Verifying the Correctness of FPGA Logic Synthesis Algorithms

Boris Ratchev, Mike Hutton, Gregg Baeckler and Babette van Antwerpen

Altera Corporation, 101 Innovation Drive, San Jose, CA 95134 (bratchev,mhutton@altera.com)

ABSTRACT

Though verification is significantly easier for FPGA-based digital systems than for ASIC or full-custom hardware, there are nonetheless many places for errors to occur.

In this paper we discuss the verification problem for FPGAs and describe several methods for verifying end-to-end correctness of synthesis algorithms, a particularly complex portion of the CAD flow. Though the primary contribution of this paper is the analysis of the overall problem, we also give an algorithm for the automatic generation of test-vectors for simulation using information from the synthesis tool, and describe a second testing method that generates purposefully difficult designs in combination with input vectors to test them. We will show the validity of these methods by standard metrics such as simulation node-coverage and through the ability for the method to locate forced errors introduced by the synthesis tool.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance. D.2.5 [**Software Engineering**]: Testing and Debugging –*Statistical Methods, Validation.*

General Terms

Algorithms, Verification

Keywords

Programmable logic, FPGA, synthesis, verification, test.

1. INTRODUCTION

One of the key advantages of FPGA-based systems is the ability for a hardware designer to very quickly generate functional hardware. Though this is often stated in terms of the high cost of mask generation and fabrication NREs, these costs are increasingly dominated by the test and verification problem.

Testing includes checking for several criteria: that the HDL code meets specification, that the CAD tools correctly processed the code, and that the eventual chip functions properly, including timing.

Figure 1 shows a typical CAD flow, highlighting the verification steps. In an FPGA-based design there is no chip-test required by the designer, because verification of the programmable logic fabric is performed by the FPGA vendor.

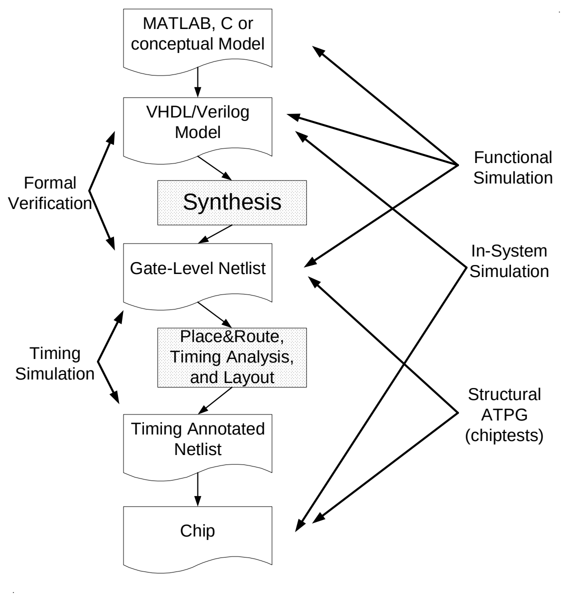

Figure 1. Verification steps.

Vendors test each chip for fabrication defects with manually and automatically generated deterministic test vectors, and then bin individual chips based on their performance. A great deal of effort can be put into the former problem, because the cost of test generation is amortized across the thousands of different hardware designs which eventually use the device. To some extent the vendor also solves the timing simulation problem because the binning of devices into speed-grades gives increased confidence to the user that functional operation will match in-system timing.

The efforts put into test generation for a specific design in the ASIC flow have a secondary benefit: they test the correctness of complicated algorithms in synthesis, placement, routing and timing analysis in the CAD tools. Chip testing by FPGA vendors guarantees the correctness of the FPGA hardware, but does not also test the design or synthesis of the design being implemented on it.

As a software problem, synthesis algorithms are tested by many standard methods, including hand-created regression tests, design and code reviews, proofs of algorithmic correctness to name a few. However, it is always advantageous to supplement these with end-to-end testing on large designs to quickly locate errors, and this is especially true during the early debugging phase.

In this paper we will concentrate on perhaps the most difficult aspect of the CAD flow to test, namely the synthesis of hardware models into gate-level netlists. Synthesis is harder to verify because relatively simple errors, such as inverting one signal, maintain the connectivity and timing paths in the gate-level netlist. Errors in connectivity can generally be identified deterministically whereas functional equivalence of Boolean logic is NP-hard.

The compounding problem from the CAD designer's point of view is more pragmatic. Software engineers working on tools for logic synthesis are almost never the same people creating the hardware design. Most complete example designs are large complicated systems provided as-is through problem reporting or to maintain customer-supplier relationships. CAD designers rarely receive simulation test-vectors or enough information about the design to create them manually, and designs are often incomplete or insufficiently tested at the behavioral level. One notable exception is FPGA vendors internally generated IP, which is a great source of high-quality finished test designs with well-designed vectors and often complicated functionality. Further problems with testing on FPGAs include non-synthesizable hardware such as embedded RAM and dedicated DSP (multiplier) hardware, and complicated registers with multiple clock-domains.

The testing goals of hardware and software designers are also different. A hardware designer is interested in testing one design over the course of its development cycle of 6 months or more and can put significant effort into test generation, formal verification, and even develop their hardware models with test in mind (BIST and design-for-test). Tool designers want to test their tools on hundreds of different designs quickly and easily and have no influence on the design of the original hardware.

