# Finding and Understanding Bugs in FPGA Synthesis Tools

Yann Herklotz, John Wickerson

Akshay G

February 7, 2022

## Introduction: questions

- Why do they matter?
- What forms of bugs does this paper address?
- How do they address?
- What results did they get?

## What bugs?

- All our programs are ultimately run by the hardware.
- Some circuit out there (eg: CPU, GPU, FPGA, ASIC) ultimately runs whatever program we write.
- Circuits are also designed today using the same type of programs as above.
- High level synthesis tools exist for this purpose.
- If the HLS tools are not designed correctly, can we trust the hardware on which our programs run?

- All our programs are ultimately run by the hardware.
- Some circuit out there (eg: CPU, GPU, FPGA, ASIC) ultimately runs whatever program we write.
- Circuits are also designed today using the same type of programs as above.
- High level synthesis tools exist for this purpose.
- If the HLS tools are not designed correctly, can we trust the hardware on which our programs run?

Including running this pdf viewer for this presentation. We assume that whatever hardware that runs our programs is perfectly built.
Today's era, even hardware is designed by writing high level programs. High level synthesis tools are used to take this program and generate hardware. So if the HLS tools are not designed correctly, can we trust the hardware on which our programs run?

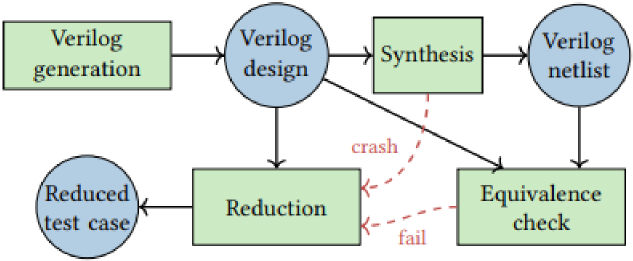
## What sort of bugs are addressed?

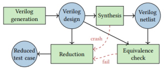
- Bugs are often in the several optimizations done by HLS tools.
- These optimizations are done to meet several needs such as timing, area, power, etc.
- The result after synthesis is a netlist (wires and their interconnection to circuit elements).
- If input program and netlist are not same, this is a bug.
- If the HLS tool crashes on a valid program, this is a bug.

- Bugs are often in the several optimizations done by HLS tools.
- These optimizations are done to meet several needs such as timing, area, power, etc.
- The result after synthesis is a netlist (wires and their interconnection to circuit elements).
- If input program and netlist are not same, this is a bug.
- If the HLS tool crashes on a valid program, this is a bug.

Sometimes the HLS tool crashes for a given program. If the program is syntactically and semantically correct, the HLS tool should not crash. Even if not, it should raise an error, not crash. This is also a bug in HLS tool.

## How are the bugs detected?





Generate Verilog program via random program generator. Give the program to HLS tool to generate netlist. If HLS tool crashes, perform reduction of input program to find out minimal test program to generate the bug. Check equivalence between the netlist and hte program we gave. If they are not equivalent, it is a bug. Perform reduction of input program to find out minimal test program to generate the bug.

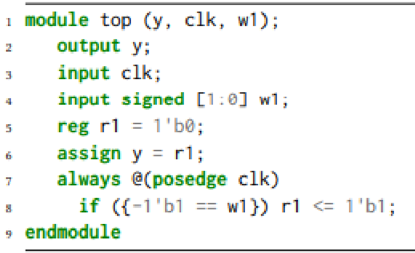
## What results were obtained?

- Four tools were tested this way: Yosys, Vivado, XST and Quartus Prime.
- Except Quartus prime, every other tool had at least one bug of the above types.
- Vivado and Yosys(dev ver.) also crashed for a few program inputs.

## Step 1: Program Elements

Program elements are:

- Modules (line 1).
- Wire (line 2,3,4) and variable (line 5).
- Unary and binary operators (line 8).
- Assignments (line 6).
- Conditionals (line 8).
- a few more ...



```
1 module top (y, clk, w1);
2    output y;
3    input clk;
4    input signed [1:0] w1;
5    reg r1 = 1'b0;
6    assign y = r1;
7    always @(posedge clk)
8      if ({-1'b1 == w1}) r1 <= 1'b1;
9 endmodule
```

## Step 2: Property of Verilog Programs generated

Each Verilog program is

- Syntactically correct if the HLS tool doesn't complain.
- Semantically correct if the HLS tool doesn't complain.
- Deterministic: they will not have
  - Divide by zero.
  - Wire input from two different sources.
  - Using wire not declared previously, etc.

