Finding and Understanding Bugs in FPGA Synthesis Tools
Akshay G
Intro
Bugs
Bug Detection
Result
Main
Code Generation
Equivalence
Reduction
Evaluation
Discussion
Pros
Cons

Finding and Understanding Bugs in FPGA Synthesis Tools

Yann Herklotz, John Wickerson

Akshay G

February 10, 2022

Who are the authors?

Yann Herklotz - PhD student supervised by John Wickerson, Imperial College London.

John Wickerson - Lecturer at Dept of Electrical and Electronic Engineering, Imperial College London.

Introduction: questions

Finding and Understanding Bugs in FPGA Synthesis Tools

Akshay G

Intro
Bugs
Bug Detection
Result
Main
Code Generation
Equivalence
Reduction
Evaluation
Discussion
Pros
Cons

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

What bugs?

- Programs are ultimately run by the hardware.
- Circuits are designed today using similar high-level program constructs.
- High level synthesis(HLS) tools exist for this purpose.
- If the HLS tools are not designed correctly, can we trust the hardware on which our programs run?

What sort of bugs are addressed?

- Optimizations in HLS 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.

How are the bugs detected?

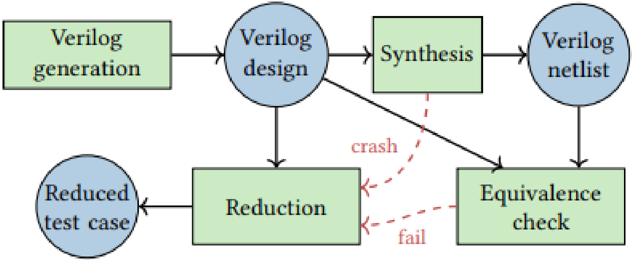
Finding and Understanding Bugs in FPGA Synthesis Tools

Akshay G

Intro
- Bugs
- Bug Detection
- Result

Main
- Code Generation
- Equivalence
- Reduction
- Evaluation

Discussion
- Pros
- Cons



What results were obtained?

- Four tools were tested this way: Yosys, Vivado, XST and Quartus Prime.
- Except Quartus prime(paid version), 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..9).
- 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.

```
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 (HLS tool).
- Pass it to SMT solver or ABC tool.
- If it returns some result:
  - If a counter example, pass it to reduction phase.
  - If no counter example, continue to next verilog design.

.

- If it timeouts cant do a thing.

Step 1: Reduction Strategy

Intro
Bugs
Bug Detection
Result
Main
Code Generation
Equivalence
Reduction
Evaluation
Discussion
Pros
Cons

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.

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.

```
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.

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 10 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).

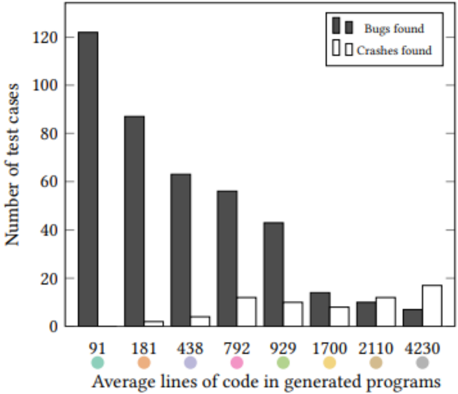
Summary of Testing

Tool	Total test cases	Failing test cases	Distinct failing test cases	Bug reports
Yosys 0.8	26400	7164 (27.1%)	≥ 1	0
Yosys 3333e00	51000	7224 (14.2%)	≥ 4	3
Yosys 70d0f38 (crash)	11	1 (9.09%)	≥ 1	1
Yosys 0.9	26400	611 (2.31%)	≥ 1	1
Vivado 18.2	47992	1134 (2.36%)	≥ 5	3
Vivado 18.2 (crash)	47992	566 (1.18%)	5	2
XST 14.7	47992	539 (1.12%)	≥ 2	0
Quartus Prime 19.2	80300	0 (0%)	0	0
Quartus Prime Lite 19.1	43	17 (39.5%)	1	0
Quartus Prime Lite 19.1 (No `$signed`)	137	0 (0%)	0	0
Icarus Verilog 10.3	26400	616 (2.33%)	≥ 1	1

Table 2: Summary of failing test cases found in each tool that was tested.

Code size ?? Bug finding

Code size vs bug finding, stats.



Xor-ing

Intro
Bugs
Bug Detection
Result
Main
Code Generation
Equivalence
Reduction
Evaluation
Discussion
Pros
Cons

Stats.

Figure 6: Synthesis and equivalence checking time as program size increases for test cases. Left: output combined using concatenation. Right: output combined to one bit using unary XOR operator.

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

Intro
Bugs
Bug Detection
Result
Main
Code Generation
Equivalence
Reduction
Evaluation
Discussion
Pros
Cons

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 identify bugs at syntax and semantic parts of HLS.
- Reduction process halves modules/outputs.
- Does not work for Verilog code based on undefined values.
- Bug examples in paper seem to only circulate around incorrect assignment of array widths after operations.

Thank you!

Intro
Bugs
Bug Detection
Result
Main
Code Generation
Equivalence
Reduction
Evaluation
Discussion
Pros
Cons

Questions?