Our goal is to supplement standard software testing with tools to automatically detect incorrect logic introduced by synthesis algorithms. The standard methods for ensuring software quality involve regression tests, algorithm and code reviews, assertions and cross-check code and multiple other forms of independent testing. These techniques can be aided by end-to-end testing of large designs. Specifically, we want to analyze the correctness of the entire synthesis system through algorithmic changes, including advanced sequential synthesis algorithms such as retiming and other manipulation of registers.

Automatic testing serves a further purpose to speed the debugging process, since many of the formal testing metrics are not yet in place during the development of an algorithm. However in a mature software system the previous version of the software is a trustworthy comparison point. Furthermore in the case of synthesis the correctness problem is well-defined.

Section 2 of this paper provides further background and outlines some existing approaches. Section 3 introduces our method for solving the problem, and how the resulting tool RVEC fits into the verification flow. Section 4 shows some empirical results on test-coverage and error detection. Section 5 outlines issues with false positives; most are solved in our current implementation but some remain open. Section 6 describes efforts to generate particularly difficult sequential designs in concert with input vectors as an additional testing method. We conclude in Section 7.

The primary contributions of this paper are in the discussion and analysis of the overall problem as much as in the proposal of a specific algorithm.

2. PREVIOUS WORK

There are a number of existing approaches for testing digital logic that are related to this problem. These include the standard ATPG testability theory, formal verification, and random vector generation.

2.1 Structural Testing

Structural testing techniques model errors in a combinational design in terms of fault models (stuck-at-0 or stuck-at-1) (e.g. [4], [1]). These techniques use the logical functionality of the gate-level netlist to create tests for simulation that sensitize each wire in the netlist. Values required to test a fault on one wire are propagated backwards through gates to the primary inputs using the functionality of the gates in the design. Testing of sequential circuits require unwrapping the netlist to the combinational model. Since the techniques are largely for generating vectors for chip-test, one of the important optimization criteria is the shortness of the test-vectors, since time on test machines is expensive.

Brand [2] applied the use of testing theory directly to our problem at hand, namely verifying the equivalence of successive stages of synthesis. He gave an $O(n^2)$ time and space algorithm for combinational circuits, and successfully verified the set of ISCAS circuits through several basic synthesis algorithms.

In practice these techniques have several drawbacks. The netlists we are considering can have over 100,000 logic elements, and the computational complexity of deterministic ATPG is too high – Rudnick and Patel [12] show empirical results on relatively small MCNC and ISCAS circuits taking upwards of 4 hours on simple circuits, and 80 hours when re-timing operations are involved. Though such time could be feasible in the context of generating a chip-test, it is not feasible for the problem of running hundreds of designs to verify changes to synthesis tools.

2.2 Formal Verification

Formal Verification techniques attempt to heuristically prove the sequential equivalence of two netlists using, for example, structural partitioning followed by BDD-based analysis. We attempted to use several commercially available formal verification tools and found that they can generate some impressive results for a designer interested in verifying a single circuit. However, to avoid the vast number of false errors, the user is required to either restrict the operation of synthesis to "verification safe" operations (similar to forbidding compiler optimizations and instruction re-ordering to ensure ease of debugging in a software debugger), or annotate the netlist with many properties of the synthesis operations. These may be reasonable for the hardware designer, but it is not feasible for the algorithmic testing problem. Commercial tools also have issues with re-timing, multiple clock domains and non-synthesizable hardware, requiring design-specific work in order to use the tools.

2.3 Random Vector Generation

The use of simulation with random vectors is more attractive than the previous two from a practical point of view. The concept is simple: randomly perturb the input vectors for the design over a period of time, observe the resulting output vectors, and then simulate the same test vectors on the modified netlist and compare the resulting output vectors.

The overwhelming advantage of using random vectors is simplicity, most importantly the ability to deal with complex hardware such as RAM and DSP blocks in simulation, but not as part of test generation. The disadvantage is the lack of complete coverage of the netlist.

There have been multiple attempts to improve the use of randomness in testing. Rudnick and Patel [12] give a hybrid algorithm to speed up structural test generation with techniques from genetic algorithms. The textbook by Mazumder and Rudnick [10] discusses multiple such approaches using genetic algorithms.

The primary issues we found with random vector generation are that naïve methods of vector generation "don't work" – to get any reasonable amount of simulation coverage, we found that more intelligent generation to deal with glitching, secondary signals, power-up state and latches was required.

3. RVEC VECTOR GENERATION

3.1 Verification Flow

RVEC is a C++ standalone executable that produces random vectors for simulation stimulus for designs compiled by Altera's Quartus II software. Though Quartus is a relatively large tool implementing the entire FPGA CAD flow, we are primarily concerned with the synthesis and simulation subsystems, which we will denote QSYN and QSIM. (Note that these terms are used to aid this presentation, and don't correspond to terms used in Quartus documentation or help). QSYN takes as input VHDL, Verilog, EDIF, Altera AHDL or schematic designs, and outputs a gate-level netlist. QSIM takes a gate or logic-element level netlist and a vector file, and provides either the output-vectors from simulation, or (if they are included in the input file), pass or fail.

