

What
interactive theorem proving
can do for
**Verilog hardware
development**

Andreas Löow

Imperial College London

Summary

Claim 1:

Today you can do ITP-based development for software, **this is useful**

Claim 2:

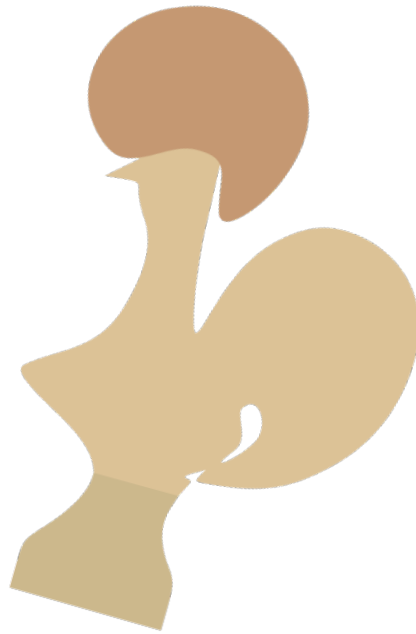
If we had ITP-based development for hardware,
it would be **equally useful for hardware**

... this talk is about adapting a development methodology from ITP-based software development for Verilog hardware development

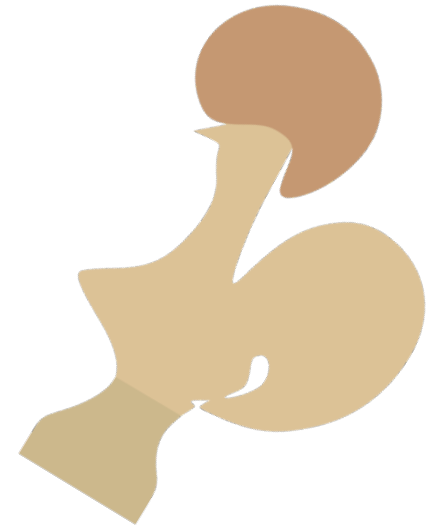
Some ITPs/proof-assistants

Examples:

- Coq
- Isabelle/HOL
- HOL4
- Lean
- ACL2
- ...



Question: Why do **ITP-based development** instead of, e.g.:
pen-and-paper mathematics,
fully automated theorem proving,
or something else?