## Step 3: Generating Verilog program

Programs are generated as follows

- Program elements assigned a frequency (modules, assigns, conditionals, etc).
- Context is list of: wire/var assigned, safe modules that can be created, parameters available for module, etc.
- Program is built sequentially using context, and updating context after each element addition.

Output is declared to be a concatenation of all the wire/vars used in the program.

## Step 1: Equivalence Definition

Equivalence is defined as:

*The output wires of the randomly generated verilog program and the netlist synthesized for it should be equal at the clock edge given the same inputs.*

```verilog
module equiv( input clk, input [6:0] w0, input [10:0] w1
            , input signed [10:0] w2, input [11:0] w3 );
  wire [49:0] y1, y2;
  top t1(y1, clk, w0, w1, w2, w3);
  top_synth_netlist t2(y2, clk, w0, w1, w2, w3);
  always @(posedge clk) assert(y1 == y2);
endmodule
```

## Step 2: Equivalence Checking

- Feed it to Yosys.
- Generate two netlists for each.
- Pass it to SMT solver.
- If it returns some result do what is needed.
- If it timeouts cant do a thing.

## Step 1: Reduction Strategy

Every reduction step does the following:

- Randomly choose half the modules from the code and remove them.
- Following this, remove half the items of remaining modules (var/nets).
- Finally, also remove half the statements(assignments/conditionals) from blocks of code.
- Lastly, half the expressions everywhere.

## Step 2: Reduction halt

- Reduction is done in a binary fashion.
- Rerun the equivalence checking after every reduction step.
- The reduction process is halted when the program no longer produces a bug (which bug?).

## Step 3: Additional optimization to reduction process

Notice that

- Output is a concatenation of all the vars/nets assigned.
- The binary search could be done using only the output.
- Remove half the var/nets associated with output, followed by removing code associated with them.

## What do we want to evaluate

Evaluation is to answer 5 key questions.

- How many unique bugs were detected?
- Does bug finding become better as test code size increases?
- How does Xor-ing all outputs affect the bug finding process?
- How is the stability of synthesis tools across different release versions?
- How does the reduction algorithm of verismith fare with that of Csmith?

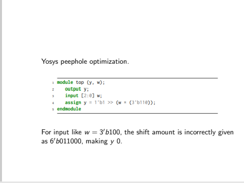
We will look at the first 3 in this presentation.

## Unique Bug 1

Yosys peephole optimization.

```verilog
module top (y, w);
   output y;
   input [2:0] w;
   assign y = 1'b1 >> (w * (3'b110));
endmodule
```

For input like $w = 3'b100$, the shift amount is incorrectly given as $6'b011000$, making $y$ 0.


Yosys peephole optimization.

```
1 module top (y, w);
2    output y;
3    input [2:0] w;
4    assign y = 1'b1 >> (w * (3'b110));
5 endmodule
```

For input like $w = 3'b100$, the shift amount is incorrectly given as $6'b011000$, making $y$ 0.

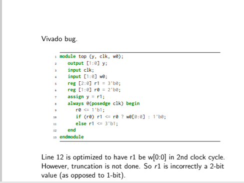
- Line 4 is optimized noting that the shift and product contains a constant.
- However, proper truncation was not done (note that the product is between 3 bit binaries)
- So the program after optimization, for an input like $w = 3'b100$ gave output of $y$ as 0, instead of 1.

## Unique Bug 2

Vivado bug.

```
1  module top (y, clk, w0);
2     output [1:0] y;
3     input clk;
4     input [1:0] w0;
5     reg [2:0] r1 = 3'b0;
6     reg [1:0] r0 = 2'b0;
7     assign y = r1;
8     always @(posedge clk) begin
9        r0 <= 1'b1;
10       if (r0) r1 <= r0 ? w0[0:0] : 1'b0;
11       else r1 <= 3'b1;
12    end
13 endmodule
```

Line 12 is optimized to have r1 be w[0:0] in 2nd clock cycle. However, truncation is not done. So *r*1 is incorrectly a 2-bit value (as opposed to 1-bit).