The flow for generating golden vectors with RVEC is shown in Figure 2. A design is compiled with QSYN to produce the file qsyn.info, which is used by RVEC to determine information about the design. RVEC takes the qsyn.info file and a parameterization file params.txt to output the vector file, which is then used by QSIM to simulate the design and generate the "golden" output vectors. The golden vector outputs are used to verify modifications to QSYN, as shown in Figure 3.

The QSIM module currently implemented in Quartus does not yet have the ability to perform behavioral simulation with a testbench. However, it is feasible to implement this flow as well, using 3rd party simulation tools such as ModelSim. In this case a behavioral model of embedded RAM and any other dedicated hardware blocks would be required—these models are already provided by Altera Corp. This simulation approach provides the additional benefits of comparing the user's HDL directly to the end-result of synthesis.

3.2 Program Description

This section discusses the input and output files for vector generation. The two input files are qsyn.info and params.txt. The output file is the set of vectors generated by RVEC

Input signals in a design cannot be treated the same. A global reset signal, for example, could force all flip-flops in the design to 0. When this signal is active the netlist is largely untested. Similarly the values seen at pins will usually only change with the

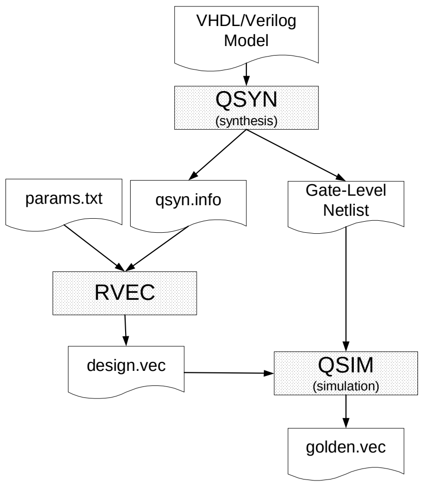

Figure 2. RVEC flow for generating golden vectors

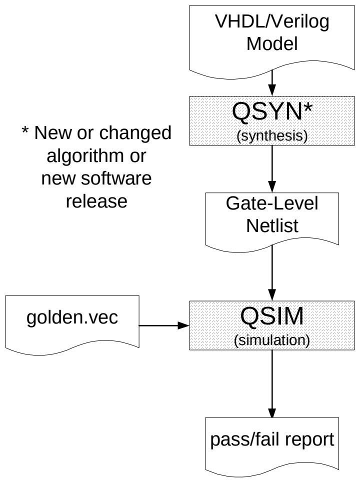

Figure 3. RVEC flow for verifying new synthesis algorithm.

controlling clock, so a random vector pattern which keeps the clock high or low for long periods of time is similarly inefficient. The generator must be aware of bidir (tristate) pins and their enables in order to separate out input from output functionality. Gated clocks (modified by user logic, e.g. to halve the frequency with a counter), though poor design practice in FPGAs, are nonetheless common.

Thus, we require some information from QSYN to effectively generate patterns for these special-case signals. The most important information is the identification of input, output and bidir pins, all clocks in the design, asynchronous secondary signals (clear, asynchronous load, clock-enable, output enable) and an indication whether they should be "mostly high" or "mostly low". This latter information is a function of the Altera-specific hardware-–an active low reset would be an example of a mostly high signal, since usually the reset is not being exercised. An output enable would be neither mostly high nor mostly low, but would also not be expected to toggle as often as a typical input.

An example qsyn.info file identifying a design with 4 data pins is shown in Figure 4. Two clocks and three asynchronous signals are identified, along with the active polarity of the secondary signals.

Most designs have several asynchronous signals in each of the high and low polarities, and a number of different clock domains.

Additional input to RVEC is contained in params.txt; an example is shown in Figure 5. This file contains general parameterization information such as the types of clocks to generate, the setup and hold times necessary to make the simulation work properly and so on. Much of the contents of params.txt could be considered default values to the algorithm; it is just more convenient to expose it to experimentation. Params.txt allows us to change information on parameters such as clock duty cycle, but we haven't put much effort into evaluating this parameter to-date. In the example above we are asking for 100 cycles, with a frequency of 1Hz, duty cycle of 0.1 and register tsu=0.5 ms.

The output of this small example is shown in Figure 6. The waveform shows four types of signals.

- The clocks are at the top and change as specified in params.txt.
- The asynchronous-high signal, in this naming convention, is high most of the time and gets reset once in a while. The resetting happens deterministically every total_cycles/10 cycles.
- The asynchronous-low signals have the mirrored behavior to that of asynchronous-high ones.
- The asynchronous-both signals are next (two of them). They are allowed to randomly change every total_cycles/5 times.
- The regular inputs are next—four of them. They are allowed to randomly change every clock cycle. One of the inputs is a bidir pin. This is why the waveform editor produces a bidir~result pin for the output of the pin. Also RVEC uses L and H instead of 0 and 1 for bidirs to prevent contention, though these details would be tool-specific.
- Finally we have the outputs with X values since there is no simulation results in this file.

Example: For a 1000 cycle vector, inputs get a chance to change every cycle, asynchronous-both have a chance to change every 100 cycles asynchronous-high or low pins get reset (not random) every 200 cycles.

