VeriFuzz: Verilog Simulator Fuzz Testing

Interim Report

Yann Herklotz Grave

January 22, 2019

Contents

1 Introduction

Fuzzing is used to test software in an automatic way, by passing randomly generated data to the program and checking that it handles the input data correctly. This could simply mean that the program does not crash when it is passed the data. Fuzzing was initially used to check the reliability of UNIX utilities [1] and it was found that even commonly used programs had some unreported bugs in them. It was also used to test cache controllers by passing random data to them and checking that they do not access any illegal states and adheres to the high-level specification of the cache controller [2]. This kind of fuzzing is also extremely effective in testing network protocols, where the input data is often noisy but should still be handled gracefully.

Instead of passing random bytes to the program being tested, a more modern way of fuzzing programs that accept a specific file format is to generate random input data that follows the specification to generate a random but valid input. The main benefit being that this input should be a more thorough test of the program itself, rather than testing the parser. This notion of fuzz testing can be applied to compilers, whereby a large number of random programs are automatically generated and given to the compiler, in the hope of discovering bugs. CSmith [3] is one example of a fuzzing tool for C compilers that has been applied with great success to conventional compilers for languages like C, and has led to the discovery of hundreds of bugs in extremely common compilers such as in GCC and Clang. It has even found bugs in verified compilers such as CompCert [4], which all occurred in unverified parts of the compiler and motivated the verification of those parts.

Other than CSmith, there has been a lot of work on fuzzing different kinds of compilers using various different fuzzing methods, such as a JavaScript fuzzer called jsfunfuzz [5], a Rust typechecker fuzzer [6], a DOM fuzzer [7]. These all show that fuzzing is an extremely effective method to test all

kinds of software. However, there has not been a lot of research into fuzzing hardware simulators, even though these are often quite unreliable.

The aim of this project is to investigate how fuzzing can be applied to test simulators for Verilog. A key component will be to devise a way to generate random (but valid) Verilog files, and then to systematically run these through Verilog compilers and simulators, to see if they are all handled correctly.

2 Project Specification

The aim of this project is to write a fuzzing tool that generates random Verilog to test different simulators and synthesis tools. The main motivation for testing these simulators is that first of all, Fuzzing has been successful in finding bugs in popular compilers such as different C compilers and other languages, by finding hundreds of bugs in modern compilers that have since been fixed. Secondly, until now there has not been much research in writing such a fuzzing tool for Verilog simulators, which means that there is a lot of potential to find more bugs and improve the quality of these tools that way.

Once we can generate random Verilog reliably, these can be analysed and reported to the simulator vendors.

2.1 Deliverables

VeriFuzz is the library that will implement the random Verilog generation and the distribution of the code to the different simulators. The core contributions of this project are the following

1. Correct, deterministic random Verilog generation.
2. Interface to simulators and synthesis tools to test the code by running it simultaneously on all of them.
3. Fuzz testing of the following synthesis tools: Yosys, XST and Vivado.
4. Fuzz testing of the following simulators: Yosim, Iverilog.

The first core deliverable is to generate random, well-formed and deterministic Verilog files, covering a fraction of the language. This is an achievable goal and would be a good start towards building a successful fuzzing tool, as the general method of generating the basic Verilog will have been established. The second core deliverable is to write a top-level loop to feed the Verilog files into two simulators and retrieve the outputs to compare them. The latter would require the fuzzing tool or a separate script to handle the flags and output format for two different simulators correctly. These two deliverables should achieve a small, working prototype of the fuzzing tool which can then be extended with the following extension goals.

First of all, the random generation could handle more of the Verilog Specification. This would mean that more of the Verilog implementation in the simulator would be tested and the fuzzing would be more likely to find a bug. Secondly, another extension could be to generate more complicated Verilog files, with more complicated logic, which would also increase the chance of finding bugs in the simulator. Another possible extension would be to handle more than two simulators. In addition to that, a parallel top-level loop for running the simulations could also be implemented, which would speed up the simulation step greatly.