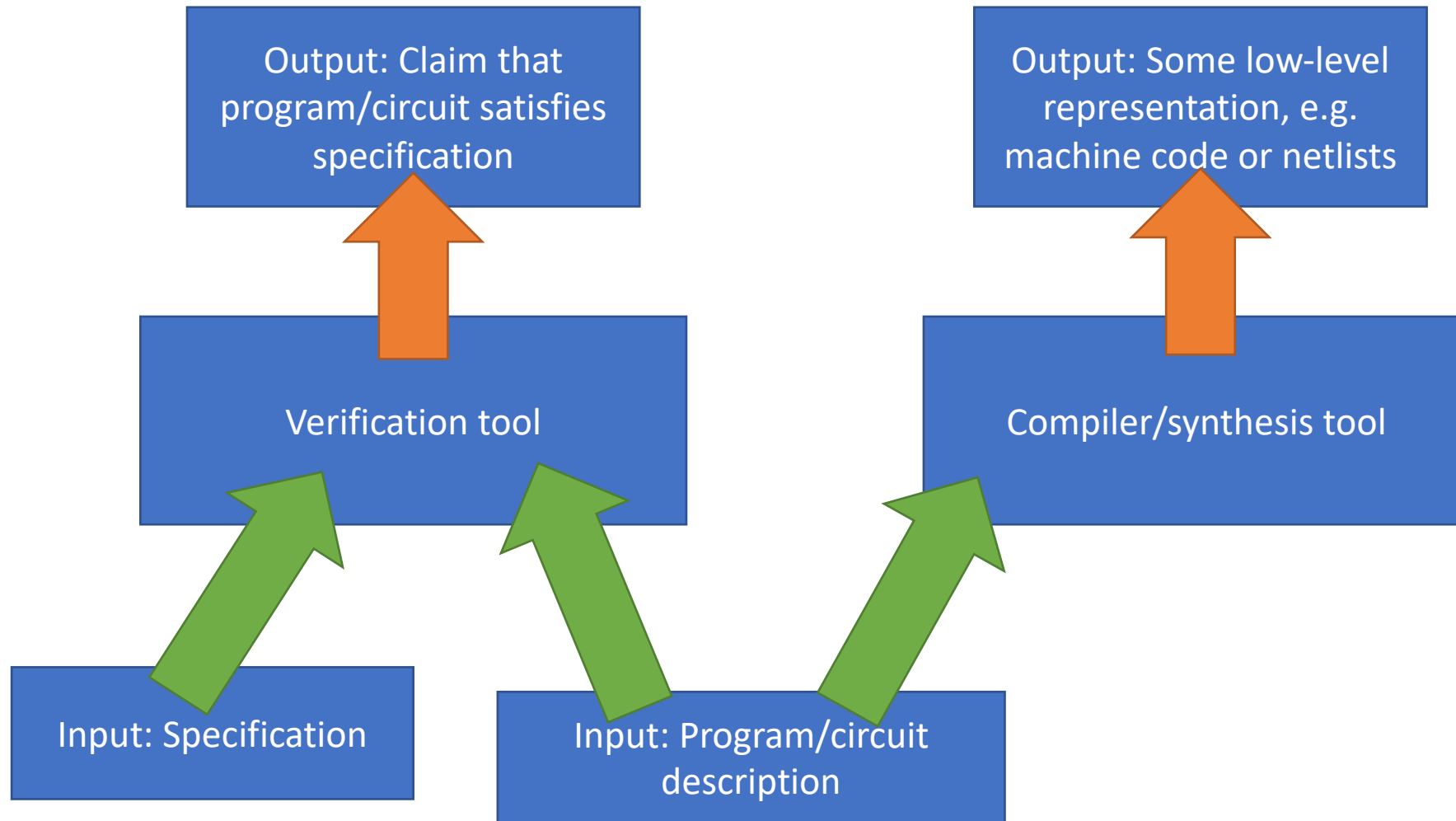


ITP-based development – why

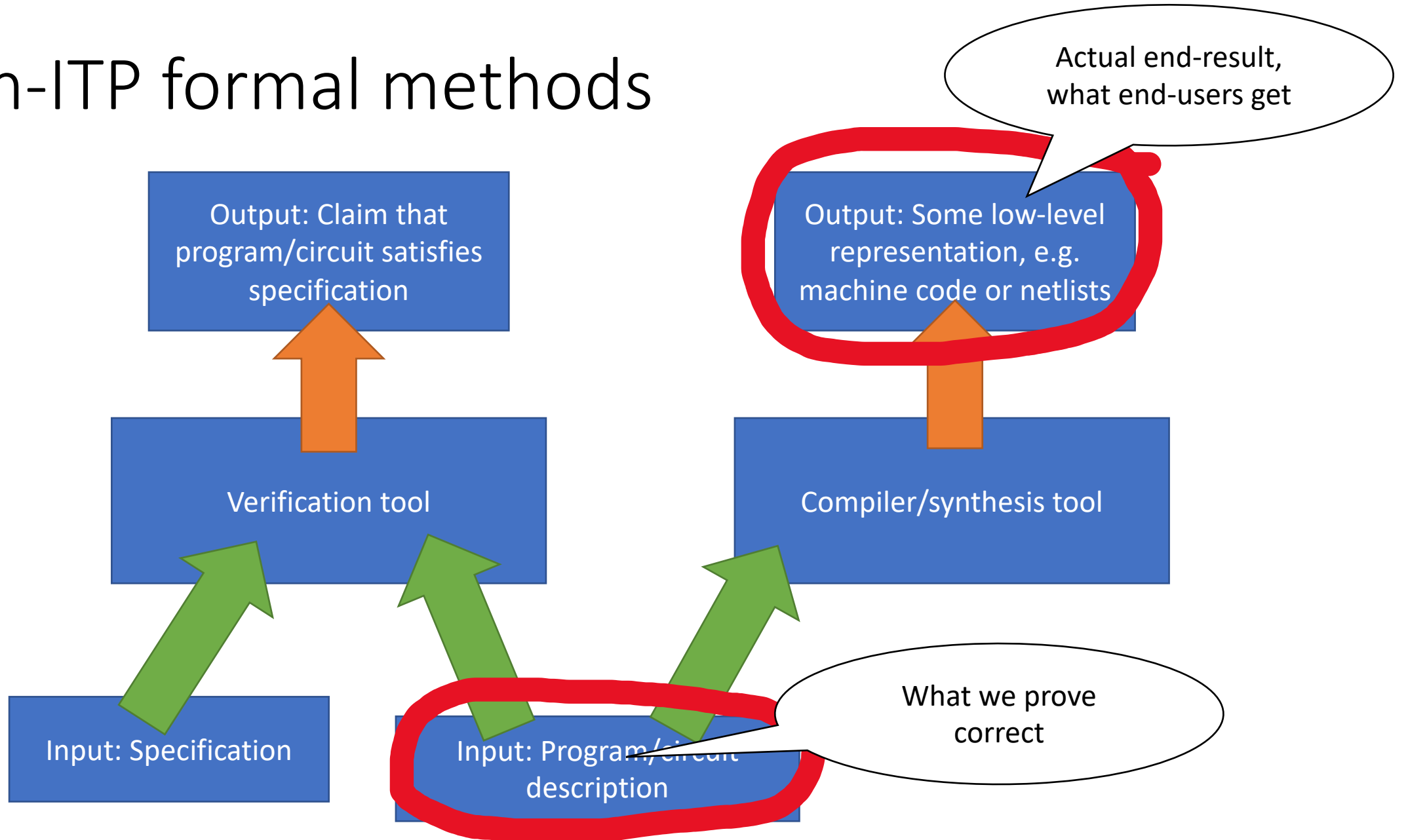
Main point of ITP:

- **Trustworthy proofs**, checked by small program (“kernel”)
- Allows for **combining human and machine reasoning** – get the strengths of both, avoid the weaknesses of both
- Allows you to check/prove that **large developments “fit together”**