One fact that is difficult to see from the picture is that no two signals change at the same time. This is done to prevent race conditions other unpredictable simulation behavior (e.g. the reset and clock signals reaching a register at the same time)—see Section 5.

3.3 Algorithm

The algorithm for generating the random vectors involves manipulating a hashtable with each vector's time as the key. Below we explain the terminology and the procedure for obtaining the vectors.

A vector is defined to be a set of values for each input signal in the design at a discrete point in time.

```
INPUTS clk1 clk2 asy_hi1 asy_both1 asy_both2  asy_lo1 in1 in2
in3 bidir ;

CLOCKS clk1 clk2;

ASYNCH_MOSTLY_HI asy_hi1 ;

ASYNCH_MOSTLY_LO asy_lo1 ;

ANY_ASYNCH asy_both1 asy_both2 ;

OUTPUTS out1 out2 out3 bidir ;
```

Figure 4. Sample QSYN.INFO file

```
FREQ 1

# The setup time (ms) we want to use. All inputs change one or
# more tsu steps before the clock's rising edge.

TSU 0.5

# The clock's duty cycle.

DUTY_CYC 0.1

# length of the random vector--it is MAX_CYCLES * Period ns

MAX_CYCLES 100
```

Figure 5. Sample PARAMS.TXT

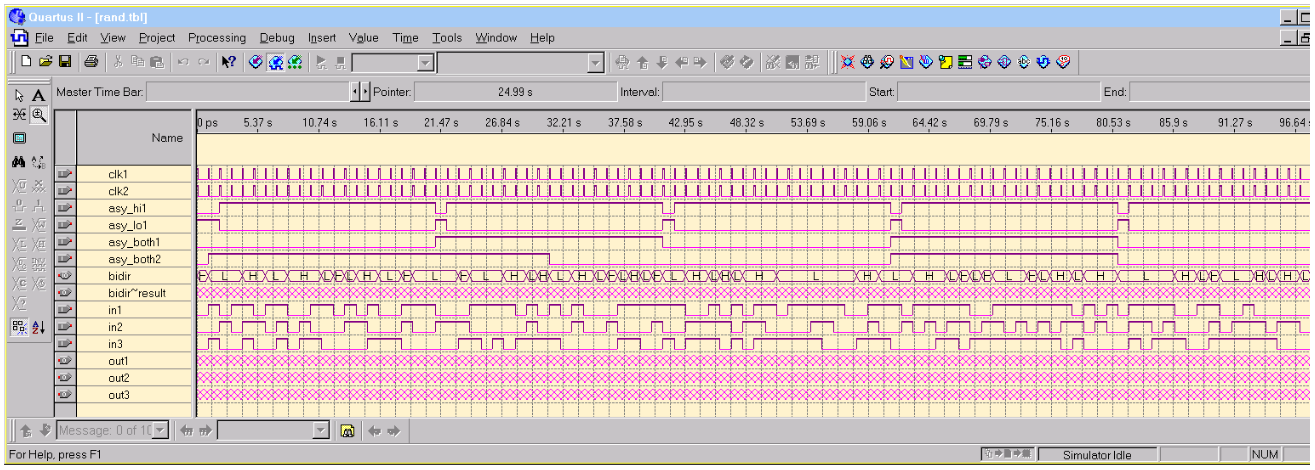

Figure 6. Resulting output of RVEC as viewed in QSIM.

An example of three consecutive vectors is:

```
102.5> 0 0 1
102.9> 1 0 1
105.0> 1 1 1
```

This example specifies that, at time 102.5 in the simulation, the three input pins change to 0, 0, and 1, respectively, independent of their previous values. They then change to 101 and 111 at times 102.9 and 105.0. The hashtable containing the above vectors would have three entries with the vector times as the keys and an array of the signal values (0 and 1) as the table entry.

The algorithm works as follows: For each clock cycle:

1. Insert clock edges (vectors): Since the clocks are what determine the allowed times for all other transitions, we insert values for each clock edge (rising and falling) for each clock (there could be any number of clocks). This must be done in a way that shifts each clock by a tsu with respect to other clocks so that there are no two edges changing at the same time.
2. Insert vectors (edges) for asynchronous signals. Again care must be taken that no two edges occur at the same time. Also the reset signals are at their non-active value for at least one high and one low clock transition (to ensure proper resetting of the circuit).
3. Insert vectors for regular inputs. Similarly to the asynchronous signals we calculate the number of edges based on the total number of such signals. For example if we have 10 inputs there will be at most 10 edges before each clock rising edge. At most because the edge occurs with probability of 50%.

The above three procedures are repeated for each cycle to obtain the whole simulation stimulus file. In the interest of saving space we skip some details having to do with how one ensures that only one signal changes at a time and that there is continuity in the vectors—i.e. no undefined values exist, etc. Those details are implementation specific and not difficult to work out.

The above algorithm produces vectors that have no two signals switching at the same time. This is very important for preventing false failures during verification. See Section 5 for more detail on why signals cannot change simultaneously.

