != sync) {
                // log_error("Multiple level sensitive events found for this signal!\n");
                many_async_rules[rstval].insert(sync_level);
                rstval = RTLIL::SigSpec(RTLIL::State::Sz, sig.size());
        }
        rstval = RTLIL::SigSpec(RTLIL::State::Sz, sig.size());
        sig.replace(action.first, action.second, &rstval);
        sync_level = sync;
}
```

Figure 6. 6. Before bug fix in *proc_dff.cc*

```
if (sync->type == RTLIL::SyncType::ST0 || sync->type == RTLIL::SyncType::ST1) {
        if (sync_level != NULL && sync_level != sync) {
                // log_error("Multiple level sensitive events found for this signal!\n");
                many_async_rules[rstval].insert(sync_level);
                if (!std::count(many_async_rules_vector.begin(), many_async_rules_vector.end(), rstval)) //added$
                        many_async_rules_vector.push_back(rstval); // also append async rule to the vector
                rstval = RTLIL::SigSpec(RTLIL::State::Sz, sig.size());
        }
        rstval = RTLIL::SigSpec(RTLIL::State::Sz, sig.size());
        sig.replace(action.first, action.second, &rstval);
        sync_level = sync;
}
```

Figure 6. 7. After bug fix in *proc_dff.cc*

Figures 6.8 and 6.9 shows the dffsr2_sub design synthesized by Yosys before and after the code change was made in *proc_dff.cc* respectively.

```
(* top =  1  *)
(* src = "dff_different_styles.v:91.1-102.10" *)
module dffsr2_sub(clk, preset, clear, d, q);
  wire _0_;
  input clear;
  input clk;
  input d;
  input preset;
  output q;
  reg q;
  assign _0_ = preset & ~(clear);
  always @(posedge clk, posedge _0_, posedge clear)
    if (clear) q <= 1'b0;
    else if (_0_) q <= 1'b1;
    else q <= d;
endmodule
```

Figure 6. 8. Synthesized dffsr2_sub with the bug

```
(* top =  1  *)
(* src = "dff_different_styles.v:91.1-102.10" *)
module dffsr2_sub(clk, preset, clear, d, q);
  wire _0_;
  input clear;
  input clk;
  input d;
  input preset;
  output q;
  reg q;
  assign _0_ = clear & ~(preset);
  always @(posedge clk, posedge preset, posedge _0_)
    if (_0_) q <= 1'b0;
    else if (preset) q <= 1'b1;
    else q <= d;
endmodule
```

Figure 6. 9. Synthesized dffsr2_sub without the bug after bug fix

6.1.2 Synthesis Bug in boundtop

Boundtop produced an error in the post-Yosys processing file *fix_carry_xc7.py* file following synthesis. It was found that two flip flops were inferred that drove the same output by this code. This result was due to a missing default case in a Verilog switch statement in the *boundcontroller* module of *boundtop*. Figure 6.10 shows two flip flops driven by the same output, which results in the error.

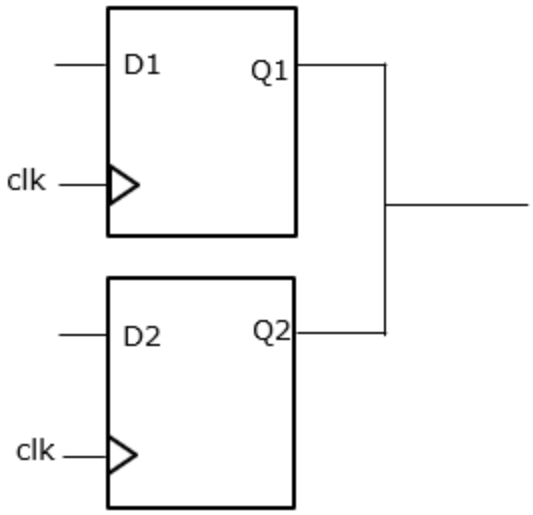

Figure 6. 10. Outputs of two flip flops drive the same wire

This issue was resolved by adding a default statement to the module. The default statement added sets the output ‘debugcount’ of the module *boundcontroller* to 0. This addition to the *boundtop* RTL prevented the creation of flip flops driven by the same output and allowed *boundtop* to generate a bitstream using SymbiFlow. Figure 6.11. shows the default statement added to *boundtop*.

```
                    end
                end
                default :
                begin
                    debugcount=13'b0;
                end
                endcase
        end
endmodule
```

Figure 6. 11. Added default statement to Boundtop

6.2 Modifying the VPR Library

The 144 benchmarks that passed equivalence checking for Yosys synthesis for Vivado were also tested when synthesized for SymbiFlow [6]. Each design include one or more input Verilog (.v) files and a constraints (.xdc) file for the Arty 35T board. SymbiFlow uses bash scripts to execute its FPGA toolchain (Yosys, packing, placement, and routing using

VPR and bitstream generation using Project X-Ray). Formal verification is performed on the output of Yosys.

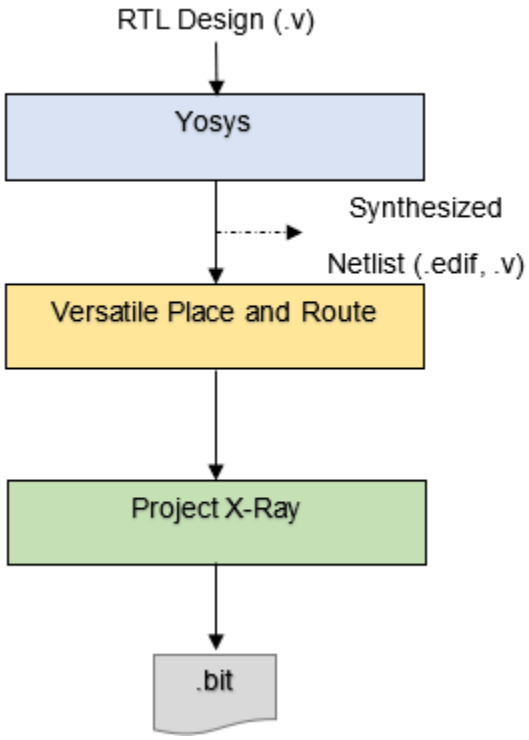

Figure 6. 12. FPGA flow from RTL to bitstream generation in SymbiFlow

Designs are synthesized in SymbiFlow using a modified version of Yosys. The differences includes the use of a slightly modified xeclib library called the VPR library. The changes in this library includes minor changes to cells such as connecting ports to ground so that the cell can pass VPR reliably. Yosys is slightly modified to include an additional technology mapping stage. It maps the standard gate-level cells to xeclib and then maps this to the VPR library. It also uses additional post-processing files to clean up any unused wires, or to resolve carry congestion. A synthesized design that is generated by the SymbiFlow optimized version of Yosys can be used by Conformal for equivalence checking. Out of the initial 144 designs that were tested, 123 passed logic equivalence testing. These designs are shown in Table A.3.

The VPR library includes cells used by Yosys to synthesize a register-transfer level RTL design that is compatible with VPR in SymbiFlow. Gate-level designs synthesized by

Yosys with the library can be used in VPR for logic packing into LUTs as part of SymbiFlow. The list of files in the library includes the following.

The *cells_sim.v* library maps cells from the Xilinx technology library files present in Yosys to a Xilinx 7-series VPR technology library. The design is initially mapped to cells from the Xilinx library (xeclib) [27], and then is processed with a second mapping pass using the VPR library. These steps are performed since some cells in the Xilinx library are not compatible with VPR. Cells in this VPR library make changes to the cells to allow them to pass VPR without errors.

The *cells_map.v* file includes cells used in a simple technology mapping pass in which unused ports on cells are removed. Most of these cells include flip flops such as the cells FDRE, FDSE, FDCE, FDPE and the newly added LDCE (transparent data latch with asynchronous clear and gate enable). This library file also includes lookup tables (LUT1, LUT2, LUT3, LUT4, LUT5 and LUT6) and distributed RAMs (RAM128X1S, RAM128X1D, RAM256X1S, RAM32X1D, RAM32M, RAM32X2S, RAM32X1S, RAM64M, RAM64X1D and RAM64X1S). Block RAMs (RAMB18E1 and RAMB36E1) and DSP block instance DSP48E1 are also included. The *carry_map.v* file converts some CARRY4 outputs to LUTs to resolve carry congestion. This action is required because VPR cannot reliably resolve SLICEL or SLICEM output usage when both Out (O) and Carry Out (CO) outputs are used. If both O and CO outputs are used, the CO output is computed using a LUT. The *clean_carry_map.v* file contains modules used by *carry_map.v*. The *retarget.v* file includes cells to remap specific D flip flops (FD) into reset-enabled D flip flops (FDRE). The *unmap.v* file unmaps the ‘CARRY_COUT’ cell specified in *carry_map.v* and maps it to a standard ‘CARRY’ cell instead.

Cell modifications were made in *cells_sim.v* and *cells_map.v*.

6.2.1 Syntax Errors in VPR Library

The updates made to *cells_sim.v* and *cells_map.v* included Verilog coding implementation updates. The files were originally not written using Verilog-2005 standards, but Yosys could parse the Verilog files and automatically make syntax corrections. Since Conformal LEC is unable to make such corrections, portions of the libraries were rewritten to meet Verilog-2005 standards. Numerous missing brackets and

punctuation (e.g., missing comma at the end of a module declaration) in the original library files were corrected. Figures 6.13 and 6.14 shows the FDRE flip flop before and after the syntax changes made to *cells_map.v*.