Non-ITP formal methods



Non-ITP formal methods



Non-ITP formal methods

Output: Claim that program/circuit satisfies specification

Output: Some low-level representation, e.g. code or netlists

If only there were a way to ensure that they compose well... 🤔

Verification tool

Compiler/synthesis tool

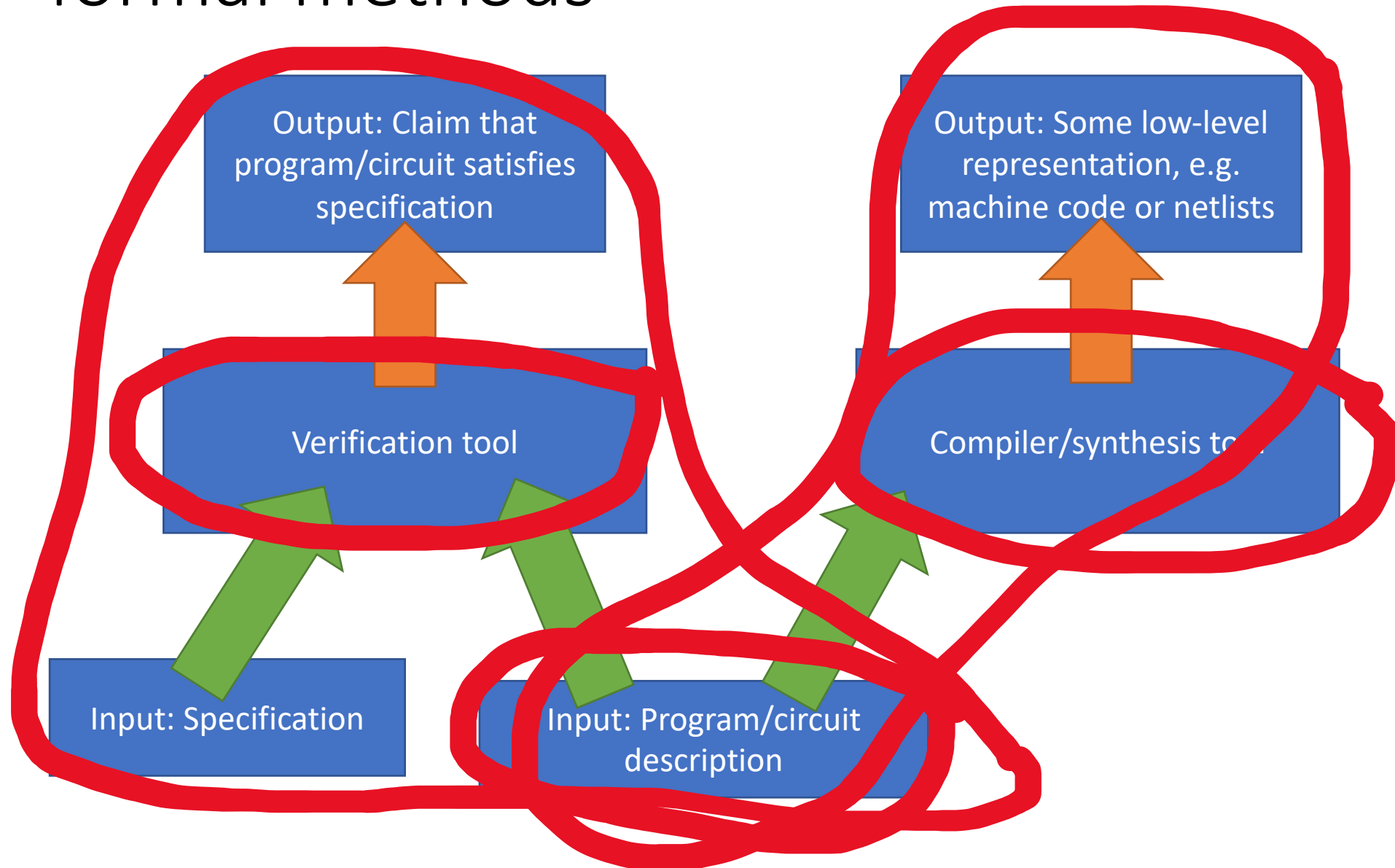
Input: Specification

Input: Program/circuit description

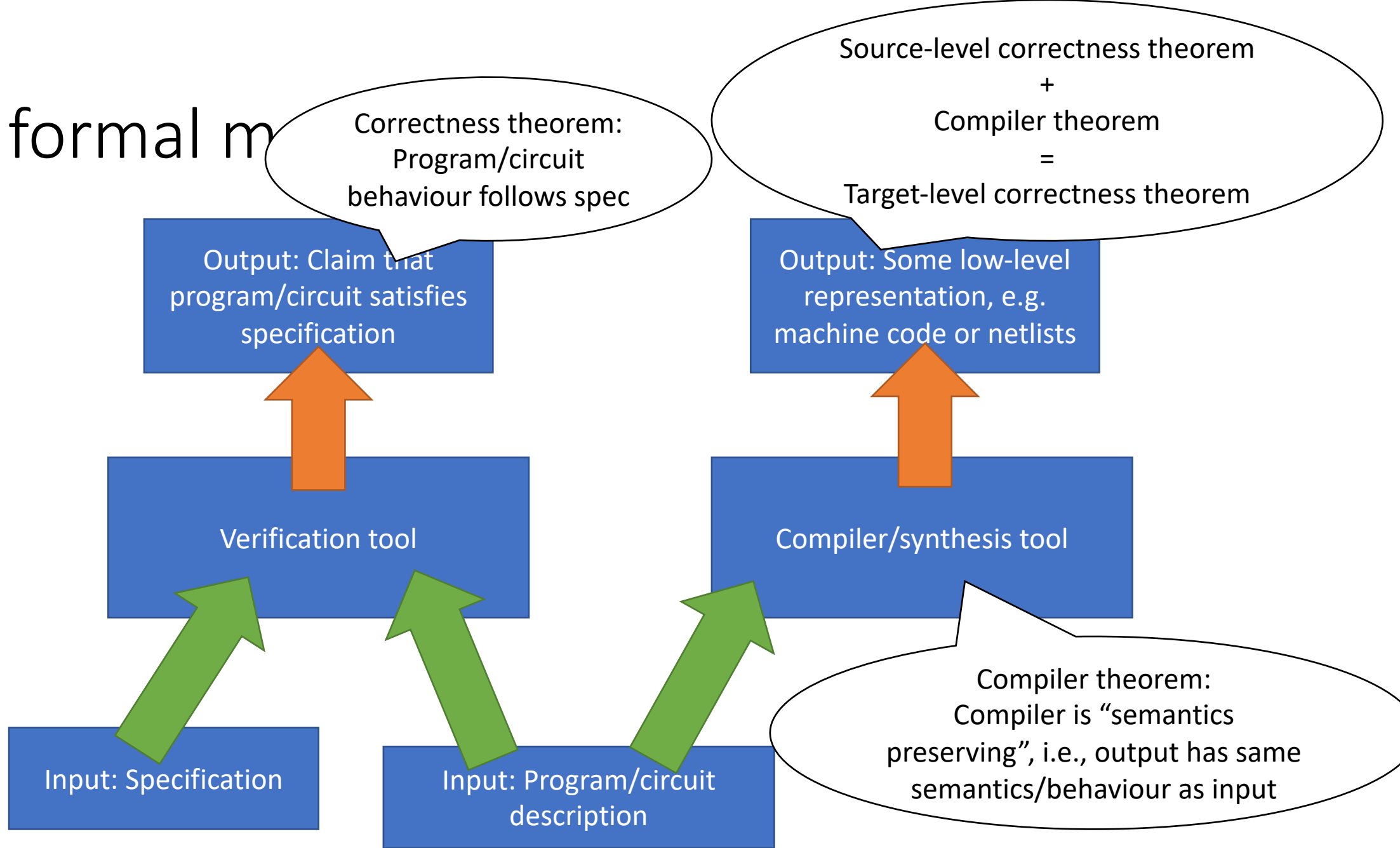
Important: The two tools must “interpret” the implementation language in the same way!



