Reconciling Verified-Circuit Development and Verilog Development

Andreas Lööw

Imperial College London

One-slide summary

Well-known: verified-program development inside ITP

+

One-slide summary

Well-known: verified-program development inside ITP

+

Well-known: Verilog development

=

One-slide summary

Well-known: verified-program development inside ITP

+

Well-known: Verilog development

=

One-slide summary

Well-known: verified-program development inside ITP

+

Well-known: Verilog development

=

One-slide summary

Well-known: verified-program development inside ITP

+

Well-known: Verilog development

=

Additionally, goal here: Want to change these two as little as possible while doing this addition

What's the problem?

Well-known: verified-program development inside ITP

+

Well-known: Verilog development

=

What's the problem?

Well-known: verified-program development inside ITP

+

Well-known: Verilog development

=

Non-semantics-preservation in compilation = bug

What's the problem?

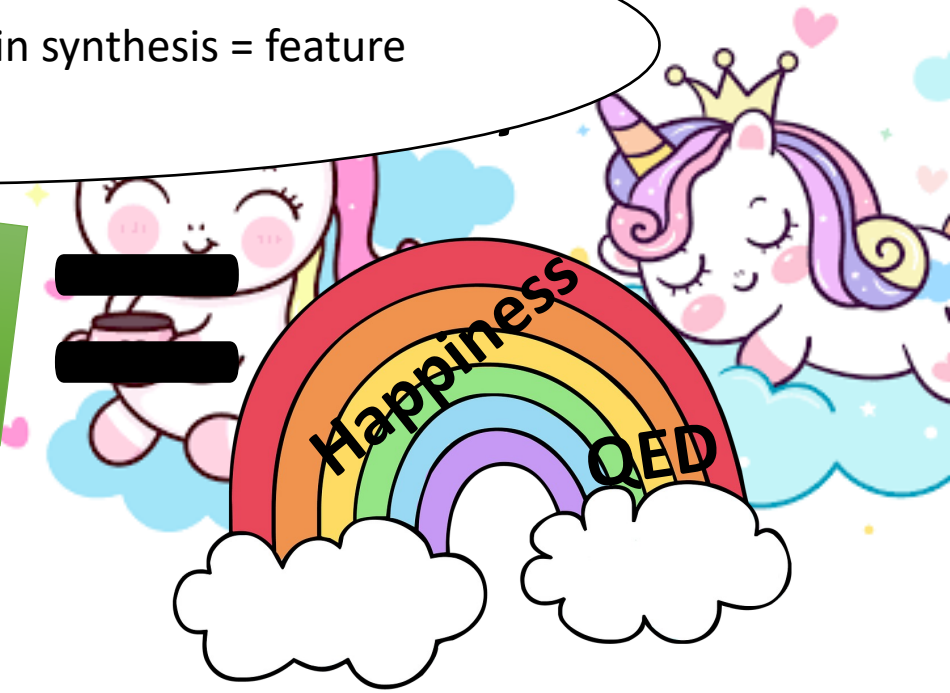
Non-semantics-preservation in synthesis = feature

Well-known: verified-program development inside ITP

+

Well-known: Verilog development

=



Non-semantics-preservation in compilation = bug

This talk

Address some of Verilog’s quirks in the process of extending the Verilog support of the verified Verilog synthesis tool Lutsig (and associated tools)

What is Lutsig?

Lutsig – a verified Verilog synthesis tool

- Developed and verified inside the HOL4 interactive theorem prover (first version published at CPP’21)
- Handles a small synthesisable subset of Verilog for synchronous designs
- Currently targets FPGAs:
  - Verified synthesis algorithm
  - Translation-validation-based technology-mapping algorithm for FPGAs (LUTs)

What do I mean by verified-program/verified-circuit development?