More interesting extension that could also be added to the project could be mutating the generated Verilog code in multiple ways, so that it still stays deterministic. One example that it could be changed in a way that does not affecting the result, and then checked on the same simulator against the original code to make sure that the simulator outputs the same result as well. The Verilog code could also be made slightly invalid, which would mean that a bug would be found if the Verilog code passes the simulation and outputs some result.

Finally, a test-case reducer would also be useful, as once bugs are found, it would help in finding what the root cause of the bug was. It would also help when reporting the bugs by only showing a minimal test case that makes the simulator or synthesis tool fail. This could be done by performing a binary search on the code and reducing it until the only code that is left is the code that is making the simulator fail.

2.2 Summary of Risks

A few risks with this project should be mentioned.

First of all, the simulation time could be an issue with regards to finding bugs in the different simulators. This is because fuzzing tools normally rely on generating millions of random programs to find bugs as quickly as possible. However, simulation takes much more time compared to compilation, therefore the generated Verilog files will have to be much more interesting to be able to find bugs in the simulator.

Another potential risk is the support of the Verilog language standard across different simulators. Many simulators support different parts of the Verilog IEEE standard, so it might not be possible to test all the different simulators with the same random code, which might be a problem.

In addition to that, there might be commercial restrictions in the large simulators that mean that we cannot use all the provided features.

Flakiness of these simulators is another potential problem that could be encountered in this project, as that would mean that the random Verilog generation could possibly find too many uninteresting errors. It could also mean that the errors found are inconsistent and therefore not important.

3 Background

Even though there has not been much work on testing hardware simulators using fuzzing techniques there have been large advances in fuzz testing that can be applied to test the simulators. Fuzz testing has transformed from generating random bytes to test if a program crashes to automatically learning how to generate better test cases to cover all the possible code paths in a program. There are many approaches that can be taken to find deeper and more meaningful bugs, notably, restricting the random input generation to a syntactically and semantically valid subset of a language a compiler accepts, as well as checking that the code generated by the compiler is correct.

Other related works consist of ways in which the reliability of hardware simulators can be improved.

3.1 Fuzzing

The term "fuzzing" was introduced by Miller *et al* [1], which stood for generating a random input that the program being tested did not expect. This was achieved by generating large chunks of random bytes that were passed directly to the program under test (PUT). The criteria that had to be fulfilled by the PUT was that it should not crash once it has received and processed the input. Even with this simple fuzzing method, many different bugs were found in common UNIX tools such as emacs, vi, csh, spell and uniq. The types of errors being found were quite diverse, ranging from array or pointer manipulation bugs to race conditions. However, as the input to the programs were just random bytes, a lot of these bugs were from the early stages of the programs execution where the input was being parsed. As fuzz testing methods developed, more advanced generation and evaluation methods were being used to achieve better results.

First of all, there are three main fuzzing techniques that greatly affect the different stages in a fuzzer [8].

Black-box fuzzing At the simplest but often more general level there are so called black-box fuzzers. The term "black-box" often refers to testing a program without making any assumptions or knowing about the inner workings of the PUT [9], [10]. Often fuzzers that fuzz specific protocols

or grammars will be black-box fuzzers, as they are normally used to test various implementations of the protocol or compilers that accept that grammar. One such example is CSmith, which generates random C programs that can be used to fuzz existing C compilers such as GCC, Clang or CompCert. As it generates a random C file completely independently from the compiler that will be tested, it does not rely on the internals of how exactly the compiler works, and can therefore be used to test any C compiler. VeriFuzz will also fall into this category, as it should generate a Verilog file completely independently to the simulator being tested.

White-box fuzzing At the other extreme, there is white-box fuzzing [11], [12], which leverages symbolic execution and dynamic test generation to create tests that are specific to the PUT. Instead of providing input values to the program, symbolic execution [13] assigns symbols to the inputs and passes through equations in terms of those symbols. If there are conditionals, one can construct a tree of the different execution paths and record the constraints that each conditional statement imposes. This can be used to generate dynamic tests that cover all the possible branches, which may have been extremely unlikely to have been tested with random black-box testing. For example, in Listing 1, if the values generated for a are random, there would only be a $1/2^{26}$ chance that the code path would be executed, whereas white-box fuzzing would generate tests for all possible execution cases.

```
int func(int a) {
    if (a < 64) { abort(); /* error */ }
    return 0;
}
```

Listing 1: Error unlikely to be hit with random black-box testing, but white-box testing will generate tests for all possible branches.