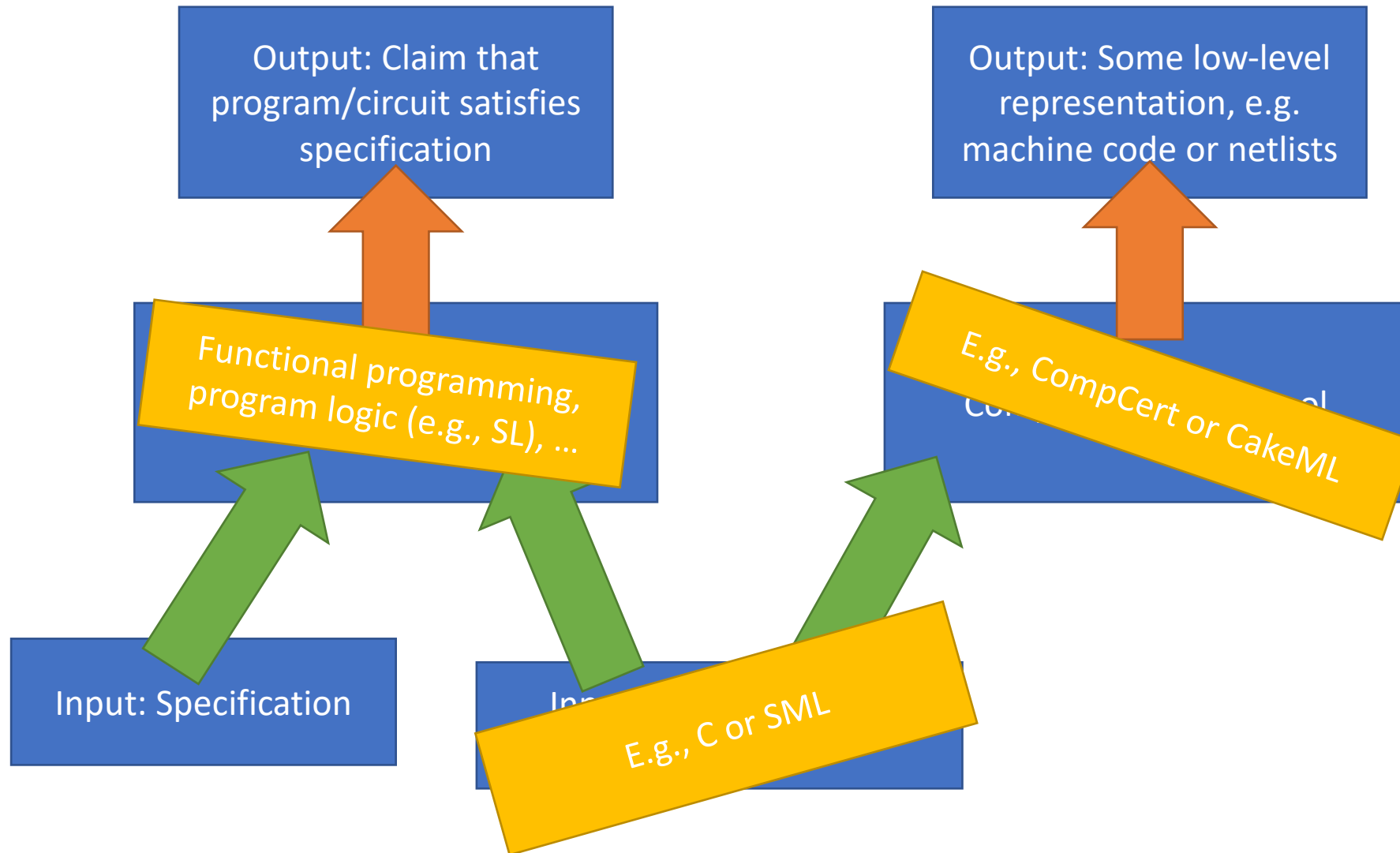
I/P formal methods



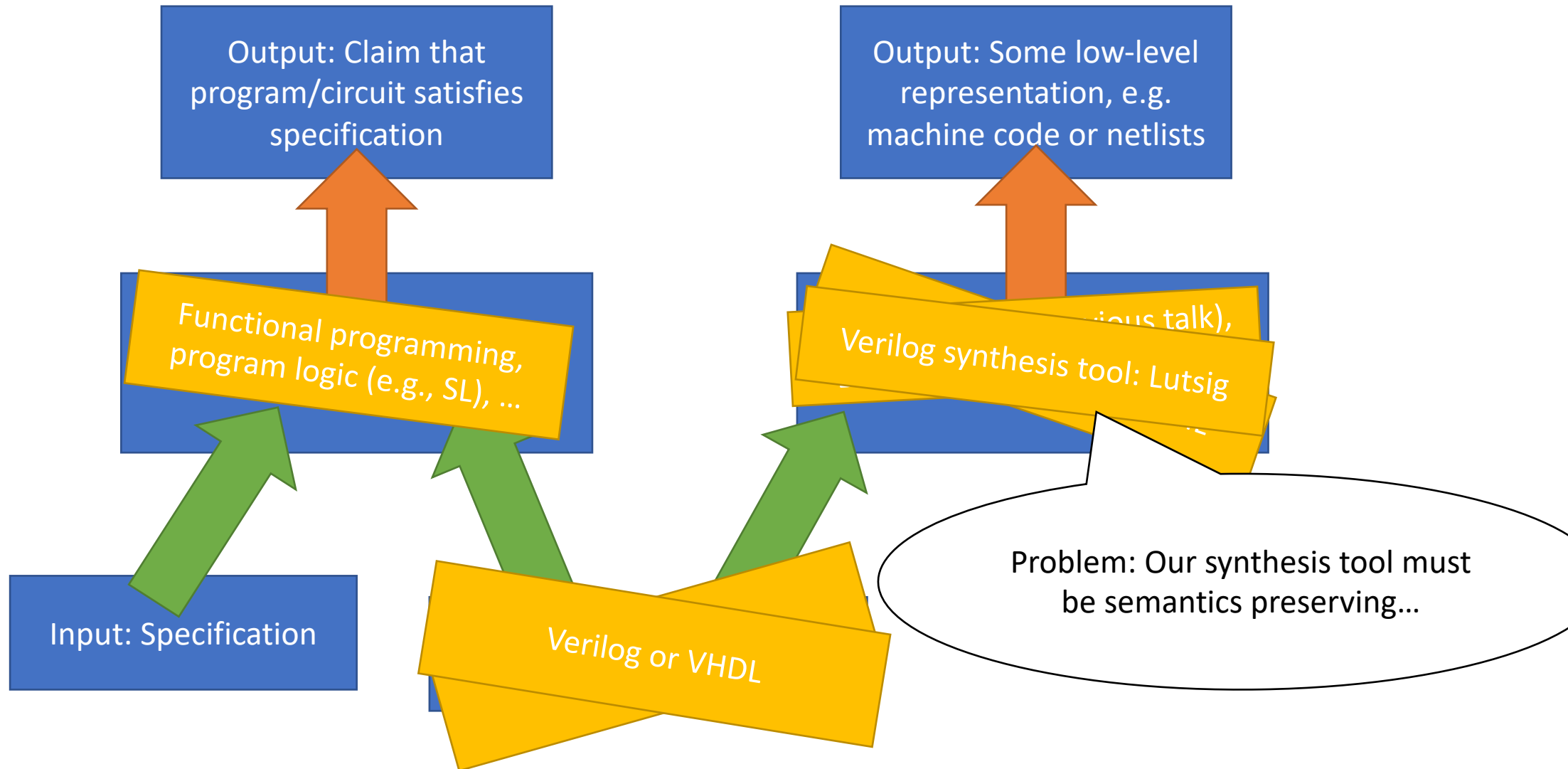
ITP formal m



ITP formal methods: Software



ITP formal methods: Hardware



Verilog



Gisselquist
Technology, LLC

- [Main/Blog](#)
- [About Us](#)
- [FPGA Hell](#)
- [Tutorial](#)
- [Formal training](#)
- [Quizzes](#)
- [Projects](#)
- [Site Index](#)
- [@zipcpu](#)
- [Reddit](#)
- [Support](#)

Reasons Simulation

Aug 4, 2018

When I first learned dig
the hardware and debu

I've since become conv
synthesizing a design. In
Vivado fully starts up and
still faster. Of course, utili
might manage to get an d

The second reason why I
within the design. For this
to simulation and try to do
allows me to be able to tur

Or ... not so quickly. On one
have the design fail when re
with simulation—but then had
drive.

But what happens when you
simulation, but fails on the har

I'll admit this happened to me

Therefore, to help keep you from
simulation not to match reality. When I asked

RTL Coding
Simulation
M

This paper de
and post-syn
silicon has
Each codi
style that