```
FDRE_ZINI #(.ZINI(!|INIT)), .IS_C_INVERTED(|0))
  _TECHMAP_REPLACE_ (.D(D), .Q(Q), .C(C), .CE(CE_SIG), .R(SR_SIG));
```

Figure 6. 13. Before the Verilog syntax change

```
FDRE_ZINI #(.ZINI(!(|INIT)), .IS_C_INVERTED(|0))
  _TECHMAP_REPLACE_ (.D(D), .Q(Q), .C(C), .CE(CE_SIG), .R(SR_SIG));
```

Figure 6. 14. After the Verilog syntax change

The '(' bracket between '!' and '|' in 'INIT' is included to make it syntactically correct. The cells that were changed to include the syntax that could be read by Conformal LEC are FDRE, FDSE, FDCE, and FDPE.

6.2.2 Addition of Cells to VPR Library

The LDCE cell is a D latch with clear and enable signals. The LDCE cell was added to both the *cells_sim.v* and *cells_map.v* library files. Figures 6.15 and 6.16 shows the addition of the LDCE library to *cells_sim.v* and *cells_map.v* respectively. The LDCE_1 denotes the inverted LDCE that is included in *cells_map.v*. LDCE was added in the same format as flip flops FDRE and FDSE in the VPR library.

```
module LDCE_ZINI (
  output reg Q,
  //(* invertible_pin = "IS_CLR_INVERTED" *)
  input CLR,
  input D,
  //(* invertible_pin = "IS_G_INVERTED" *)
  input G,
  input GE
);
  parameter [0:0] INIT = 1'b0;
  //parameter [0:0] IS_CLR_INVERTED = 1'b0;
  parameter [0:0] IS_G_INVERTED = 1'b0;
  //parameter MSGON = "TRUE";
  //parameter XON = "TRUE";
  initial Q <= !INIT;
  /*wire clr = CLR ^ IS_CLR_INVERTED;
  wire g = G ^ IS_G_INVERTED;
  always @*
    if (clr) Q <= 1'b0;
    else if (GE && g) Q <= D;*/
  generate case (|IS_G_INVERTED)
    1'b0: always @(posedge G, posedge CLR) if (CLR) Q <= 1'b0; else if (GE) Q <= D;
    1'b1: always @(negedge G, posedge CLR) if (CLR) Q <= 1'b0; else if (GE) Q <= D;
  endcase endgenerate

endmodule
```

Figure 6. 15. LDCE added to *cells_sim.v*

```
module LDCE (output reg Q, input G, GE, D, CLR);

parameter [0:0] INIT = 1'b0;
//parameter MSGON = "TRUE";
//parameter XON = "TRUE";
parameter [0:0] IS_G_INVERTED = 1'b0;
wire CE_SIG;
wire SR_SIG;

CESR_MUX cesr_mux(
    .CE(GE),
    .SR(CLR),
    .CE_OUT(CE_SIG),
    .SR_OUT(SR_SIG)
);

LDCE_ZINI #(.INIT(!(|INIT)), .IS_G_INVERTED(|0))
  _TECHMAP_REPLACE_ (.D(D), .Q(Q), .G(G), .GE(CE_SIG), .CLR(SR_SIG));

endmodule

module LDCE_1 (output reg Q, input G, GE, D, CLR);

parameter [0:0] INIT = 1'b0;
//parameter MSGON = "TRUE";
//parameter XON = "TRUE";
parameter [0:0] IS_G_INVERTED = 1'b0;
wire CE_SIG;
wire SR_SIG;

CESR_MUX cesr_mux(
    .CE(GE),
    .SR(CLR),
    .CE_OUT(CE_SIG),
    .SR_OUT(SR_SIG)
);

LDCE_ZINI #(.INIT(!(|INIT)), .IS_G_INVERTED(|1))
  _TECHMAP_REPLACE_ (.D(D), .Q(Q), .G(G), .GE(CE_SIG), .CLR(SR_SIG));

endmodule
```

Figure 6. 16. LDCE added to *cells_map.v*

Cells for 2:1 multiplexers named MUXF6, MUXF7, and MUXF8 were added to *cells_sim.v*. Multiplexers MUXF6, MUXF7, MUXF8 and MUXF9 were copied from the xeclib library [27] to *cells_sim.v*.

6.2.3 Formally Verifiable Block RAMs

The VPR library was modified to support block RAMs for LEC logic equivalence testing. Block RAMs [45] (BRAMs) are bulk data storage resources located in FPGAs. In

Xilinx Artix-7 FPGAs, BRAMs contain 36 or 18 Kbits of storage (RAMB36 or RAMB18). Although Yosys can infer BRAMs from RTL code, Conformal LEC cannot directly verify these inferred BRAMs since it is unaware how Yosys has mapped memory to BRAMs. To support logic equivalence testing, a user must assign memory structures to BRAMs prior to Yosys synthesis, effectively creating a golden register-transfer level (RTL) memory file for use by both Yosys and LEC. The VPR library was modified to allow Artix-7 block RAMs to be formally verifiable by LEC both 36 and 18 Kbit cells were added to the library. Example ports for an 18 Kbit BRAM are shown in Figure 6.17.

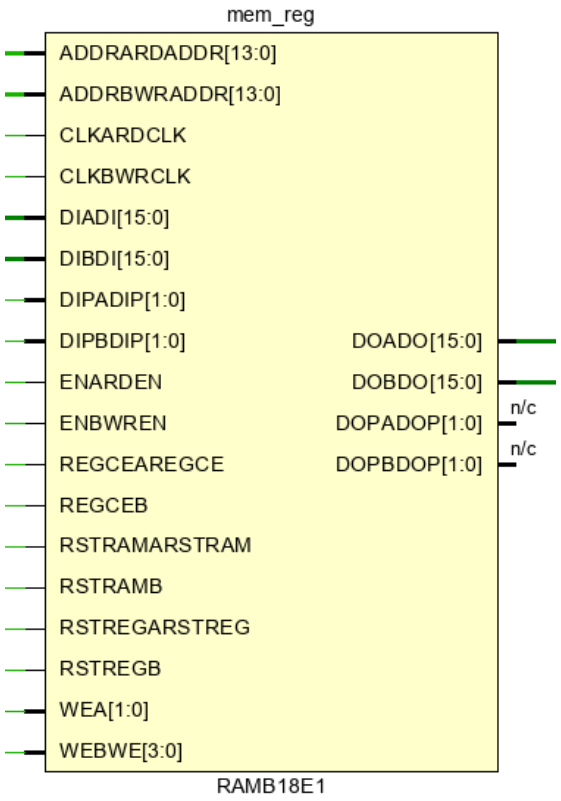

Figure 6. 17. RAMB18E1 Xilinx primitiveT

A BRAM36 module that instantiates a Xilinx RAMB primitive of 36-bit width and 512 words deep is shown in Figure 6.18. The parameter statement sets the bit width and word size of the module. The ports of a BRAM that is typically inferred by the synthesis tool and mapped to a Xilinx primitive of a BRAM are assigned in the module.