Grey-box fuzzing White-box fuzzing is time consuming and computationally expensive, as it requires in depth analysis of the PUT which may be unfeasible for large programs. Instead of performing in depth analysis using symbolic execution, a more lightweight analysis of the PUT can be made, such as using static analysis on the code, or introducing some instrumentation to improve the test case generation. An example of a grey-box fuzzer is AFL [14] which uses compile-time instrumentation and genetic algorithms to discover interesting test cases that would change the internal state of the binary in interesting ways. Another example is VUzzer [15] which uses static and dynamic analysis of the programs it is supposed to test to produce better test cases that will provide the most coverage.

The focus will now be on analysing different black-box fuzzers, as these are the most common type of fuzzer and are also used the most to test compilers. As mentioned earlier, VeriFuzz is also a black-box fuzzer.

3.2 VlogHammer

VlogHammer [16] is a differential tester targeting simulators, both commercial and open source. It was initially built to test the open source Verilog synthesis suite called Yosys [17], which was written by Clifford Wolf, but because of the nature of differential testing, it found bugs in other simulators as well. It now supports many common commercial Verilog synthesis tools such as Quartus [18], Xilinx ISE (XST) [19] and Vivado [20], as well as simulators such as Xilinx Isim and Xsim which come with ISE and Vivado respectively, Modelsim which is included in Quartus, and finally an open source simulator called Icarus Verilog [21]. There is a distinction between synthesis and simulation in Verilog simulators and the aim of VlogHammer is to test the correctness of synthesis tools by using simulators to check the output.

3.2.1 Synthesis and Simulation

Logic synthesis is the transformation of higher level code that does not have a direct translation to hardware, into lower level components that can be modelled directly by using an FPGA or ASIC [22]. Synthesis tools such as Yosys, Vivado or Quartus can synthesize higher level Verilog to a lower level model called a net list, which is a representation of the lower level components. The format of the net list can vary widely, but for analysis purposes, these tools allow the net list to be expressed in Verilog again, by using an extremely limited set of features. The net list is tailored specifically to a type of hardware that it will eventually be placed on, meaning that different synthesis tools will make different assumptions about what operations the hardware supports natively and which operations have to expressed in the net list separately. Finally, To implement the net list in hardware, a process called Place and Route then takes place to map the net list exactly onto the hardware and outputs a bit stream that will configure the FPGA correctly.

On the other hand, simulators do not target hardware, but instead aim to simulate the Verilog code by simulating the circuit in software. The circuit can then be tested in different ways by being passed values and checking that the wave forms in different parts of the design behave properly. The benefits of using a simulator is that synthesising and performing Place and Route on a design can often take hours, whereas simulators can execute the design with an appropriate test bench quite quickly. Simulators also often support useful Verilog constructs which cannot be synthesized, such as system tasks or the tri-state wire, which cannot be modelled correctly in hardware. Simulators come in many different forms. One example is Verilator [23], which generates efficient multi-threaded C++ from the Verilog model which can then be compiled and executed. Another example is Icarus Verilog [21] which instead compiles the Verilog down to an intermediate form that can then be executed separately.

3.2.2 Workflow

VlogHammer aims to test the correctness of synthesis tools. The main workflow used by VlogHammer is to first generate many small random Verilog modules. These are then passed to each of the synthesis tools to be tested and synthesized by each of them. If there are any failures during this process, the failing tools will be noted and the failing files will be set aside to be included in the final report. However, this should be extremely unlikely, as unlike software compilers, these synthesis tools will try to output a synthesised file, even with unsound Verilog files [REFERENCE].

1. Synthesis

   Once the synthesized files are generated, they have to be made architecture independent to be compared to each other, as each synthesis tool has a different target platform. These architecture differences are expressed in the synthesised Verilog files by specific module instantiations that the target platform supports natively. As the author of VlogHammer is also the author of Yosys, the synthesised files are made architecture independent by converting them the Yosys net list format called "ilang". Extra Verilog source files are supplied for the conversion which provide Verilog models of every possible module that could be instantiated in the Verilog net lists for different hardware modules that the target architecture already supports.