apply to

Stuart Sutherland and Don Mills

Verilog and System Verilog Gotchas

101 Common Coding Errors
and How to Avoid Them

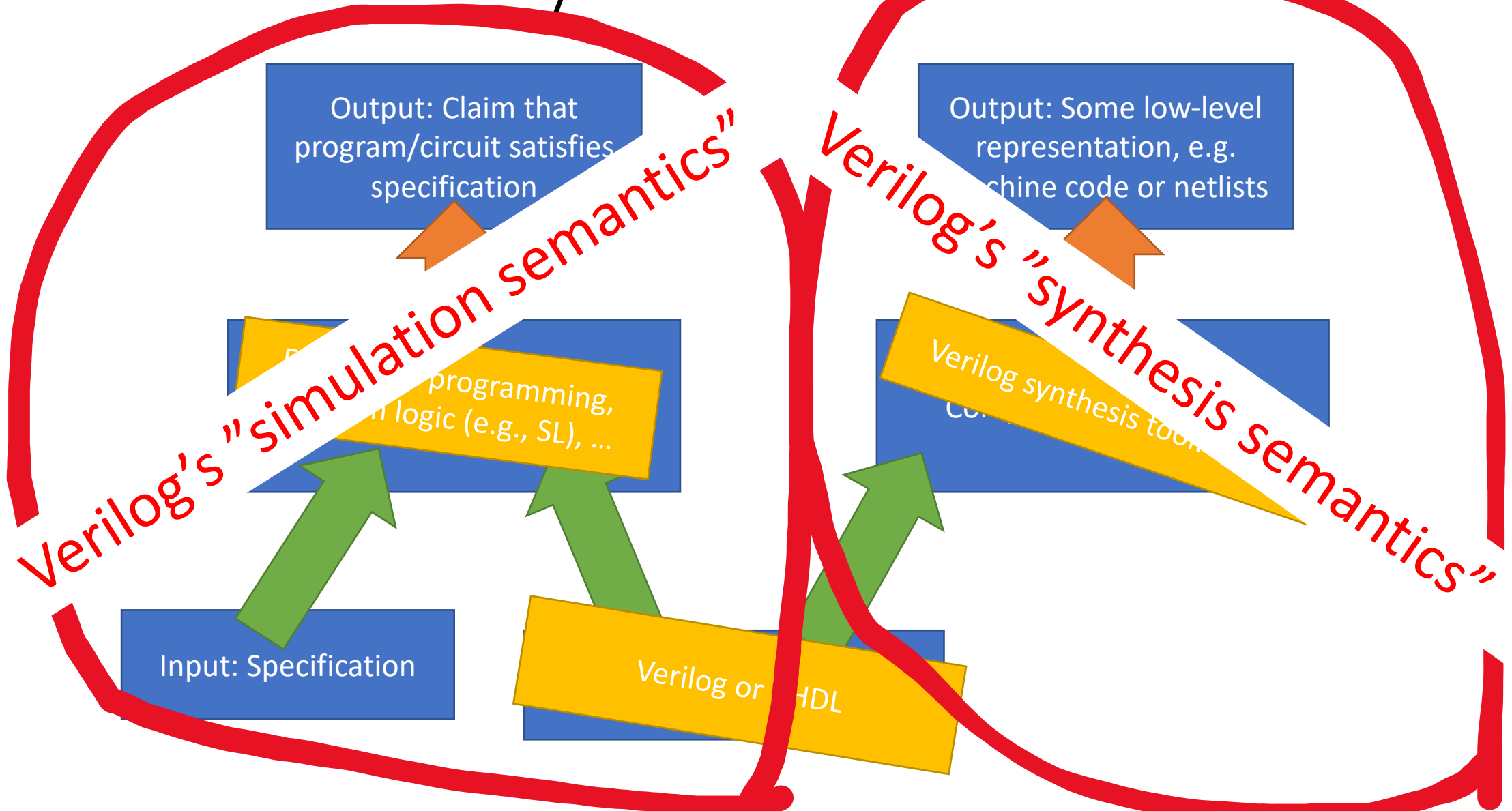


Springer

ABST

The semantics of X in Verilog RTL are extremely dangerous. RTL simulations to incorrectly pass where netlist simulations because formal equivalence checkers are configured to ignore that equivalence checking is fast replacing netlist simulations. This problems in order to raise awareness of X issues in many different often poorly understood by RTL designers and EDA vendors alike. It overcome X issues in new designs (including good coding styles) and technical existing designs (including automated formal proofs). New terminology is in subtle interpretations of X by EDA tools, along with recommendations to avoid this paper describes how to change the default settings of equivalence checkers are otherwise far too sneaky to detect). In short, if you are using EDA tools for simulation coverage, synthesis or equivalence checking, you must be aware of the problems and described in this paper.