```
module BRAM36
  #(parameter bits = 36,
    parameter words = 512,
    parameter addrWidth = $clog2(words))
  ( datain,
    dataout,
    raddr,
    waddr,
    re,
    we,
    clkr,
    clkw);
  input [bits-1:0] datain;
  output [bits-1:0] dataout;
  input [addrWidth-1:0] raddr, waddr;
  input we, re, clkr, clkw;
  RAMB18E1 #(
    .IS_CLKARDCLK_INVERTED(1'h0),
    .IS_CLKBWRCLK_INVERTED(1'h0),
    .RAM_MODE("SDP"),
    .READ_WIDTH_A('d36),
    .READ_WIDTH_B('d36),
    .SIM_DEVICE("7SERIES"),
    .WRITE_MODE_A("READ_FIRST"),
    .WRITE_MODE_B("READ_FIRST"),
    .WRITE_WIDTH_A('d36),
    .WRITE_WIDTH_B('d36)
  ) U0 (
    .ADDRARDADDR({raddr, 7'b1111111}),
    .ADDRBWRADDR({waddr, 7'b1111111}),
    .CLKARDCLK(clkr),
    .CLKBWRCLK(clkw),
    .DIADI(32'h00000000),
    .DIBDI(datain[31:0]),
    .DIPADIP(4'b0000),
    .DIPBDIP(datain[35:32]),
    .DOADO(dataout[31:0]),
    .DOPADOP(dataout[35:32]),
    .ENARDEN(re),
    .ENBWREN(we),
    .REGCEAREGCE(1'h1),
    .REGCEB(1'h1),
    .RSTRAMARSTRAM(1'h0),
    .RSTRAMB(1'h0),
    .RSTREGARSTREG(1'h0),
    .RSTREGB(1'h0),
    .WEBWE(8'hff)
  );

endmodule
```

Figure 6. 18. BRAM36 module

READ_WIDTH_A and READ_WIDTH_B are parameters in the *cells_map.v* file that decide whether an 18K or a 36K BRAM should be used based on the bit size. The library consists of if-conditions that checks whether the bit width is greater than 18 bits. If it is, an error is produced. A 36-bit wide and 512-word deep BRAM can use an 18K BRAM, but an error is thrown by SymbiFlow during synthesis. The *cells_map.v* VPR library file was modified to increase the limit of the width parameter to 36 bits. A similar fix was performed

with the 36K BRAMs to support a width of 72 bits.

In some cases, BRAMs are given initial values during instantiation. SymbiFlow uses `define statements and parameters to initialize values in 36 Kbit BRAMs. The `define statements create an INIT_BLOCK that initializes values. Each line of the BRAM is initialized to 0 using these statements. This block, however, crashes Conformal LEC. BRAMs usually have initial values set to 'X' or unknown values by default. Thus, removing this block allows the design to go through Conformal without crashing.

In BRAMs, the clock is connected to ports CLKARDCLK (read clock) and CLKBWRCLK (write clock) in a single port RAM. In a dual port RAM, they can act as clocks for two ports: the A port and the B port. The VPR library sets a condition for the CLKARDCLK and CLKBWRCLK based on the parameters _TECHMAP_CONSTMSK_CLKARDCLK and _TECHMAP_CONSTVAL_CLKARDCLK_. The parameter _TECHMAP_CONSTMSK_CLKARDCLK_ is set to 0 by default and is used to check if an error is generated during synthesis. The _TECHMAP_CONSTVAL_CLKARDCLK_ parameter checks if the signal is a constant value, and if the condition evaluates to 0, CLKARDCLK is set to 1. The if-statement to connect the clock to the CLKARDCLK port is shown in Figure 6.19.

```
  wire clkardclk =
(_TECHMAP_CONSTMSK_CLKARDCLK_ == 1)  ? 1'b1 :
(_TECHMAP_CONSTVAL_CLKARDCLK_ === 0) ? 1'b1 : CLKARDCLK;
```

Figure 6. 19. clkardclk wire depends on parameter values

This statement indicates that the _TECHMAP_CONSTMSK_CLKARDCLK_ and _TECHMAP_CONSTVAL_CLKARDCLK_ should be 0 and 1 respectively for the connection to be made to CLKARDCLK. Yosys can infer this statement in the VPR library correctly but Conformal cannot. Conformal does not connect the clock to the CLKARDCLK port. Our fix directly connects the wire *clkardclk* to the BRAM port CLKARDCLK. This fix allows Conformal to make the appropriate connections. Figure 6.20 shows the change made to the VPR library.