   Once all the "ilang" files are generated for all the synthesised modules for the synthesis tools, they can finally be compared against each other for logical equivalence, by using a form of differential testing. The original Verilog module that was passed to all the synthesis tools is also compared against the output after synthesis, to make sure that these are also equivalent. Yosys provides a SAT solver that can be used for various tasks to prove properties of the Verilog tools. It is built on top of MiniSat [24] and provides support for many different formats such as Verilog, the aforementioned "ilang" and SMT-LIBv2 [25], a language standard that is often used for defining conditions and CNF (Conjuncive Normal Form) expressions. An example query is shown in Listing 2, which proves that `y1`, defined in one module, is equivalent and therefore always equal to `y2`, defined in another module, no matter what is passed to the inputs `a1`, `a2`, `a3`. The inputs are passed to both modules which output `y1` and `y2` respectively.

```
sat -timeout 20 -verify-no-timeout -show a1,a2,a3,y1,y2 \
    -ignore_div_by_zero -prove y1 y2 verilog_module
```

Listing 2: Example SAT query in Yosys to prove equality between y1 and y2.

2. Simulation

   As the comparison step between the synthesised Verilog modules uses formal proofs to show that they are equivalent or not, there can never be any false positives. If the synthesised modules are found to be equivalent, they will always have the same output for all possible inputs to the module. This would mean that it is extremely likely that both of those simulators are correct and generate the correct code after synthesising the module, as it is unlikely that they both generate the wrong code. In addition to that, if the synthesised code is shown to be equivalent to the input module, that means that the synthesis step was definitely correct.

   If, however, the tool finds that the two modules are not equivalent, that does not necessarily mean that one of the modules generated wrong code, as false negatives are possible. In Verilog, there are expressions that will evaluate to an undefined value. Although this can be handled by simulators, as they can assign `x` to values that are undefined and continue simulation, synthesis tools cannot as in hardware, wires or registers cannot have an undefined value. Instead, any undefined value during synthesis is interpreted as an undefined signal, which means that its value is implementation defined and can change in between synthesis tools.

   To detect these false negatives, the initial Verilog modules together with the synthesised modules are simulated with a test bench that randomly assigns values to the inputs and records the outputs. In addition to random inputs, the counterexamples found by the SAT solver are also added to the test bench to trigger the edge case that produced the inequality between the two synthesised modules. All the outputs of the simulator are hashed, and if there are any undefined bits in the output, they are masked. The hashes that were obtained from all the synthesised modules are then compared. If the modules were found to not be equal to each other, but the output hashes from the two simulators agree, that would mean that it was a false positive and there were undefined symbols in the design.

3. Evaluation

   Finally, a table, which can be seen in Figure 1 is generated to get an overview of the failures and where they occurred, and is combined into a large report

	xst	yosys	icarus	yosim
xst	PASS	FAIL	643e585a	643e585a
yosys	FAIL	PASS	4f20d544	4f20d544
rtl	FAIL	PASS	4f20d544	4f20d544

Figure 1: Table included in report for test case failure in XST, generated by VlogHammer.

3.2.3 Limitations

Even though VlogHammer has found many bugs in all the synthesis tools that were tested, as synthesis is extremely complicated to get right, there are still some limitations with it.

The main limitation is that VlogHammer does not support the generating behavioural code, such as `always` and `initial` statements. Instead, it focuses on generating correct Verilog expressions and testing those.

3.3 Verilog

Verilog [26] is a Hardware Description Language (HDL), which is commonly used to design and verify digital circuits. Verilog contains a subset that can be synthesized to hardware, such as being

placed on an FPGA. The rest of the Verilog language can be simulated and has many tools to make verification easier. That means that a design can be written in Verilog which can eventually be synthesized, but is tested thoroughly before hand using a test bench by simulating it.