Vivado bug.

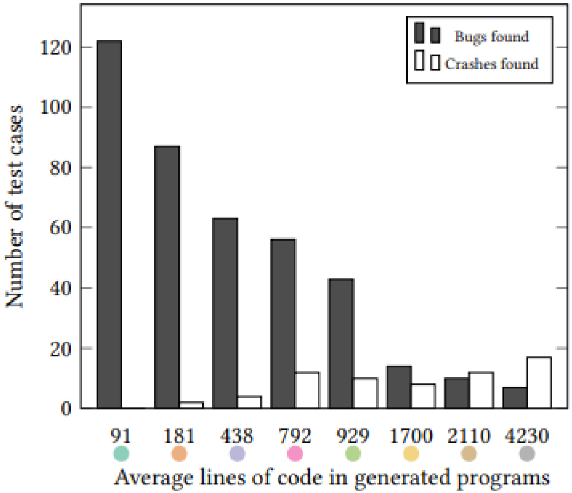
```
1  module top (y, clk, w0);
2     output [1:0] y;
3     input clk;
4     input [1:0] w0;
5     reg [2:0] r1 = 3'b0;
6     reg [1:0] r0 = 2'b0;
7     assign y = r1;
8     always @(posedge clk) begin
9        r0 <= 1'b1;
10       if (r0) r1 <= r0 ? w0[0:0] : 1'b0;
11       else r1 <= 3'b1;
12    end
13 endmodule
```

Line 12 is optimized to have r1 be w[0:0] in 2nd clock cycle. However, truncation is not done. So $r1$ is incorrectly a 2-bit value (as opposed to 1-bit).

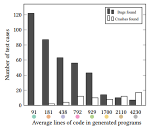
- Note that in first clock cycle, Line 11 sets r0 to 1.
- Now in the second Clock cycle, Line 12 will always set r1 to w[0:0].
- However, this optimization forgets truncation, thus assigning r1 to w.
- So when for instance $w = 2'b10$, $r1$ is incorrectly $2'b10$ instead of $1'b0$.

# Code size ?? Bug finding

Code size vs bug finding, stats.



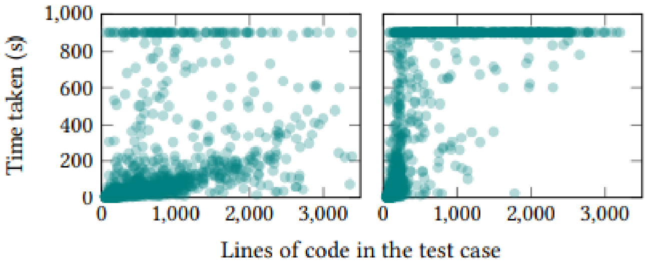
Code size vs bug finding, stats.



- The colums from left to right represent how many bugs found for each configuration of verismith.
- Configuration was in terms of how many lines of code a test case must contain (with the leftmost being the least).
- Interesting to note that the fewer lines of code, the more bugs were uncovered.
- However, the bigger the lines of code, the more crashes were identified.

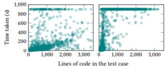
# Xor-ing

Stats.



Bug finding time worse if we Xor outputs. Counter intuitive.

Stats.



Bug finding time worse if we Xor outputs. Counter intuitive.

- Each chart represents how much time the equivalence check took given lines of code.
- The left chart is that when output is a concatenation.
- Whereas the right is that when it is a XOR.
- Interesting to note that when the output is XOR, the equivalence checking takes way longer and 1/3rd of the time times out.

## Pros?

- Bugs exist in HLS tools too. :)
- Reduction of programs that crash HLS tools.
- Exposing that optimization is also a problem for correctness at synthesis level.
- Equivalence definition using output as a concatenation of vars/nets assigned.

## Cons?

- Cannot perform syntax and semantic parts of HLS bugs.
- Equivalence checking done using an HLS tool itself (Yosys).
- Reduction process halves modules/outputs. Perhaps a better way is to build a dependency tree of all vars/nets and remove mostly connected nodes.
- Does not work for Verilog code based on undefined values. Perhaps the optimizations should not rely on undefined values to be aggressive like C11.

## Thank you!

Questions?