Some further implementation details:

- All clocks have the same frequency since we have no way of knowing how fast the user meant to run them. In future versions of the tool we would like to represent related clocks, at least those generated by simple multiply and divide relationships, or generated by phase-locked loops.
- Keep the clock duty cycle low (e.g. 10-30%) This makes it easier for RVEC to keep track of what signal changes go with what tick. Since the frequency used is very slow, the clock will be high for long enough.
- The random generator has a fixed seed for now so that every time you run RVEC on the same design you will get the same vector (provided the parameters in params.txt are not changed). This allows consistent results during experimentation. Further, it facilitates tracing of simulation mismatches.

4. RESULTS AND METRICS OF SUCCESS

4.1 Simulation Coverage

The effectiveness of simulation vectors is often evaluated by the percentage of nodes that transition at least once during the simulation. Note that even 100% coverage does not mean 100% chance of finding an error.

For the statistics reported here we are using more than 150 designs with a median size of 5000 LEs (4-LUT+FF). They are drawn from a number of sources, including complete industrial (customer) designs, internally generated IP and specific testing designs.

The first several attempts at vector generation with RVEC generated very poor simulation coverage, in the order of 10-20% of nodes. Modifications to correctly deal with setup and hold on clocks brought this up further, and the correct handling of asynchronous signals and latches gained an additional 10%. The final results of RVEC as described in Section 3 have 67% node coverage on smaller designs and 40% on larger designs. The overall simulation coverage is 52%.

The simulation coverage is better viewed as a histogram. Figure 7, ignoring the split bars for now, shows a histogram of the number of designs in each coverage bin.

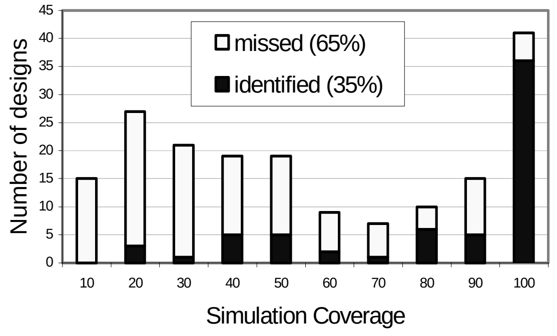

Figure 7. Histogram of simulation coverage, and breakdown into identified vs. missed errors using RVEC.

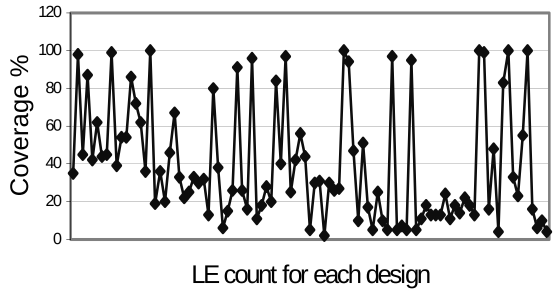

Figure 8. Coverage partly correlated to design-size.

These coverage metrics are determined using a relatively short simulation time, only 1000 clock cycles. In general, generating longer series of vectors will increase coverage. However, as will be seen shortly, it really isn't necessary to do so, because we are able to locate errors in the design with relatively high confidence even with very short simulations.

Some of the design-specific reasons for low coverage, such as latches and gated clocks are discussed in Section 5. Though we get a better coverage for smaller than larger designs in a bipartition of the design-set, and there is a mild correlation between design-size and simulation coverage with vectors of length 1000, it is really the more complicated nature of the larger designs that is determining coverage. This is illustrated in Figure 8, which shows the simulation coverage for the larger half of designs. Designs are sorted from smallest to largest LE count and plotted against their coverage. Though there is a definite downward trend in coverage it does not apply to all designs. Well-written synchronous designs with no combinational cycles or gated clocks are able to achieve near 100% simulation coverage, even with only 1000 clock cycles of testing time.

4.2 Identification of Forced Errors

The true effectiveness of the system is in the ability to find errors introduced by the synthesis algorithms. To test this we created debug-options for the synthesis tool to intentionally invert one random logic signal in the netlist.

Since current designs typically contain from 5000 to 100,000 logic elements, this type of forced error is a relatively small change to the netlist. It does not change the connectivity, which would make the error quite easy to find, and this is also a relatively common sort of problem for a synthesis tool to introduce at the gate-level.

The split bars of Figure 7 indicate the proportion of designs, by simulation coverage, for which simulation identifies the erroneous connection. Overall, we can identify the forced error 35% of the time on simulation coverage of 52%. The metric is higher for smaller designs, where the error can be seen 50.6% of the time on simulation coverage of 67%. On larger designs only 22% of errors are detected on simulation coverage of 40%. This follows intuition, because we are only observing external pins, and larger designs, which could have greater latency, can take longer to propagate the error. In general, then, it makes sense to simulate longer for larger designs.

One might think that 35% probability of finding an error is low. In the case of chip-tests it is not at all useful. But for the purpose of identifying problems in synthesis as a supplemental tool it is more than sufficient – suppose that a single such error might occur in only 10% of our 150 designs and we have a 35% chance of noticing it with this quick simulation. Then the probability that such a problem in the synthesis tool will not be caught is (0.65)^15 or <<1%. This is excellent for debugging purposes, as well as being a strong addition to other testing schemes. Here our assumption is that a wire inversion is a good model of possible synthesis errors. Even if the reader disagrees with that model our point remains valid—the power of testing more than one design at a time should be evident.