3.3.1 Grammar specification

To make an effective tool that can fuzz the Verilog grammar, on has to find a way to model the grammar. It is therefore useful to analyse the grammar to check if it is recursively enumerable. A recursively enumerable (r.e.) grammar is a language L for which there exists a Deterministic Turing Machine (DTM) which accepts the language and does not terminate otherwise, meaning it will accept any input $x \in L$ and will not terminate if $x \notin L$. Verilog itself is recursively enumerable, as it is possible to construct a parser that will parse any syntactically correct Verilog code and hang forever if there is a syntax error. It is therefore interesting to examine the following points.

1. R.e. semantically correct Verilog
2. R.e. deterministic Verilog

3.3.2 Determinism

There are constructs in Verilog that are not deterministic, meaning that they might give different results when run on different simulators, however, that may not be a bug in either of the simulators. One such example can be seen in Appendix A.2.

4 Implementation Plan

First of all, the main basis of the VeriFuzz library has to be implemented, so that testing of the proposed synthesis tools and simulators can begin. After that has been done, many different improvements can be added to the library to make the testing more efficient and hopefully find more bugs.

5 Evaluation Plan

The main point that should be used to evaluate the VeriFuzz library is the number of bugs that it finds in the various synthesis tools and simulators. This can be done in multiple different ways. First of all, one can compare different revisions of the library, which would compare how effective the new improvements to the library were, and if they did in fact improve the rate of finding bugs. Secondly, the rate at which VlogHammer finds bugs can be compared against VeriFuzz, which could show that making the test generation a bit more complex does have benefits and finds different bugs when compared to VlogHammer. VeriFuzz supports random Verilog code generation in three different ways, and more methods of generating random code might be added in the future. These can also be evaluated against each other, to examine if any of the methods are more effective at finding bugs or not. Some random generation methods, such as the generation from a Directed Acyclic Graph (DAG), are more complicated than other naive methods of generating Verilog. It is therefore expected that the more complex generation methods should give the best results, as it would otherwise not be worth it to make those optimisations.

6 Ethical, Legal and Safety Plan

There are no ethical, legal or safety implications of the project.

7 Conclusion

In conclusion, the next step is to make the library functional.

References

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[2] D. A. Wood, G. A. Gibson, and R. H. Katz, "Verifying a multiprocessor cache controller using random test generation," *IEEE Design Test of Computers*, vol. 7, no. 4, pp. 13–25, Aug. 1990, ISSN: 0740-7475. DOI: 10.1109/54.57906.

[3] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," *SIGPLAN Not.*, vol. 46, no. 6, pp. 283–294, Jun. 2011, ISSN: 0362-1340. DOI: 10.1145/1993316.1993532.

[4] X. Leroy *et al.*, "The CompCert Verified Compiler," *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[5] J. Ruderman, *Introducing JsFunFuzz*, 2007. [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz (visited on 11/23/2018).

[6] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using clp (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 482–493. DOI: 10.1109/ASE.2015.65.

[7] M. Zalewski, *Announcing crossfuzz*, Jan. 2011. [Online]. Available: http://lcamtuf.blogspot.fr/2011/01/announcing-crossfuzz-potential-0-day-in.html (visited on 11/23/2018).

[8] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[9] B. Beizer, *Black-box Testing: Techniques for Functional Testing of Software and Systems*. New York, NY, USA: John Wiley & Sons, Inc., 1995, ISBN: 0-471-12094-4.

[10] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. New York, NY, USA: John Wiley & Sons, Inc., 2011, ISBN: 978-1-118-03196-4.

[11] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *NDSS*, vol. 8, 2008, pp. 151–166.

[12] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *SIGPLAN Not.*, vol. 43, no. 6, pp. 206–215, Jun. 2008, ISSN: 0362-1340. DOI: 10.1145/1379022.1375607.

[13] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: https://doi.org/10.1145/360248.360252.

[14] M. Zalewski, *American fuzzy lop*, 2015. [Online]. Available: http://lcamtuf.coredump.cx/afl/ (visited on 01/15/2019).

[15] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA: NDSS, Feb. 2017. DOI: 10.14722/ndss.2017.23404.