```
 // wire clkardclk =
//(_TECHMAP_CONSTMSK_CLKARDCLK_ == 1)  ? 1'b1 :
//(_TECHMAP_CONSTVAL_CLKARDCLK_ === 0) ? 1'b1 : CLKARDCLK;
 wire clkardclk = CLKARDCLK;
```

Figure 6. 20. Direct connection of clkardclk wire to CLKARDCLK port

The similar fix was performed for the CLKBWRCLK port for both 18K and 36K BRAMs in the VPR library.

Using the three fixes, BRAMs can be formally verified in Conformal LEC. Logical memories of various bit and word sizes were tested, from 1-bit, 32768 words to 72-bit, 16384 words.

CHAPTER 7

RESULTS

In this chapter we compare Yosys synthesis results for designs with optimizations turned on and turned off. These results are compared to Vivado synthesis results with optimizations. Place-and-route without design logic optimization is performed by Xilinx Vivado. All designs were mapped to the Xilinx Artix-7 FPGA architecture. Timing constraints were not used for any synthesis activities.

Table 7.1 shows the results of synthesizing five designs. Not surprisingly, benchmarks synthesized with optimizations yielded smaller designs. This result shows area improvement due to optimizations such as FSM recoding and register merging. All synthesized designs shown in the table pass LEC equivalence checking.

Table 7. 1. Logic Design Area Comparison (LUTs and Flip Flops)

Design Name and optimizations turned off	**Area Metrics (Lower is better)**					
	Number of LUTs			**Number of Flip Flops**		
	Yosys without optimizations	**Yosys with optimizations**	Xilinx Vivado	Yosys without optimizations	**Yosys with optimizations**	Xilinx Vivado
softusb_navre (-nofsm)	1018	**1105**	877	340	**353**	340
blob_merge (-nofsm -nowreduce -noopt_dff -noopt_merge -keepff)	3461	**3659**	5457	552	**575**	575
stereovision0	10396	**3611**	3359	13258	**9069**	8081

(-nobram -nolutram -nosrl -nofsm -nowreduce -noopt_dff -noopt_merge -keepff)						
ch_intrinsic (-nobram -nolutram -nosrl -nofsm -nowreduce -noopt_dff -noopt_merge -keepff)	209	**92**	29	496	**211**	82
sha (-nofsm -nowreduce -noopt_dff -noopt_merge -keepff)	1400	**1479**	1245	910	**893**	903
mkPktMerge (-nobram -nolutram -nosrl -nofsm -nowreduce -noopt_dff -noopt_merge -keepff)	10371	**609**	121	7926	**495**	171

The BRAMs and DSP blocks for designs that generate them is shown in Table 7.2.

Table 7. 2. Logic Design Area Comparison (BRAMs and DSP Blocks)

Design Name and optimizations turned off	**Area Metrics (Lower is better)**					
	Number of BRAMs			**Number of DSP Blocks**		
	Yosys without optimizations	**Yosys with optimizations**	Xilinx Vivado	Yosys without optimizations	**Yosys with optimizations**	Xilinx Vivado
ch_intrinsic	0	**0**	1	0	**0**	0
mkPktMerge	0	**0**	5	0	**0**	0

Table 7. 3. Design Performance Comparison

Design Name	**Performance Metrics (Higher is Better)**		
	Clock Speed (MHz)		
	Yosys without optimizations (Formally Verifiable)	**Yosys with optimizations (Formally Verifiable)**	Xilinx Vivado (Not Formally Verifiable)
softusb_navre	73.56	**80.45**	92.91
blob_merge	48.48	**49.04**	69.74
stereovision0	129.05	**200.80**	221.43
ch_intrinsic	265.25	**270.70**	360.23
sha	122.02	**120.29**	128.65

In addition to the five designs shown in Table 7.1, 55 designs were subjected to the same experiments. Following synthesis, each design was subjected to Vivado place and route to determine performance. The timing constraints used includes setting an initial clock frequency to 1 GHz and reducing the clock speed until a positive skew is generated to obtain maximum frequency. The average clock speed of the fifty-five designs synthesized by Yosys with optimizations is 311.3 MHz and the average clock speed of the designs synthesized by Vivado is 344.8 MHz - a 10% difference. The fifty-five designs are used as

several designs from yosys-simple are combinational logic and do not have clocks set in their design. Several submodules were also included as benchmarks, but these designs did not have clocks. Thus, the submodules were not considered in the results. Some designs from yosys-bigsim and vtr-benchmarks had too many I/Os or resources for the Arty-35 board, therefore place-and-route could not be performed. All synthesized designs passed equivalence checking.

Table 7. 4. Arithmetic and Geometric Means of Designs Synthesized in Yosys and Vivado

	Average of 55 Synthesized Designs in Yosys					**Average of 55 Synthesized Designs in Vivado**				
	LUT	**Flip Flop**	**BRAM**	**DSP**	**Clock Frequency**	**LUT**	**Flip Flop**	**BRAM**	**DSP**	**Clock Frequency**
Arithmetic Mean	1468.20	887.76	0	12.53	311.29	862.18	716.76	0.02	4.96	344.80
Geometric Mean	26.56	10.61	0	1.36	240.80	20.98	8.75	0	1.19	287.25

CHAPTER 8

CONCLUSION

In this thesis document, we presented a robust synthesis flow for formal verification of designs synthesized with an open-source tool, Yosys. Over 144 benchmarks ranging from simple to complex designs were put through Yosys to create synthesized netlists. The designs pass equivalence checking using Conformal LEC. Several complex benchmarks passed equivalence checking following source code modifications within Yosys to allow for the ability to turn off optimization passes. The performance of designs synthesized with Yosys was compared to the results obtained by Vivado, which does not produce formally verifiable netlists but allows for improved area and improved performance. We evaluated open-source tools for formal verification that includes bounded model checking and equivalence checking. We found that all 13 designs tested for bounded model checking passed, and 58% of designs pass the open-source equivalence checker.

In this work, we fixed several Yosys bugs that affected synthesis for SymbiFlow, an open-source FPGA toolchain. The fixes allow designs to successfully complete bitstream generation using other SymbiFlow tools. Additions were made to a LEC verification library to fix Verilog syntax errors and add cells needed for verification. SymbiFlow Block RAM formal verification support was added to Conformal through modifications to the VPR library. Support for 72-bit wide BRAMs is now available.

9.1 Future Work

9.1.1 Additional Benchmarks

As this work was limited to around 160 benchmarks, there is potential to add more benchmarks for synthesis and formal verification. The work can also be extended to additional designs synthesized for SymbiFlow.

9.1.2 Protection of Hints Files

The generation of hints files in Yosys such as during FSM recoding and flip flop merging helps logic equivalence testing. These hints files should be protected so they cannot be examined by malicious attackers. This action can be accomplished by file encryption after synthesis. A logic equivalence tool could decrypt the files during formal verification.

9.1.3 Robust Yosys Equivalence Checker

Currently, only simple designs pass the Yosys equivalence checker while more complex designs generally fail. The checker could be modified to support memories and more complex logic. The current tool splits a design into cells and checks equivalence using SAT solvers. A Conformal-based approach could be implemented in which a Boolean expression is generated for the golden and revised netlists and compared.

APPENDIX

Table A. 1. All Verilog Benchmarks

Sl. No.	Design Name	Top Module	Benchmark Suite	Pass/Fail
1	aes_kexp128	aes_key_expand_128	simple	Pass