Simulation-and-synthesis mismatches



Simulation-and-synthesis mismatches



Verilog

Antics”

Verilog

Antics”

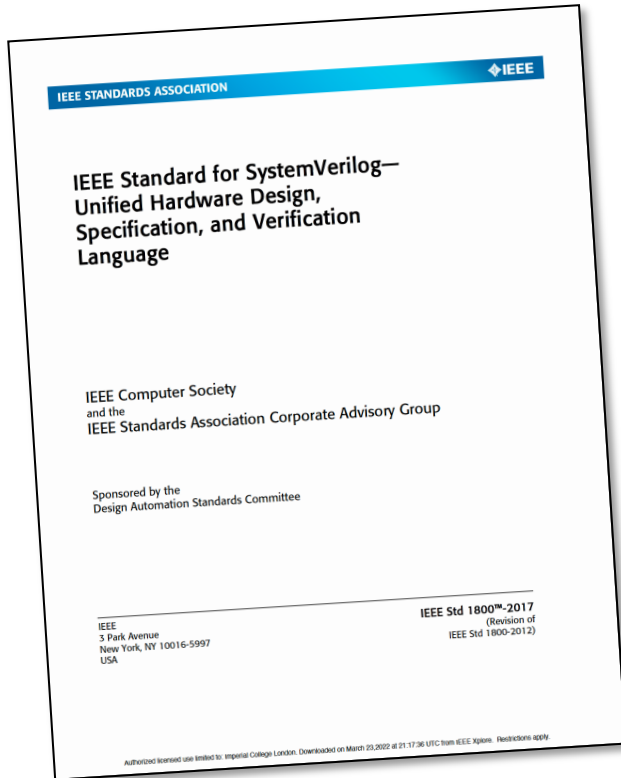
Verilog or HDL

Combinational logic -- mismatch
example and handling it formally
in Lutsig

“Mis-ordered” assignments

B.5 Assignment statements mis-ordered

```
module andor1a(  
    output logic y,  
    input logic a, b, c);  
    logic tmp;  
  
    always_comb begin  
        y = tmp | c;  
        tmp = a & b; // write after read  
    end  
endmodule
```



“Mis-ordered” assignments

Example from the “synthesis standard”

Essentially, a prose-specified event-driven operational semantics

Assignment statements mis-ordered

IEEE STANDARD

IEEE Standard for SystemVerilog—
Unified Hardware Design,
Specification, and Verification
Language

```
module andor1
```

```
output y
```

```
input a, b
```

```
logic tmp
```

```
always_comb begin
```

```
  y = tmp | c;
```

```
  tmp = a & b; // write after read
```

```
end
```

```
endmodule
```

This block induces a software-like thread that will run each time something the block depends on change value

There is an (stratified) event queue, handling of events, etc.

The statements run in the given order

“Mis-ordered” a

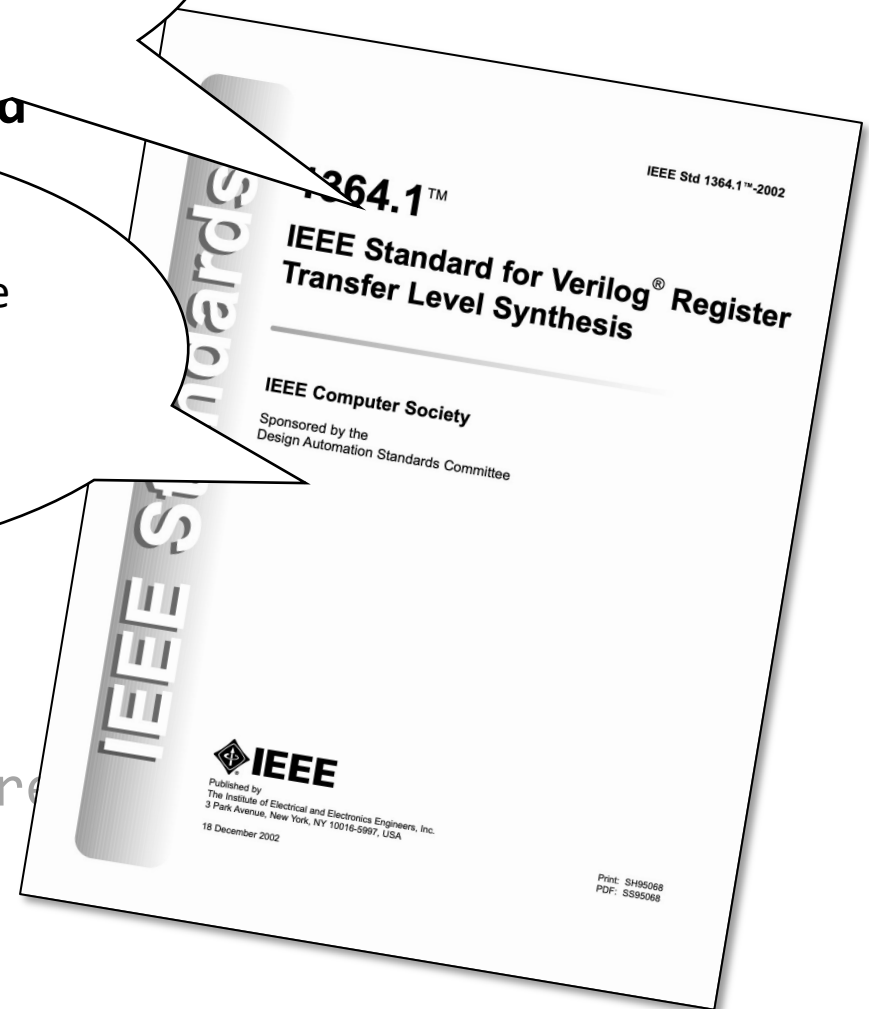
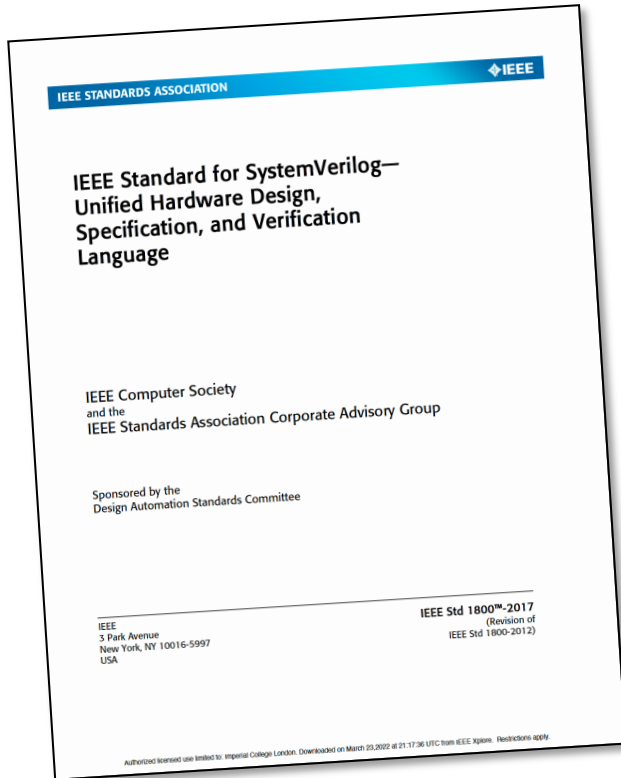
“This standard defines a set of modeling rules for writing Verilog HDL descriptions for synthesis.”