We further tested the method when more than one forced errors are introduced per design. These results are shown in Figure 9.

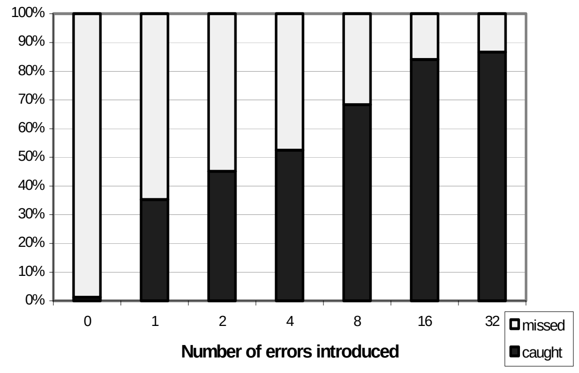

Figure 9. Probability of finding errors as more of them are introduced into the netlist.

The probability of identifying incorrect results in a single design increases asymptotically with the number of errors introduced, from 35% in the base case to almost 90% when 32 forced errors are introduced in the netlist.

5. FALSE POSITIVES AND OTHER PITFALLS

The largest problem with verification is the analysis of false positives – test-cases that are reported as simulation mismatches (flagged errors) even though the synthesis is correct. Debugging of false positives provides useful information into the testing problem in itself, but only if the number of false positives is manageable.

As a metric of false positives, we can compare the results of two trusted synthesis outputs using the methodology of Section 4, and then manually explore the simulation mismatches. This shows that we obtain a manually verified false positive in about 2% of our design-set for the default simulation time, and up to 5% when the designs are simulated for 10X the normal time.

This process was done in concert with the development of RVEC. In reality we began with very high proportions of false positives and this 2% rate represents the current status of the tool.

It is interesting to look at the source of false positives, how they influenced modifications to the vector generation process, and those issues that we can't actually solve at this time.

We define three types of simulation. Functional simulation occurs on the netlist before most synthesis processing. Zero-delay simulation occurs on the netlist after conversion to basic FPGA logic elements, but where all routing and LUT delays are set to 0. Timing simulation occurs on the annotated netlist after placement when all routing and cell delays are known. Our experimentation is using zero-delay simulation.

The two primary classifications of false positives are those arising from simultaneous switching and those arising from combinational cycles in the netlist (i.e. latches).

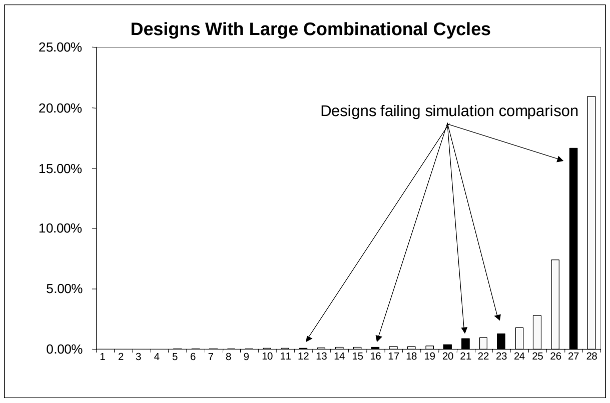

Figure 10. Nodes that are part of a long combinational cycle are shown as percentage of total nodes (y-axis). The dark bars are designs with false positives.

When two signals such as a DFF clock and asynchronous secondary signal (e.g. clear) arrive simultaneously, the order in which ties are processed in slightly different netlists will cause different results in simulation for the following clock-cycle until the clear is applied. Simultaneous switching is solved in RVEC by the interaction with the synthesis tool. Because we know the characteristics of such signals before vector generation, we can ensure prioritization of the signals by slightly delaying one in a consistent manner. Note that the hardware implementation of the FF in the FPGA typically has prioritization on these signals, so we are basically emulating this for functional simulation. Regular inputs should not change at the same time as well. A good rule of thumb we came up with is that every vector (entry in the hash table) should contain one and only one signal value that is different from the previous vector (no simultaneous signal switching at any one time).

Simulation mismatches dropped by 5X when the proper handling of asynchronous and clock signals and non-simultaneous switching were added to RVEC.

A second issue is latches. Randomly generated vectors perform poorly on designs that make extensive use (intentional or unintentional) of asynchronous logic. A well-designed latch will perform properly in both functional and timing simulation given valid input, but can oscillate given input that is not expected by the designer. In FPGA based designs the problem is not really intentional latches, but poor coding style that generates false-path latches. These become large combinational cycles in the timing graph, and cause different behavior on different netlist orderings, unless the paths are known to be false-paths for timing.

About 10% of designs will have some asynchronous logic. The designs with large combinational cycles are typically ones with false paths. Figure 10 shows the top 30 or so designs in the design-set along with their number of long combinational cycles as a percentage of design-size in LEs. We identify the 5 designs which generate false positives in simulation, all of which fall at the top-end of the scale (remaining designs with no long cycles are not shown).

We have no effective way of dealing with designs containing long combinational cycles using this method, so these are tested with standard regression tests and other means. We hope to remove this limitation in the future. For practicality reasons we simply exclude the known "bad apples" from the designs used in practice for testing with this method.