[16] C. Wolf, *Vloghammer*. [Online]. Available: http://www.clifford.at/yosys/vloghammer.html (visited on 01/11/2019).

[17] ——, *Yosys Open SYnthesis Suite*. [Online]. Available: http://www.clifford.at/yosys/ (visited on 01/11/2019).

[18] Intel, *Intel Quartus*. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html (visited on 01/14/2019).

[19] Xilinx, *Xst synthesis overview*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_using_xst_for_synthesis.htm (visited on 01/11/2019).

[20] ——, *Vivado Design Suite*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html (visited on 01/14/2019).

[21] S. Williams and M. Baxter, "Icarus verilog: Open-source verilog more than a year later," *Linux J.*, vol. 2002, no. 99, pp. 3–, Jul. 2002, ISSN: 1075-3583. [Online]. Available: http://dl.acm.org/citation.cfm?id=513581.513584.

[22] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Springer Science & Business Media, 2006, ISBN: 978-0-387-31004-6.

[23] W. Snyder, P. Wasson, and D. Galbi, "Verilator," *Direct search methods: then and now*, 2007.

[24] N. Sorensson and N. Een, "Minisat: A sat solver with conflict-clause minimization," *SAT*, vol. 2005, no. 53, pp. 1–2, 2005.

[25] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017, Available at www.SMT-LIB.org.

[26] "Ieee standard for verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–560, 2006. DOI: 10.1109/IEEESTD.2006.99495.

A Verilog

A.1 Test bench and design example

Verilog test bench testing a module which could be synthesized.

```
module and_comb(input wire in1, input wire in2, output wire out );

   and and1(out, in1, in2);

endmodule

module main;
   reg a, b;
   wire c;

   and_comb gate(.in1(a), .in2(b), .out(c));

   initial
     begin
        a = 1'b1;
        b = 1'b1;
        #1;
        $display("%d & %d = %d", a, b, c);
        $finish;
     end
endmodule
```

A.2 Non-determinism example

Example of code that has undefined behaviour and therefore depends on the simulator being used.

```
module top;
   reg ready;
   integer result;

   initial begin
      #10;
      ready <= 1;
      result <= 5;
```

```
   end

   initial begin
      @(posedge ready);
      if(result == 5) begin
         $display("result was ready");
      end
      else begin
         $display("result was not ready");
      end
   end
endmodule
```