B.5 Assignment statement

```
module a  
  output  
  input  
  logic tmp,
```

“Combinational logic shall be modeled using [...] or an always statement.”

```
always_comb begin  
  y = tmp | c;  
  tmp = a & b; // write after read  
end  
endmodule
```



“Always” assignment

Describes an event-driven language, could have equally well been a (weird) event-driven software programming language

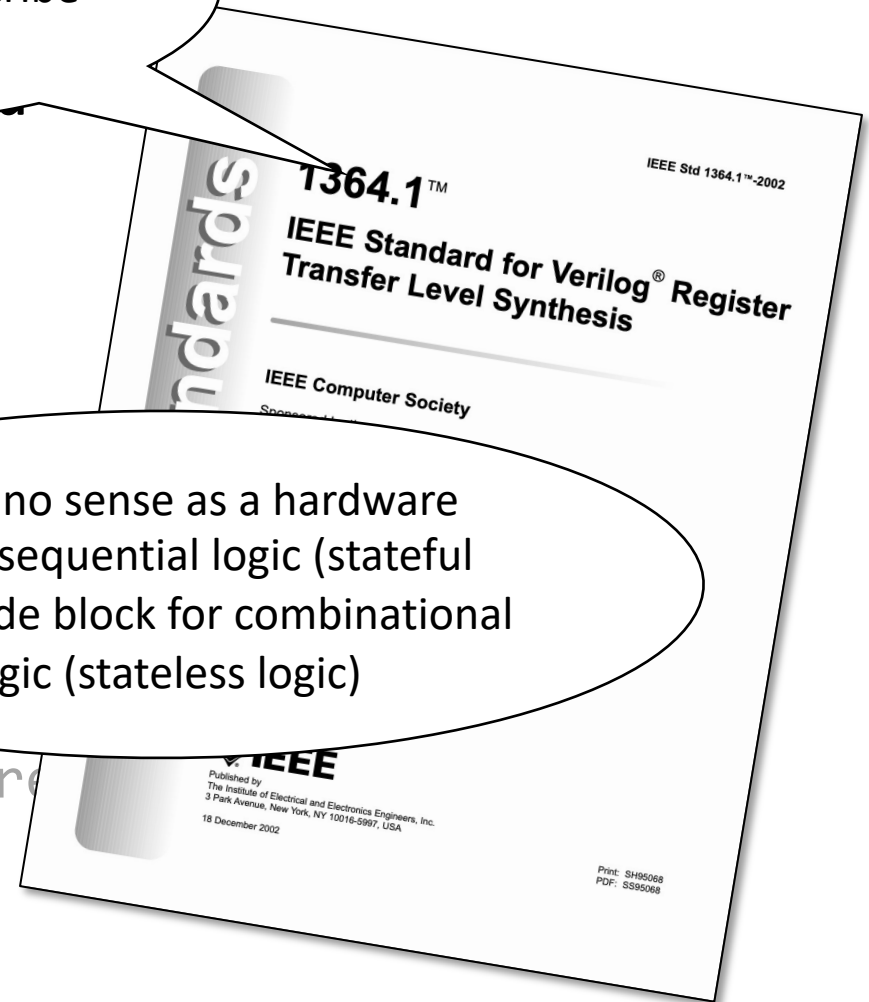
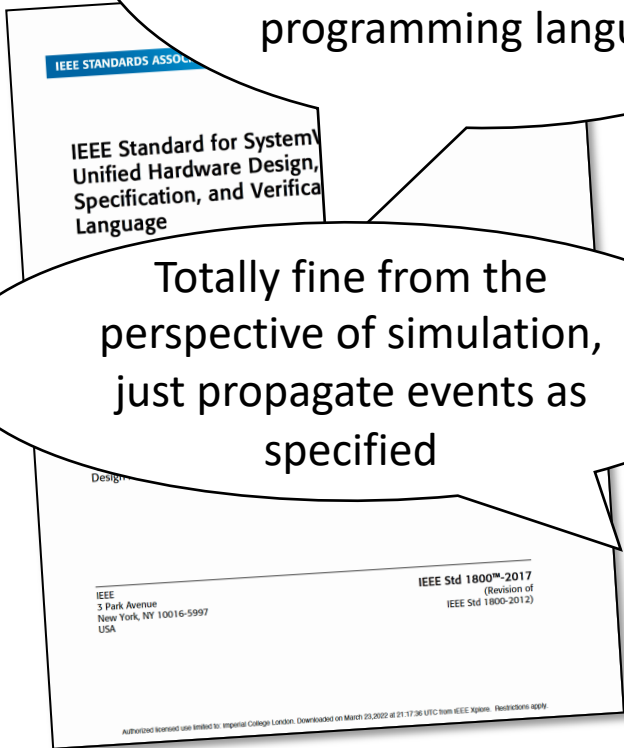
Gives us “modelling rules” for how to model/describe hardware