2	always01	uut_always01	simple	Pass
3	always02	uut_always02	simple	Pass
4	always03	uut_always03	simple	Pass
5	arraycells	array_test001	simple	Pass
6	arraycells	aoi12	simple	Pass
7	arrays01	uut_arrays01	simple	Pass
8	attrib01_module	bar	simple	Pass
9	attrib01_module	foo	simple	Pass
10	attrib02_port_decl	bar	simple	Pass
11	attrib02_port_decl	foo	simple	Pass
12	attrib03_parameter	foo	simple	Pass
13	attrib04_net_var	bar	simple	Pass
14	attrib04_net_var	foo	simple	Pass
15	attrib06_operator_suffix	bar	simple	Pass
16	attrib06_operator_suffix	foo	simple	Pass
17	attrib08_mod_inst	bar	simple	Pass
18	attrib08_mod_inst	foo	simple	Pass
19	attrib09_case	bar	simple	Pass
20	attrib09_case	foo	simple	Pass
21	carryadd	carryadd	simple	Pass
22	constmuldivmod	constmuldivmod	simple	Pass
23	dff_different_styles	dff	simple	Pass

24	dff_different_styles	dffa	simple	Pass
25	dff_different_styles	dffa1	simple	Pass
26	dff_different_styles	dffa2	simple	Pass
27	dff_different_styles	dffa4	simple	Pass
28	dff_different_styles	dffsr1	simple	Pass
29	dff_different_styles	dffsr2	simple	Pass
30	dff_different_styles	dffsr2_sub	simple	Pass
31	dff_init	dff0_test	simple	Pass
32	dff_init	dff1_test	simple	Pass
33	dff_init	dff0a_test	simple	Pass
34	dff_init	dff1a_test	simple	Pass
35	dff_init	dff_test_997	simple	Pass
36	dynslice	dynslice	simple	Pass
37	fiedler-cooley	up3down5	simple	Pass
38	forgen02	uut_forgen02	simple	Pass
39	forloops	forloops01	simple	Pass
40	forloops	forloops02	simple	Pass
41	fsm	fsm_test	simple	Pass
42	generate	gen_test1	simple	Pass
43	generate	gen_test2	simple	Pass
44	generate	gen_test3	simple	Pass
45	generate	gen_test4	simple	Pass
46	generate	gen_test5	simple	Pass
47	generate	gen_test6	simple	Pass
48	graphtest	graphtest	simple	Pass
49	hierarchy	top	simple	Pass
50	hierdefparam	hierdefparam_a	simple	Pass
51	hierdefparam	hierdefparam_b	simple	Pass
52	i2c_master_tests	i2c_test01	simple	Pass
53	i2c_master_tests	i2c_test02	simple	Pass

54	implicit_ports	alu	simple	Pass
55	implicit_ports	named_ports	simple	Pass
56	loops	aes	simple	Pass
57	macros	test_def	simple	Pass
58	macros	test_ifdef	simple	Pass
59	macros	test_comment_in_macro	simple	Pass
60	mem2reg	mem2reg_test1	simple	Pass
61	mem2reg	mem2reg_test2	simple	Pass
62	mem2reg	mem2reg_test3	simple	Pass
63	mem2reg	mem2reg_test4	simple	Pass
64	mem2reg	mem2reg_test5	simple	Pass
65	mem2reg	mem2reg_test6	simple	Pass
66	mem_arst	MyMem	simple	Pass
67	memory	memtest00	simple	Pass
68	memory	memtest01	simple	Pass
69	memory	memtest03	simple	Pass
70	memory	memtest04	simple	Pass
71	memory	memtest05	simple	Pass
72	memory	memtest10	simple	Pass
73	memory	memtest11	simple	Pass
74	memory	memtest12	simple	Pass
75	memory	memtest13	simple	Pass
76	multiplier	Multiplier_flat	simple	Pass
77	multiplier	Multiplier_2D	simple	Pass
78	multiplier	FullAdder	simple	Pass
79	muxtree	usb_tx_phy	simple	Pass
80	muxtree	default_cases	simple	Pass
81	muxtree	select_leaves	simple	Pass
82	omsp_dbg_uart	omsp_dbg_uart	simple	Pass

83	paramods	pm_test1	simple	Pass
84	paramods	pm_test2	simple	Pass
85	paramods	pm_test3	simple	Pass
86	partsel	partsel_test001	simple	Pass
87	partsel	partsel_test002	simple	Pass
88	partsel	partsel_test003	simple	Pass
89	partsel	partsel_test004	simple	Pass
90	partsel	partsel_test005	simple	Pass
91	partsel	partsel_test006	simple	Pass
92	partsel	partsel_test007	simple	Pass
93	process	blocking_cond	simple	Pass
94	process	uut	simple	Pass
95	process	uart	simple	Pass
96	realexpr	demo_001	simple	Pass
97	realexpr	demo_004	simple	Pass
98	repwhile	repwhile_test001	simple	Pass
99	retime	retime_test	simple	Pass
100	rotate	a23_barrel_shift_fpga_rotate	simple	Pass
101	scopes	scopes_test_01	simple	Pass
102	signedexpr	signed_test01	simple	Pass
103	sincos	d	simple	Pass
104	subbytes	subbytes_00	simple	Pass
105	task_func	task_func_test03	simple	Pass
106	task_func	task_func_test04	simple	Pass
107	task_func	task_func_test05	simple	Pass
108	usb_phy_tests	usb_phy_test01	simple	Pass
109	values	test_signed	simple	Pass
110	values	test_const	simple	Pass
111	vloghammer	test01	simple	Pass

112	vloghammer	test02	simple	Pass
113	vloghammer	test03	simple	Pass
114	vloghammer	test04	simple	Pass
115	vloghammer	test07	simple	Pass
116	vloghammer	test08	simple	Pass
117	vloghammer	test09	simple	Pass
118	vloghammer	test10	simple	Pass
119	wreduce	wreduce_test0	simple	Pass
120	wreduce	wreduce_test1	simple	Pass
121	xilinxip	xilinxIP	single-design	Pass
122	counter_bram	counter_bram	single-design	Pass
123	aes_5cycle_2stage	aes_top	yosys-bigsim	Pass
124	bch_encode	bch_encode	yosys-bigsim	Pass
125	openmsp430	openMSP430	yosys-bigsim	Pass
126	reed_solomon_decoder	RS_dec	yosys-bigsim	Pass
127	softusb_navre	softusb_navre	yosys-bigsim	Pass
128	verilog-pong	top	yosys-bigsim	Pass
129	scripts	RAM256X8	yosys-bigsim	Pass
130	elliptic_curve_group	point_scalar_mult	yosys-bigsim	Pass
131	bgm	bgm	vtr-benchmarks	Pass
132	blob_merge	RLE_BlobMerging	vtr-benchmarks	Pass
133	boundtop	paj_boundtop_hierarchy_no_mem	vtr-benchmarks	Pass
134	raygentop	paj_raygentop_hierarchy_no_mem	vtr-benchmarks	Pass
135	ch_intrinsics	memset	vtr-benchmarks	Pass
136	diffeq1	diffeq_paj_convert	vtr-benchmarks	Pass
137	diffeq2	diffeq_f_systemC	vtr-benchmarks	Pass
138	mkPktMerge	mkPktMerge	vtr-benchmarks	Pass

139	LU8PEeng	LU8PEEng	vtr-benchmarks	Pass
140	stereovision0	sv_chip0_hierarchy_no_mem	vtr-benchmarks	Pass
141	stereovision1	sv_chip1_hierarchy_no_mem	vtr-benchmarks	Pass
142	stereovision2	sv_chip2_hierarchy_no_mem	vtr-benchmarks	Pass
143	stereovision3	sv_chip3_hierarchy_no_mem	vtr-benchmarks	Pass
144	or1200	or1200_flat	vtr-benchmarks	Pass
145	sha	sha1	vtr-benchmarks	Pass
146	PicoSoC	top	single-design	Pass
147	mkSMAdapter	mkSMAdapter4B	vtr-benchmarks	Pass
148	mkDelayWorker	mkDelayWorker32B	vtr-benchmarks	Pass
149	LU32PEeng	LU32PEEng	vtr-benchmarks	Pass
150	mcml	mcml	vtr-benchmarks	Fail
151	amber23	a23_core	yosys-bigsim	Fail
152	lm32	lm32_top	yosys-bigsim	Pass
153	bch_verilog	bch_decode	yosys-bigsim	Fail
154	forgen01	uut_forgen01	simple	Pass
155	cons_branch_finish	case_branch_finish_top	simple	Fail
156	hierdefparam	hierdefparam_top	simple	Pass
157	localparams_attr	uut_localparam_attr	simple	Pass
158	operators	optest	simple	Fail
159	param_attr	uut_param_attr	simple	Pass
160	string_format	string_format_top	simple	Fail
161	task_func_test01	task_func_test01	simple	Fail
162	task_func_test02	task_func_test02	simple	Fail
163	undef_eqx_nex	undef_eqx_nex	simple	Fail

164	wandwor	wandwor_test0	simple	Fail
165	wandwor	wandwor_test1	simple	Fail
166	top_bram_n1	top	single-design	Fail
167	top_bram_n2	top	single-design	Fail
168	top_bram_n3	top	single-design	Fail

Table A. 2. Initial 160 Designs Tested

Sl. No.	**Design Name**	**Top Module**	**Benchmark Suite**
1	aes_kexp128	aes_key_expand_128	simple
2	always01	uut_always01	simple
3	always02	uut_always02	simple
4	always03	uut_always03	simple
5	arraycells	array_test001	simple
6	arraycells	aoi12	simple
7	arrays01	uut_arrays01	simple
8	attrib01_module	bar	simple
9	attrib01_module	foo	simple
10	attrib02_port_decl	bar	simple
11	attrib02_port_decl	foo	simple
12	attrib03_parameter	foo	simple
13	attrib04_net_var	bar	simple
14	attrib04_net_var	foo	simple
15	attrib06_operator_suffix	bar	simple
16	attrib06_operator_suffix	foo	simple
17	attrib08_mod_inst	bar	simple
18	attrib08_mod_inst	foo	simple

19	attrib09_case	bar	simple
20	attrib09_case	foo	simple