A.3 VlogHammer generated code

Auto-generated code by VlogHammer.

```
module expression_00002(a0, a1, a2, a3, a4, a5, b0, b1, b2, b3, b4, b5, y);
   input [3:0] a0;
   input [4:0] a1;
   input [5:0] a2;
   input signed [3:0] a3;
   input signed [4:0] a4;
   input signed [5:0] a5;

   input [3:0]        b0;
   input [4:0]        b1;
   input [5:0]        b2;
   input signed [3:0] b3;
   input signed [4:0] b4;
   input signed [5:0] b5;

   wire [3:0]         y0;
   wire [4:0]         y1;
   wire [5:0]         y2;
   wire signed [3:0]  y3;
   wire signed [4:0]  y4;
   wire signed [5:0]  y5;
   wire [3:0]         y6;
   wire [4:0]         y7;
   wire [5:0]         y8;
   wire signed [3:0]  y9;
   wire signed [4:0]  y10;
   wire signed [5:0]  y11;
   wire [3:0]         y12;
   wire [4:0]         y13;
   wire [5:0]         y14;
   wire signed [3:0]  y15;
   wire signed [4:0]  y16;
   wire signed [5:0]  y17;

   output [89:0]      y;
   assign y = {y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15,y16,y17};

   localparam [3:0] p0 = (2'd1);
   localparam [4:0] p1 = {(3'd0),(3'd7)};
   localparam [5:0] p2 = (5'd10);
   localparam signed [3:0] p3 = {((4'd15)?(2'sd0):(4'sd3)),
```

```
                                (((4'd15)^(5'd31))|((5'd27)?(3'd5):(2'd1))),
                                ((4'd5)?(4'd7):(5'd21))};
    localparam signed [4:0] p4 = (^((-5'sd0)<=(2'd2)));
    localparam signed [5:0] p5 = ((((5'sd15)|(4'd11))==
                                 ((3'd0)?(3'd7):(-5'sd4)))&({4{(3'd2)}}==
                                                            {1{((3'd5)?(4'd11):
                                                                (4'd13))}}));
    localparam [3:0] p6 = (-(((~((-4'sd6)>(-3'sd1)))!==((3'd5)<<(5'sd0)))!=(5'd0)));
    localparam [4:0] p7 = {2{(!(~{3{(-5'sd9)}}))}};
    localparam [5:0] p8 = ((((-4'sd4)+(2'sd1))<((4'sd5)&(5'sd10)))<
                           ((4'd2 ** (4'd13))&((-2'sd0)<<(5'd3))));
    localparam signed [3:0] p9 = {{(-3'sd1),(3'd1),(4'sd3)},{(2'd1),(-4'sd1),(5'sd15)}};
    localparam signed [4:0] p10 = {(~|((2'd3)?(4'd11):(4'd9))),
                                   (~^{1{((3'd1)?(3'd1):(5'd14))}})};
    localparam signed [5:0] p11 = ({3{(3'd4)}}?({2{(5'sd9)}}^~
                                                {(5'sd9),(4'd3),(-5'sd3)}):
                                   (((2'd1)>>(-4'sd0))-((5'd2)?(2'sd1):(3'd0))));
    localparam [3:0] p12 = ((((5'sd0)^~(4'sd6))!==((3'sd3)<<(-5'sd5)))?
                            (-((5'd6)?(4'd15):(-5'sd8))):(5'd6));
    localparam [4:0] p13 = ((3'sd2)?(-5'sd5):(5'd1));
    localparam [5:0] p14 = (~^(!{{(-2'sd1),(-5'sd8)},(~|((4'd7)<(4'd3))),
                               (!{(-4'sd6),(3'sd0),(4'sd0)})}));
    localparam signed [3:0] p15 = (((5'd11)&(-4'sd5))^((5'sd4)<<(2'd1)));
    localparam signed [4:0] p16 = {3{((2'd1)<(3'sd1))}};
    localparam signed [5:0] p17 = ({3{(2'd0)}}?((3'd0)?(-5'sd1):(2'sd1)):
                                   (((-3'sd0)===(3'sd0))-((3'd3)===(5'sd14))));

    assign y0 = ({((4'd15)===({4{b3}}>>>(a4==a1)))}!={3{(p9<=p14)}});
    assign y1 = ((6'd2 ** p0)^(2'sd1));
    assign y2 = {2{(({3{b0}}!=(p6==p1))!={1{{1{(5'd30)}}}})}};
    assign y3 = ({4{b1}}?{4{a5}}:{1{{4{b2}}}});
    assign y4 = $unsigned((&(^$unsigned((+$signed($signed(a1)))))));
    assign y5 = (+(p6?p3:p14));
    assign y6 = (-(b3<b4));
    assign y7 = ((3'sd3)<<<((-2'sd0)));
    assign y8 = ((!(((b1-a5)&&(a1!==b0))===(|(b1&b1))))<<<
                 ((^(!(a0<<p12)))>=((a4^a5)!==(b5!==b5))));
    assign y9 = ((~$unsigned({(b3),(a5?a0:a5),{3{b4}}}))?
                 {4{(a3?p8:a2)}}:$unsigned((~|{4{a1}})));
    assign y10 = (~|((~(~(-p10)))^(-(-(p4||a4)))));
    assign y11 = ({a5,b1,a1}==(5'd24));
    assign y12 = (((-4'sd4))^(((p2+b2)==(4'sd3))|((a0==b1)==(2'sd0))));
    assign y13 = ((b2<b0)==(p7>=p6));
    assign y14 = {((5'd2 ** (-(+b2)))<<$signed({(b5>>p8),{p4,p16}}))};
    assign y15 = $signed((-(~({2{b2}}?$signed((b5-b1)):(p5>>>a0)))));
    assign y16 = $unsigned((~^(3'd2)));
    assign y17 = (~&(5'd14));
endmodule
```