Other designs that are problematic for this type of testing are complicated I/O protocols, such as PCI. It would be difficult for any automatic tool to generate appropriate vectors to accomplish the necessary startup handshaking to properly initialize the design, and without initialization the circuit cannot be exercised.

6. VERGEN: DESIGNS WITH VECTORS

RVEC is a useful addition to the overall testing methodology. However, for debug and test purposes, we also want to provide particularly difficult designs for the synthesis tool to deal with while at the same time using realistic constructs. Real designs don't necessarily hit all the corner cases.

VERGEN is a ~2000 line C program that spits out pseudo-random Verilog HDL. It also generates compile scripts and a set of vector inputs for MAX+PLUS II (Altera's previous-generation software) or Quartus. It can be tuned at compile time for designs of any size, the current favorite settings make designs from ~7K to ~12K lines of Verilog which map into ~1.5K to ~2.5K logic cells. Thus far, we have only used VERGEN to exercise synthesis algorithms, which generally don't need 50,000 LUTs, but we hope to apply it to place&route in the future by generating large designs. Table 1 shows some sample small designs output by VERGEN along with their LE counts. Though these designs are 1500 to 2500 LEs, in theory the designs can be any size.

Table 1. Designs from VERGEN, showing Verilog lines of code and 4-LUT logic elements in Stratix [9].

Design	Verilog	LE
vergen01	9525	2256
vergen02	8585	2022
vergen03	8485	1931
vergen04	9118	2183
vergen05	8960	1972
vergen06	10233	2305
vergen07	9113	2024
vergen08	9393	2240
vergen09	8298	1933
vergen10	8314	1788
vergen11	9239	2256
vergen12	9063	2017
vergen13	9292	2027
vergen14	8996	2021
vergen15	11068	2023
vergen16	9177	2667
vergen17	7371	2093
vergen18	9113	1554
vergen19	9081	2085
vergen20	8792	2026
vergen21	9232	1937
vergen22	9104	1931
vergen23	8317	2001
vergen24	8653	1959
vergen25	9114	1972

The algorithm creates a bank of input pins then installs chunks of logic reusing a percentage of the signals as it goes. Leftover signals are routed to output pins. The selection of logic chunks is random with some heuristic tuning and parameterization to prevent creating too few or too many output signals. Other controllable attributes are critical path length and numbers of registers (before minimization) and the ability of different constructs being used.

The high-level blocks used include finite state machines, 4:1 MUX, random 3 in 3 out tables, 8 bit linear feedback shift-registers, tristate MUXes, tangles of XOR gates, inverters, priority encoders, 3 input < comparators, and registers with an assortment of secondary signals on 2 clock and reset domains.

Because VERGEN is generating the Verilog code, it also generates appropriate test vectors for use in simulating the code. So RVEC is not needed on VERGEN designs.

Figure 11 shows an example output of VERGEN. The actual output is about 8000 lines of Verilog, which we have hand-edited to fit in one column. Not all language constructs are shown, but this illustrates a number of the ones used: Notice the comparator used to generate alpha in the first always block, marvin is a simple DFF, ebay is a registered 4:1 mux, and {starbucks,atchoo,thor,hinge,hulk} form a 5-bit counter. VERGEN is particularly useful at exercising state-machine processing, arithmetic and high-level synthesis operations, though we are easily able to add other interesting objects (multipliers, switching networks, etc.) as the needs arise.

The amusing node names serve the very serious purpose of speeding up debugging. People can remember that they saw a particular signal called atchoo but not dff23421.

VERGEN generates clean synchronous design models (e.g. no incomplete case statements) so we get very high simulation coverage – 85% to 95% over 1000 clock cycles, and no false positives. Error detection on these designs is 81% for 1000 clock-cycles, vs. the 35% seen on the large industry designs. As a further advantage, we can target structures that trigger specific algorithms and thus exercise them well and introduce a good chance of finding errors.

Note that various different approaches to generating synthetic test designs exist in the literature (e.g. [7], [11], [6]). Of these Iwama and Hino [7] is the only one intended for synthesis evaluation but this is aimed at quality of results testing rather than correctness, and generates gate-level netlists rather than HDL, by modifying a seed circuit, so does not exercise Verilog elaboration and high-level synthesis.

7. CONCLUSIONS

In this paper we have discussed the overall problem of synthesis verification using simulation. This included a study of the use of random-vector generation, identifying the issues and pitfalls that arise in simulation with random vectors, and with FPGA design simulation in general.