21	carryadd	carryadd	simple
22	constmuldivmod	constmuldivmod	simple
23	dff_different_styles	dff	simple
24	dff_different_styles	dffa	simple
25	dff_different_styles	dffa1	simple
26	dff_different_styles	dffa2	simple
27	dff_different_styles	dffa4	simple
28	dff_different_styles	dffsr1	simple
29	dff_different_styles	dffsr2	simple
30	dff_different_styles	dffsr2_sub	simple
31	dff_init	dff0_test	simple
32	dff_init	dff1_test	simple
33	dff_init	dff0a_test	simple
34	dff_init	dff1a_test	simple
35	dff_init	dff_test_997	simple
36	dynslice	dynslice	simple
37	fiedler-cooley	up3down5	simple
38	forgen02	uut_forgen02	simple
39	forloops	forloops01	simple
40	forloops	forloops02	simple
41	fsm	fsm_test	simple
42	generate	gen_test1	simple
43	generate	gen_test2	simple
44	generate	gen_test3	simple
45	generate	gen_test4	simple
46	generate	gen_test5	simple
47	generate	gen_test6	simple
48	graphtest	graphtest	simple

49	hierarchy	top	simple
50	hierdefparam	hierdefparam_a	simple
51	hierdefparam	hierdefparam_b	simple
52	i2c_master_tests	i2c_test01	simple
53	i2c_master_tests	i2c_test02	simple
54	implicit_ports	alu	simple
55	implicit_ports	named_ports	simple
56	loops	aes	simple
57	macros	test_def	simple
58	macros	test_ifdef	simple
59	macros	test_comment_in_macro	simple
60	mem2reg	mem2reg_test1	simple
61	mem2reg	mem2reg_test2	simple
62	mem2reg	mem2reg_test3	simple
63	mem2reg	mem2reg_test4	simple
64	mem2reg	mem2reg_test5	simple
65	mem2reg	mem2reg_test6	simple
66	mem_arst	MyMem	simple
67	memory	memtest00	simple
68	memory	memtest01	simple
69	memory	memtest03	simple
70	memory	memtest04	simple
71	memory	memtest05	simple
72	memory	memtest10	simple
73	memory	memtest11	simple
74	memory	memtest12	simple
75	memory	memtest13	simple
76	multiplier	Multiplier_flat	simple
77	multiplier	Multiplier_2D	simple

78	multiplier	FullAdder	simple
79	muxtree	usb_tx_phy	simple
80	muxtree	default_cases	simple
81	muxtree	select_leaves	simple
82	omsp_dbg_uart	omsp_dbg_uart	simple
83	paramods	pm_test1	simple
84	paramods	pm_test2	simple
85	paramods	pm_test3	simple
86	partsel	partsel_test001	simple
87	partsel	partsel_test002	simple
88	partsel	partsel_test003	simple
89	partsel	partsel_test004	simple
90	partsel	partsel_test005	simple
91	partsel	partsel_test006	simple
92	partsel	partsel_test007	simple
93	process	blocking_cond	simple
94	process	uut	simple
95	process	uart	simple
96	realexpr	demo_001	simple
97	realexpr	demo_004	simple
98	repwhile	repwhile_test001	simple
99	retime	retime_test	simple
100	rotate	a23_barrel_shift_fpga_rotate	simple
101	scopes	scopes_test_01	simple
102	signedexpr	signed_test01	simple
103	sincos	d	simple
104	subbytes	subbytes_00	simple
105	task_func	task_func_test03	simple
106	task_func	task_func_test04	simple

107	task_func	task_func_test05	simple
108	usb_phy_tests	usb_phy_test01	simple
109	values	test_signed	simple
110	values	test_const	simple
111	vloghammer	test01	simple
112	vloghammer	test02	simple
113	vloghammer	test03	simple
114	vloghammer	test04	simple
115	vloghammer	test07	simple
116	vloghammer	test08	simple
117	vloghammer	test09	simple
118	vloghammer	test10	simple
119	wreduce	wreduce_test0	simple
120	wreduce	wreduce_test1	simple
121	xilinxip	xilinxIP	single-design
122	counter_bram	counter_bram	single-design
123	aes_5cycle_2stage	aes_top	yosys-bigsim
124	bch_encode	bch_encode	yosys-bigsim
125	openmsp430	openMSP430	yosys-bigsim
126	reed_solomon_decoder	RS_dec	yosys-bigsim
127	softusb_navre	softusb_navre	yosys-bigsim
128	verilog-pong	top	yosys-bigsim
129	scripts	RAM256X8	yosys-bigsim
130	elliptic_curve_group	point_scalar_mult	yosys-bigsim
131	bgm	bgm	vtr-benchmarks
132	blob_merge	RLE_BlobMerging	vtr-benchmarks
133	boundtop	paj_boundtop_hierarchy_no_mem	vtr-benchmarks
134	raygentop	paj_raygentop_hierarc	vtr-benchmarks

		hy_no_mem	
135	ch_intrinsics	memset	vtr-benchmarks
136	diffeq1	diffeq_paj_convert	vtr-benchmarks
137	diffeq2	diffeq_f_systemC	vtr-benchmarks
138	mkPktMerge	mkPktMerge	vtr-benchmarks
139	LU8PEeng	LU8PEEng	vtr-benchmarks
140	stereovision0	sv_chip0_hierarchy_no_mem	vtr-benchmarks
141	stereovision1	sv_chip1_hierarchy_no_mem	vtr-benchmarks
142	stereovision2	sv_chip2_hierarchy_no_mem	vtr-benchmarks
143	stereovision3	sv_chip3_hierarchy_no_mem	vtr-benchmarks
144	or1200	or1200_flat	vtr-benchmarks
145	sha	sha1	vtr-benchmarks
146	forgen01	uut_forgen01	simple
147	cons_branch_finish	case_branch_finish_top	simple
148	hierdefparam	hierdefparam_top	simple
149	localparams_attr	uut_localparam_attr	simple
150	operators	optest	simple
151	param_attr	uut_param_attr	simple
152	string_format	string_format_top	simple
153	task_func_test01	task_func_test01	simple
154	task_func_test02	task_func_test02	simple
155	undef_eqx_nex	undef_eqx_nex	simple
156	wandwor	wandwor_test0	simple
157	wandwor	wandwor_test1	simple
158	top_bram_n1	top	single-design

159	top_bram_n2	top	single-design
160	top_bram_n3	top	single-design

Table A. 3. Benchmarks Used In Results Generation

Sl. No.	Design Name	Top Module	Benchmark Suite
1	aes_kexp128	aes_key_expand_128	simple
2	attrib01_module	foo	simple
3	attrib02_port_decl	foo	simple
4	attrib03_parameter	foo	simple
5	attrib04_net_var	foo	simple
6	attrib06_operator_suffix	foo	simple
7	attrib08_mod_inst	foo	simple
8	attrib09_case	foo	simple
9	dff_different_styles	dff	simple
10	dff_different_styles	dffa	simple
11	dff_different_styles	dffa1	simple
12	dff_different_styles	dffa2	simple
13	dff_different_styles	dffa4	simple
14	dff_different_styles	dffsr1	simple
15	dff_different_styles	dffsr2	simple
16	dff_init	dff0_test	simple
17	dff_init	dff1_test	simple
18	dff_init	dff0a_test	simple
19	dff_init	dff1a_test	simple
20	dff_init	dff_test_997	simple
21	dynslice	dynslice	simple
22	fiedler-cooley	up3down5	simple
24	forloops	forloops02	simple
25	fsm	fsm_test	simple
26	generate	gen_test6	simple

27	i2c_master_tests	i2c_test01	simple
28	i2c_master_tests	i2c_test02	simple
29	memory	memtest00	simple
30	memory	memtest01	simple
31	memory	memtest03	simple
32	memory	memtest04	simple
33	memory	memtest12	simple
34	memory	memtest13	simple
35	retime	retime_test	simple
36	subbytes	subbytes_00	simple
37	usb_phy_tests	usb_phy_test01	simple
38	aes_5cycle_2stage	aes_top	yosys-bigsim
39	bch_encode	bch_encode	yosys-bigsim
40	openmsp430	openMSP430	yosys-bigsim
41	reed_solomon_decoder	RS_dec	yosys-bigsim
42	softusb_navre	softusb_navre	yosys-bigsim
43	verilog-pong	top	yosys-bigsim
44	bgm	bgm	vtr-benchmarks
45	blob_merge	RLE_BlobMerging	vtr-benchmarks
46	ch_intrinsics	memtest	vtr-benchmarks
47	mkPktMerge	mkPktMerge	vtr-benchmarks
48	stereovision0	sv_chip0_hierarchy_no_mem	vtr-benchmarks
49	stereovision1	sv_chip1_hierarchy_no_mem	vtr-benchmarks
50	stereovision2	sv_chip2_hierarchy_no_mem	vtr-benchmarks
51	stereovision3	sv_chip3_hierarchy_no_mem	vtr-benchmarks
52	sha	sha1	vtr-benchmarks

53	diffeq1	diffeq_paj_convert	vtr-benchmarks
54	diffeq2	diffeq_f_systemC	vtr-benchmarks
55	xilinxip	xilinxIP	single-design

Table A. 4. Benchmarks That Pass Equivalence Checking Through Yosys in SymbiFlow

Sl. No.	**Design Name**	**Top Module**	**Benchmark Suite**
1	aes_kexp128	aes_key_expand_128	simple
2	always01	uut_always01	simple
3	always02	uut_always02	simple
4	always03	uut_always03	simple
5	arraycells	array_test001	simple
6	arraycells	aoi12	simple
7	arrays01	uut_arrays01	simple
8	attrib01_module	bar	simple
9	attrib01_module	foo	simple
10	attrib02_port_decl	bar	simple