Assignment statement

```
module andor1a(  
    output logic y,  
    input logic a, b, c);  
    logic tmp;  
  
    always_comb begin  
        y = tmp | c;  
        tmp = a & b; // write after read  
    end  
endmodule
```

Totally fine from the perspective of simulation, just propagate events as specified

Makes no sense as a hardware model, sequential logic (stateful logic) inside block for combinational logic (stateless logic)



What happens when you give today's synthesis tools a problematic design?

Basically anything, today's synthesis tools might:

- abort (good case)
- emit warnings (borderline case)
- silently synthesise nonsense (bad case)

Such synthesis tools are **not** semantics preserving, i.e., this is bad

Lutsig – a verified Verilog synthesis tool

- Developed and verified inside the HOL4 interactive theorem prover
- Designed to fit into ITP-based hardware development
- Specifically, semantics preserving
- (Can be used outside formal development as well, like any other synthesis tool.)

Lutsig – a verified Verilog synthesis tool

- Handles a small synthesisable subset of Verilog for synchronous designs
- Targets FPGAs:
 - Verified synthesis algorithm, based on open source CSYN synthesis tool
 - Translation-validation-based technology-mapping algorithm for FPGAs (LUTs)
 - Remaining steps outside formal development (e.g., P&R, bitstream encoding)

Lutsig's correctness theorems (simplified)

Correctness w.r.t. (Lutsig's) Verilog simulation semantics:

$Lutsig(D) = OK(N) \implies \text{forall } n, \text{run_verilog}(D, n) = \text{run_netlist}(N, n)$

(except for X-related behavior, which is allowed to be removed)

Correctness w.r.t. synthesis idiom for `always_comb`:

$Lutsig(D) = OK(N) \implies$

forall Verilog variables v in D ,
if v written to by `always_comb` block \implies
no register with name v in netlist N

Lutsig vs. today's synthesis tools

- Design your Verilog module using the old familiar synthesis idioms
- If Lutsig successfully gives back a synthesised netlist:
 - because of Lutsig's correctness theorem, the synthesised netlist must have the same behaviour as the input Verilog module
 - i.e., simulation-and-synthesis mismatches are ruled out using mathematical proof
- If Lutsig errors out:
 - revisit your design
 - This happens e.g. when the simulation and synthesis semantics point in different directions, because Lutsig abides by both semantics, Lutsig is forced to abort if this happens

What does Lutsig actually do?

- Sequential blocks (`always_ff`) straightforward to handle, with check that blocking vs. nonblocking assignments are not misused

- Combinational blocks:

- Sort blocks topologically w.r.t. read dependencies, e.g.:

```
always_comb b = a + 1;  
always_comb a = inp;
```

- (Abort if cannot sort.)
 - Examples of individual blocks to follow...

Combinational example 1: Scalars

For straight-line code, read as netlist:

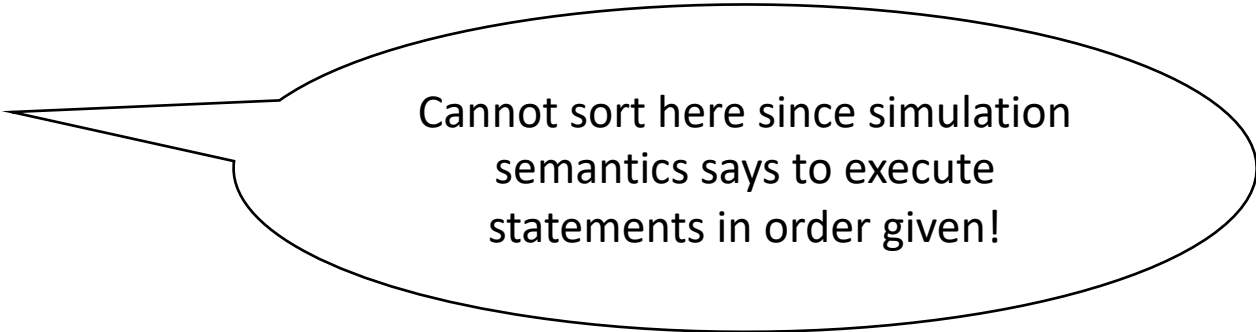
```
always_comb begin
```

```
// Lutsig would die here since tmp  
// read before written to
```

```
y = tmp | c;
```

```
tmp = a & b;
```

```
end
```



Cannot sort here since simulation semantics says to execute statements in order given!

Combinational example 2: Arrays

For straight-line code, read as netlist:

```
logic[1:0] foo;
```

```
always_comb begin
```

```
    foo[0] = inp1;
```

```
    foo[1] = inp2;
```

```
    // ok reading foo here since whole array covered
```

```
    foo = foo + 1;
```

```
end
```

Combinational example 3: If-statements

Generate mux for if-statements, fail if not assigned in all branches:

```
always_comb  
    if (c)  
        a = inp;  
//else  
//    a = 'x;
```

Remember: Lutsig is formally verified

- Previous slides are pretty much the same checks a helpful synthesis tool or a linter would do
- Lutsig, however, is formally verified
- So, we know that the checks done are **sufficient to *guarantee* semantics-preserving synthesis**, i.e., input Verilog module and output netlist behave the same

To do: Consider more sources of simulation-and-synthesis mismatches

- Obviously, need to go through the same process for other sources of simulation-and-synthesis mismatches
- SystemVerilog solves some things: e.g. incomplete sensitivity lists
- Some things should just be prohibited: e.g. delays
- Sometimes the simulation model is slightly off, e.g. dual-port block RAM

Conclusion

- Clearly, we want to do development inside ITPs
- Verilog is a... tricky language
- Lutsig is one attempt at doing formal hardware development using Verilog nevertheless
- Verilog is the most popular HDL, so it's doing *something* right