In concert with this analysis we outlined a tool, called RVEC, to intelligently generate vectors for simulation testing on large FPGA designs. This tool constructs vectors in combination with information from the synthesis tool to provide proper behavior for clocks, asynchronous signals, bi-directional pins and other special logic, and solve the problem of false positives arising from simultaneous logic transitions and partially solve the problem on latches which occur on these designs. The purpose of RVEC is to

```
// Made by one of Gregg's many toys - 04-10-02
module vsm_009 (
  bravo, charlie, delta, babette, yeanyow
  <<snip>>
  clock0, clock1, reset0, reset1);
  <<snip>>
  input jupiter;
  <<snip>>
  output lala;
  <<snip>>
  reg purplehaze;
  <<snip>>
    parameter alpha_0=0,alpha_1=1,alpha_2=2,...
    reg [2:0] alpha;


always @(posedge clock0 or posedge reset0)
  begin
    if (reset0)
      alpha = alpha_0;
    else
      case (alpha)
        alpha_0: begin
          if ({kappa,salmon,ebi } == 4)
            alpha = alpha_6;
          else if ({kappa,salmon,ebi} == 5)
            alpha = alpha_4;
          <<snip>>
          end
        default: alpha = alpha_0;
      endcase
  end

always @(posedge clock0 or posedge reset0)
  begin
    if (reset0)
      marvin = 0;
    else
      marvin = bullet;
  end

always @(posedge clock0 or posedge reset0)
  begin
    if (reset0)
      ebay = 0;
    else
    case ({nail,yankee})
       0 : ebay = duck;
       1 : ebay = mortar;
       2 : ebay = romeo;
       3 : ebay = juliet;
       default : ebay = 0;
    endcase
  end

always @(ocha)
  begin
      tako = !ocha;
  end

always @(posedge clock1 or posedge reset1)
  begin
    if (reset1)
      {starbucks_atchoo_thor,hinge,hulk} = 0;
    else
      {starbucks,atchoo,thor,hinge,hulk} =
        {starbucks,atchoo,thor,hinge,hulk}+1;
  end
endmodule
```

Figure 11. Edited code snippets from 8000 line VERGEN design in Verilog.

allow testing on large industrial designs that do not already have test-vectors.

To further exercise specific constructs seen in synthesis, we described an additional tool, VERGEN, which outputs both large Verilog designs and vectors to test them. VERGEN is intended to exercise corner cases in synthesis by connecting together common building blocks. Though not described in detail, the circuit-generation provided by VERGEN is also a strong step forward in the automatic generation of synthetic test designs.

Through various statistical means, we showed the validity of these tools to help the testing process. With RVEC we were able to show that an error in a synthesis tool only needs to appear a single time in about 10% of netlists to have a high probability of being caught automatically. With designs from VERGEN this probability is significantly higher, because we control both the design and vector generation processes.

We should emphasize that these methods and tools are additive to standard software testing methodologies (e.g. hand design of regression tests, code reviews, assertions and self-checking code) and testing designs with human-generated vectors, plus other tools not described. They provide additional confidence in the quality of synthesis results as black-box tests, and provide guidance on problems during the debug of new algorithms. We would not recommend using any one tool as the only testing mechanism on a complicated software system.

We have successfully applied these tools and others for debugging and testing Quartus native synthesis (QNS) vs. MAX+PLUS II, QNS across releases and for new synthesis algorithms, and in particular for debugging some of the more difficult algorithms such as observability don't-care minimization and register re-timing. Others at Altera have used them for debugging and testing advanced algorithms in place&route that manipulate LUT-masks, perform register re-timing, or otherwise modify the connectivity of the netlist.

ACKNOWLEDGEMENTS

The authors would like to thank Sinan Kaptanoglu and David Mendel for reading the paper and making many valuable suggestions on the exposition.

REFERENCES

[1] M. Abramovici, M.A. Breuer and A.D. Friedman, *Digital Systems Testing and Testable Design.* IEEE Press, 1990.

[2] D. Brand, "Verification of Large Synthesized Designs", in *Proc. Int'l Conference on Computer-Aided Design (ICCAD).* 1993. pp. 534-537.

[3] A. El-Maleh, T. Marchok, J. Rajski and W. Maly, "On Test Set Preservation of Retimed Circuits", in *Proc. Design Automation Conference (DAC).* pp. 176-182, 1995.

[4] G.D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms.* Kluwer, 1998.

[5] S.Y. Huang, K.T. Cheng and K.C. Chen, "Verifying Sequential Equivalence using ATPG Techniques." In *ACM Trans. Design Automation of Electronic Systems (TODAES).* Vol.6 No.2, pp 244-275, 2001.

[6] M. Hutton, J. Rose and D. Corneil. "Automatic Generation of Synthetic Sequential Benchmark Circuits", *IEEE Trans. CAD.* Vol. 21 No. 8, pp 928-940, 2002.

[7] K. Iwama and K. Hino, "Random Generation of Test Instances for Logic Optimizers," in Proc. *Design Automation Conference (DAC).* pp. 430-434, 1994.

[8] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning", in *Proc. Int'l Conference on Computer-Aided Design (ICCAD).* pp. 538-543, 1993.

[9] D. Lewis *et. al.*, "The Stratix PLD Routing and Logic Architecture". In *Proc. ACM/IEEE Symposium on FPGAs (FPGA),* (to appear), 2003.

[10] P. Mazmuder and E.M. Rudnick. *Genetic Algorithms for VLSI Design, Layout and Test Automation*. Prentice Hall, 1999.

[11] J. Pistorius, E. Eegai and M. Minoux, "PartGen: A generator of very large circuits to benchmark the partitioning of FPGAs," *IEEE Trans. Computer-Aided Design.* Vol. 19, pp. 1314-1321, 2000.

[12] E.M. Rudnik and J. Patel, "Combining Deterministic and Genetic Approaches for Sequential Circuit Test Generation", in *Proc. Design Automation Conference (DAC).* pp. 183-188, 1995.