11	attrib02_port_decl	foo	simple
12	attrib03_parameter	foo	simple
13	attrib04_net_var	bar	simple
14	attrib04_net_var	foo	simple
15	attrib06_operator_suffix	bar	simple
16	attrib06_operator_suffix	foo	simple
17	attrib08_mod_inst	bar	simple
18	attrib08_mod_inst	foo	simple
19	attrib09_case	bar	simple

20	attrib09_case	foo	simple
21	carryadd	carryadd	simple
22	constmuldivmod	constmuldivmod	simple
23	dff_different_styles	dff	simple
24	dff_different_styles	dffa	simple
25	dff_different_styles	dffa1	simple
26	dff_different_styles	dffa2	simple
27	dff_different_styles	dffa4	simple
28	dff_different_styles	dffsr1	simple
29	dff_init	dff0_test	simple
30	dff_init	dff1_test	simple
31	dff_init	dff0a_test	simple
32	dff_init	dff1a_test	simple
33	dff_init	dff_test_997	simple
34	dynslice	dynslice	simple
35	fiedler-cooley	up3down5	simple
36	forgen02	uut_forgen02	simple
37	forloops	forloops01	simple
38	forloops	forloops02	simple
39	fsm	fsm_test	simple
40	generate	gen_test1	simple
41	generate	gen_test2	simple
42	generate	gen_test3	simple
43	generate	gen_test5	simple
44	generate	gen_test6	simple
45	graphtest	graphtest	simple
46	hierarchy	top	simple
47	hierdefparam	hierdefparam_a	simple
48	hierdefparam	hierdefparam_b	simple
49	i2c_master_tests	i2c_test01	simple

50	i2c_master_tests	i2c_test02	simple
51	implicit_ports	alu	simple
52	implicit_ports	named_ports	simple
53	loops	aes	simple
54	macros	test_def	simple
55	macros	test_ifdef	simple
56	macros	test_comment_in_macro	simple
57	mem2reg	mem2reg_test1	simple
58	mem2reg	mem2reg_test2	simple
59	mem2reg	mem2reg_test3	simple
60	mem2reg	mem2reg_test4	simple
61	mem2reg	mem2reg_test5	simple
62	mem2reg	mem2reg_test6	simple
63	mem_arst	MyMem	simple
64	memory	memtest00	simple
65	memory	memtest01	simple
66	memory	memtest03	simple
67	memory	memtest04	simple
68	memory	memtest05	simple
69	memory	memtest10	simple
70	memory	memtest11	simple
71	memory	memtest12	simple
72	memory	memtest13	simple
73	multiplier	Multiplier_flat	simple
74	multiplier	Multiplier_2D	simple
75	multiplier	FullAdder	simple
76	muxtree	usb_tx_phy	simple
77	muxtree	default_cases	simple
78	muxtree	select_leaves	simple

79	omsp_dbg_uart	omsp_dbg_uart	simple
80	paramods	pm_test1	simple
81	paramods	pm_test2	simple
82	paramods	pm_test3	simple
83	partsel	partsel_test001	simple
84	partsel	partsel_test002	simple
85	partsel	partsel_test003	simple
86	partsel	partsel_test004	simple
87	partsel	partsel_test005	simple
88	partsel	partsel_test006	simple
89	partsel	partsel_test007	simple
90	process	blocking_cond	simple
91	process	uut	simple
92	process	uart	simple
93	realexpr	demo_001	simple
94	realexpr	demo_004	simple
95	retime	retime_test	simple
96	rotate	a23_barrel_shift_fpga_rotate	simple
97	scopes	scopes_test_01	simple
98	signedexpr	signed_test01	simple
99	sincos	d	simple
100	subbytes	subbytes_00	simple
101	task_func	task_func_test03	simple
102	task_func	task_func_test04	simple
103	usb_phy_tests	usb_phy_test01	simple
104	values	test_signed	simple
105	values	test_const	simple
106	vloghammer	test01	simple
107	vloghammer	test02	simple

108	vloghammer	test03	simple
109	vloghammer	test04	simple
110	vloghammer	test07	simple
111	vloghammer	test08	simple
112	vloghammer	test09	simple
113	vloghammer	test10	simple
114	wreduce	wreduce_test0	simple
115	wreduce	wreduce_test1	simple
116	xilinxip	xilinxIP	single-design
117	counter_bram	counter_bram	single-design
118	aes_5cycle_2stage	aes_top	yosys-bigsim
119	bch_encode	bch_encode	yosys-bigsim
120	softusb_navre	softusb_navre	yosys-bigsim
121	blob_merge	RLE_BlobMerging	vtr-benchmarks
122	diffeq2	diffeq_f_systemC	vtr-benchmarks
123	sha	sha1	vtr-benchmarks

Table A. 5. Designs that generate bitstreams in SymbiFlow

Sl. No.	**Design Name**	**Top Module**	**Benchmark Suite**
1	aes_kexp128	aes_key_expand_128	simple
2	always01	uut_always01	simple
3	always02	uut_always02	simple
4	always03	uut_always03	simple
5	arraycells	array_test001	simple
6	arraycells	aoi12	simple
7	arrays01	uut_arrays01	simple
8	attrib01_module	bar	simple
9	attrib01_module	foo	simple
10	attrib02_port_decl	bar	simple
11	attrib02_port_decl	foo	simple

12	attrib03_parameter	foo	simple
13	attrib04_net_var	bar	simple
14	attrib04_net_var	foo	simple
15	attrib06_operator_suffix	bar	simple
16	attrib06_operator_suffix	foo	simple
17	attrib08_mod_inst	bar	simple
18	attrib08_mod_inst	foo	simple
19	attrib09_case	bar	simple
20	attrib09_case	foo	simple
21	carryadd	carryadd	simple
22	constmuldivmod	constmuldivmod	simple
23	dff_different_styles	dff	simple
24	dff_different_styles	dffa	simple
25	dff_different_styles	dffa1	simple
26	dff_different_styles	dffa2	simple
27	dff_different_styles	dffa4	simple
28	dff_different_styles	dffsr1	simple
29	dff_different_styles	dffsr2	simple
30	dff_different_styles	dffsr2_sub	simple
31	dff_init	dff0_test	simple
32	dff_init	dff1_test	simple
33	dff_init	dff0a_test	simple
34	dff_init	dff1a_test	simple
35	dff_init	dff_test_997	simple
36	dynslice	dynslice	simple
37	fiedler-cooley	up3down5	simple
38	forgen02	uut_forgen02	simple
39	forloops	forloops01	simple
40	forloops	forloops02	simple
41	fsm	fsm_test	simple

42	generate	gen_test1	simple
43	generate	gen_test2	simple
44	generate	gen_test3	simple
45	generate	gen_test4	simple
46	generate	gen_test6	simple
47	graphtest	graphtest	simple
48	hierarchy	top	simple
49	hierdefparam	hierdefparam_a	simple
50	hierdefparam	hierdefparam_b	simple
51	i2c_master_tests	i2c_test01	simple
52	i2c_master_tests	i2c_test02	simple
53	implicit_ports	alu	simple
54	implicit_ports	named_ports	simple
55	loops	aes	simple
56	macros	test_def	simple
57	macros	test_ifdef	simple
58	macros	test_comment_in_macro	simple
59	mem2reg	mem2reg_test1	simple
60	mem2reg	mem2reg_test2	simple
61	mem2reg	mem2reg_test3	simple
62	mem2reg	mem2reg_test4	simple
63	mem2reg	mem2reg_test5	simple
64	mem2reg	mem2reg_test6	simple
65	mem_arst	MyMem	simple
66	memory	memtest00	simple
67	memory	memtest01	simple
68	memory	memtest03	simple
69	memory	memtest04	simple
70	memory	memtest05	simple
71	memory	memtest10	simple

72	memory	memtest11	simple
73	memory	memtest12	simple
74	memory	memtest13	simple
75	multiplier	Multiplier_flat	simple
76	multiplier	Multiplier_2D	simple
77	multiplier	FullAdder	simple
78	muxtree	usb_tx_phy	simple
79	muxtree	default_cases	simple
80	muxtree	select_leaves	simple
81	omsp_dbg_uart	omsp_dbg_uart	simple
82	paramods	pm_test1	simple
83	paramods	pm_test2	simple
84	paramods	pm_test3	simple
85	partsel	partsel_test001	simple
86	partsel	partsel_test003	simple
87	partsel	partsel_test004	simple
88	partsel	partsel_test005	simple
89	partsel	partsel_test006	simple
90	partsel	partsel_test007	simple
91	process	blocking_cond	simple
92	process	uut	simple
93	process	uart	simple
94	realexpr	demo_001	simple
95	realexpr	demo_004	simple
96	repwhile	repwhile_test001	simple
97	retime	retime_test	simple
98	rotate	a23_barrel_shift_fpga_rotate	simple
99	scopes	scopes_test_01	simple
100	signedexpr	signed_test01	simple

101	sincos	d	simple
102	subbytes	subbytes_00	simple
103	task_func	task_func_test03	simple
104	task_func	task_func_test04	simple
105	task_func	task_func_test05	simple
106	usb_phy_tests	usb_phy_test01	simple
107	values	test_signed	simple
108	values	test_const	simple
109	vloghammer	test01	simple
110	vloghammer	test02	simple
111	vloghammer	test03	simple
112	vloghammer	test04	simple
113	vloghammer	test07	simple
114	vloghammer	test08	simple
115	vloghammer	test09	simple
116	vloghammer	test10	simple
117	wreduce	wreduce_test0	simple
118	wreduce	wreduce_test1	simple
119	xilinxip	xilinxIP	single-design
120	counter_bram	counter_bram	single-design
121	bch_encode	bch_encode	yosys-bigsim
122	reed_solomon_decoder	RS_dec	yosys-bigsim
123	softusb_navre	softusb_navre	yosys-bigsim
124	verilog-pong	top	yosys-bigsim
125	scripts	RAM256X8	yosys-bigsim
126	bgm	bgm	vtr-benchmarks
127	blob_merge	RLE_BlobMerging	vtr-benchmarks
128	boundtop	paj_boundtop_hierarchy_no_mem	vtr-benchmarks
129	ch_intrinsics	memset	vtr-benchmarks

130	diffeq1	diffeq_paj_convert	vtr-benchmarks
131	diffeq2	diffeq_f_systemC	vtr-benchmarks
132	stereovision3	sv_chip3_hierarchy_no_mem	vtr-benchmarks
133	sha	sha1	vtr-benchmarks

BIBLIOGRAPHY

[1] C. Wolf, Yosys Open SYnthesis Suite, http://www.clifford.at/yosys/, 2020. https://github.com/YosysHQ/yosys-bench

[2] C. Wolf, Yosys Manual, http://www.clifford.at/yosys/files/yosys_manual.pdf, 2020.

[3] Cadence Encounter Conformal. Cadence Design Systems. Available: https://www.cadence.com/en_US/home/tools/digital-design-and signoff/conformal-overview.html

[4] Conformal Equivalence Checking Command Reference, Cadence Design Systems, Product version 21.1, May 2021

[5] Conformal Equivalence Checking User Guide, Cadence Design Systems, Product version 21.1, May 2021

[6] SymbiFlow open-source FPGA tooling Available: https://SymbiFlow.github.io/

[7] Xilinx Vivado. Xilinx Corporation. Available: https://www.xilinx.com/products/design-tools/vivado.html

[8] Xilinx Vivado – Design Flows Overview. Xilinx Corporation. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug892-vivado-design-flows-overview.pdf

[9] Versatile Place and Route. Available: https://docs.verilogtorouting.org/en/latest/vpr/

[10] ABC: A System for Sequential Synthesis and Verification, https://people.eecs.berkeley.edu/~alanmi/abc/

[11] Arty 35T website, Digilent. Available : https://digilent.com/reference/programmable-logic/arty-a7/start

[12] Xilinx, 7 Series FPGAs Data Sheet Overview. Available: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[13] Project X-Ray. Available: https://SymbiFlow.readthedocs.io/projects/prjxray/en/latest/index.html

[14] Yosys-simple benchmarks Available: https://github.com/YosysHQ/yosys-bench

[15] Yosys-bigsim benchmarks Available: https://github.com/YosysHQ/yosys-bigsim

[16] VTR-benchmarks documentation page Available: http://docs.verilogtorouting.org/en/latest/vtr/benchmarks/

[17] Herklotz, Y., Wickerson, J. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20)* (2020). pp. 277–287. DOI:https://doi.org/10.1145/3373087.3375310.

[18] M. Girish, G. Gopakumar and D. S. Divya, "Formal and Simulation Verification: Comparing and Contrasting the two Verification Approaches," *2021 2nd International Conference on Advances in Computing*, Communication, Embedded and Secure Systems (ACCESS), 2021, pp. 41-44, doi: 10.1109/ACCESS51619.2021.9563305.

[19] D. Shah, E. Hung, C. Wolf, Serge Bazanaski, "Yosys+nextpnr: an Open Source Framework from Verilog to Bitstream for Commercial FPGAs," *IEEE FCCM 2019*, 2019.

[20] Grimm T, Lettnin D, Hübner M, "A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip," *Electronics*. 2018; 7(6):81. https://doi.org/10.3390/electronics7060081

[21] Wolf, Clifford, Johann Glaser and Johannes Kepler. "Yosys-A Free Verilog Synthesis Suite." (2013).

[22] M. Aagaard and M. Leeser, "A formally verified system for logic synthesis," *[1991 Proceedings] IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1991, pp. 346-350, doi: 10.1109/ICCD.1991.139915.

[23] M. Girish, G. Gopakumar and D. S. Divya, "Formal and Simulation Verification: Comparing and Contrasting the two Verification Approaches," *2021 2nd International Conference on Advances in Computing, Communication, Embedded and Secure Systems (ACCESS)*, 2021, pp. 41-44, doi: 10.1109/ACCESS51619.2021.9563305.

[24] Syme, Don & Granicz, Adam & Cisternino, Antonio, "Lexing and Parsing," 2017, 10.1007/978-1-4302-0285-1_16.

[25] Yan Zhang, Xinyu Gao, Ce Bian, Ding Ma and Baojiang Cui, "Homologous detection based on text, Token and abstract syntax tree comparison," *2010 IEEE International Conference on Information Theory and Information Security*, 2010, pp. 70-75, doi: 10.1109/ICITIS.2010.5689624.

[26] L. Titarenko, O. Hebda and A. Barkalov, "Design of Moore finite state machine with coding space stretching," *2014 7th International Conference on Human System Interactions (HSI)*, 2014, pp. 238-242, doi: 10.1109/HSI.2014.6860482.

[27] Xilinx Vivado Design Suite 7 Series FPGA Libraries Guide Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_2/ug953-vivado-7series-libraries.pdf

[28] Synopsis Equivalence Checking Available: https://www.synopsys.com/glossary/what-is-equivalence-checking.html

[29] V. N. Possani, A. Mishchenko, R. P. Ribas and A. I. Reis, "Parallel Combinational Equivalence Checking," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 3081-3092, Oct. 2020, doi: 10.1109/TCAD.2019.2946254.

[30] Genus Synthesis Solution. Cadence Design Systems. Available : https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

[31] Precision FPGA Synthesis. Siemens EDA Software. Available : https://eda.sw.siemens.com/en-US/ic/precision/

[32] Verismith., Y. Herklotz, J. Wickerson. Imperial College London. Available: https://github.com/ymherklotz/verismith

[33] Lattice Semiconductor. Available: https://www.latticesemi.com/

[34] Xilinx Artix-7 FPGAs. Available : https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html

[35] Yosys-SMTBMC. Available : https://github.com/YosysHQ/yosys/blob/master/backends/smt2/smtbmc.py

[36] Xilinx Vivado. Xilinx Corporation. Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[37] 7 Series DSP48E1 Slice User Guide (UG479) – Xilinx, Available: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[38] Intel Quartus Prime Software Suite. Available : https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html

[39] PicoRV32: A Size-Optimized RISC-V CPU. Available: https://github.com/YosysHQ/picorv32

[40] Z3 Theorem Prover. Available : https://github.com/Z3Prover/z3

[41] Satisfiability Modulo Theories. Available : https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

[42] Yices SMT Solver. Available: https://yices.csl.sri.com/

[43] SymbiYosys. Available: https://symbiyosys.readthedocs.io/en/latest/index.html

[44] Xilinx Artix-7 FPGA AC701 Evaluation Kit. Available: https://www.xilinx.com/products/boards-and-kits/ek-a7-ac701-g.html

[45] 7 Series FPGAs Memory Resources User Guide – Xilinx. Available: https://docs.xilinx.com/v/u/en-US/ug473_7Series_Memory_Resources

[46] Nexys Video Artix-7 FPGA: Trainer Board for Multimedia Applications. Available: https://digilent.com/shop/nexys-video-artix-7-fpga-trainer-board-for-multimedia-applications/

[47] Using SystemVerilog Assertions in RTL. Available: https://www.design-reuse.com/articles/10907/using-systemverilog-assertions-in-rtl-code.html

[48] M. Rathmair, F. Schupfer and C. Krieg, "Applied formal methods for hardware Trojan detection," *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 169-172, doi: 10.1109/ISCAS.2014.6865092.

[49] IEEE Computer Society, "IEEE Standard for Verilog Hardware Description Language", *IEEE 1364-2005*, 2006.