Development flow

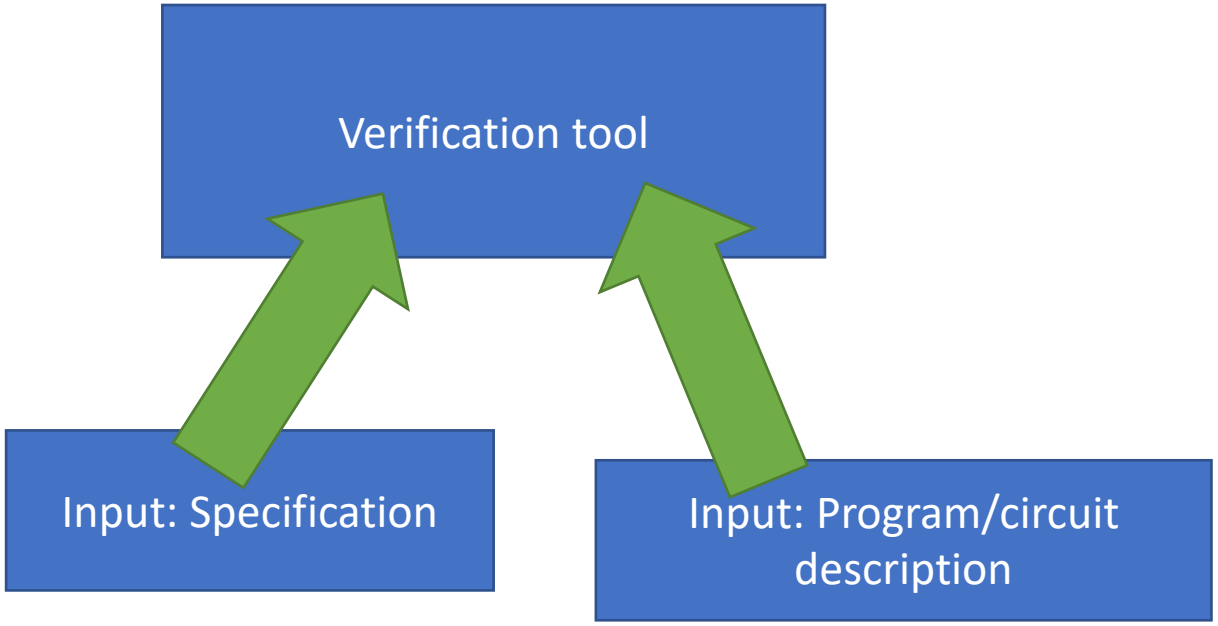
Input: Program/circuit description

Development flow

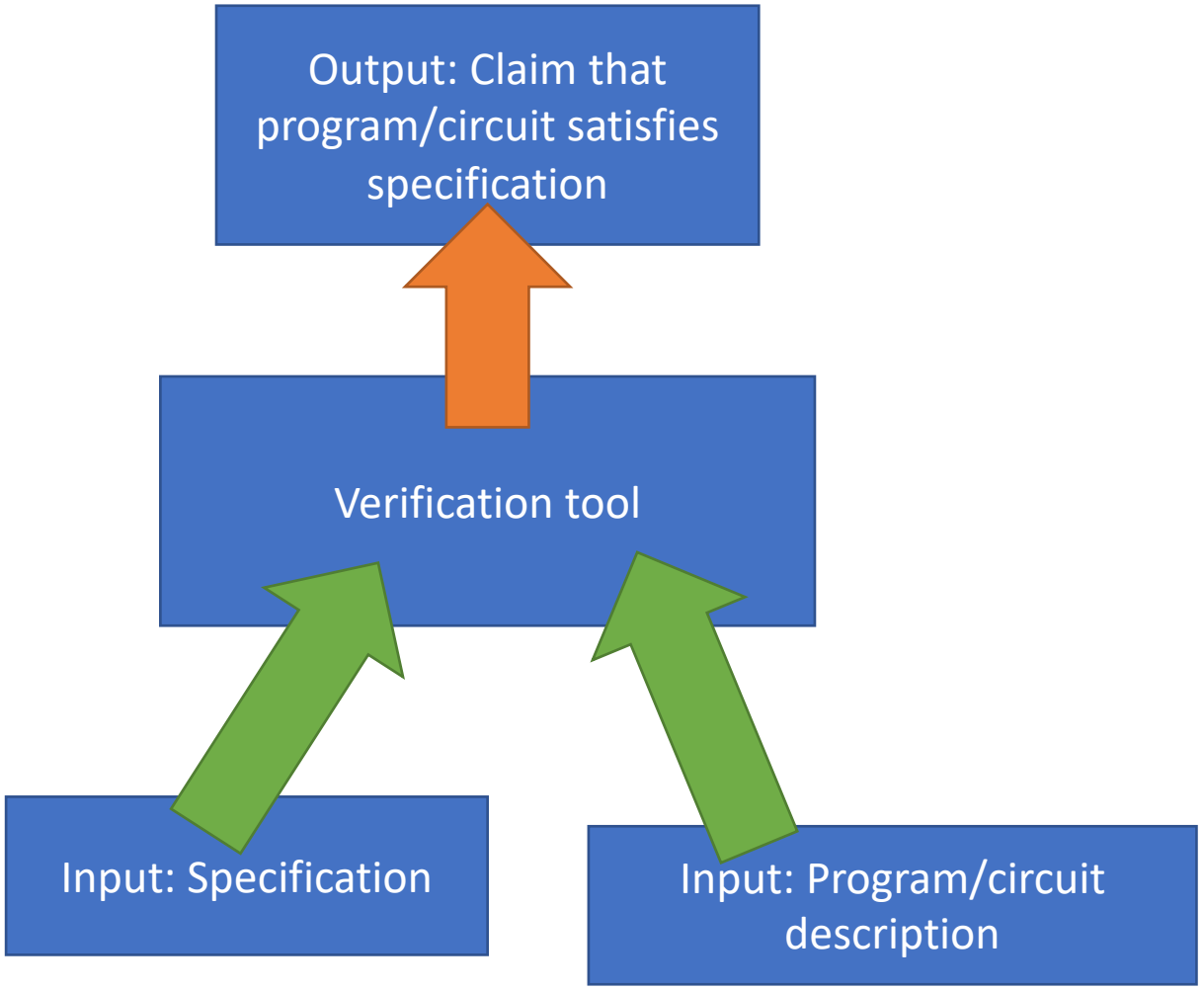
Input: Specification

Input: Program/circuit description

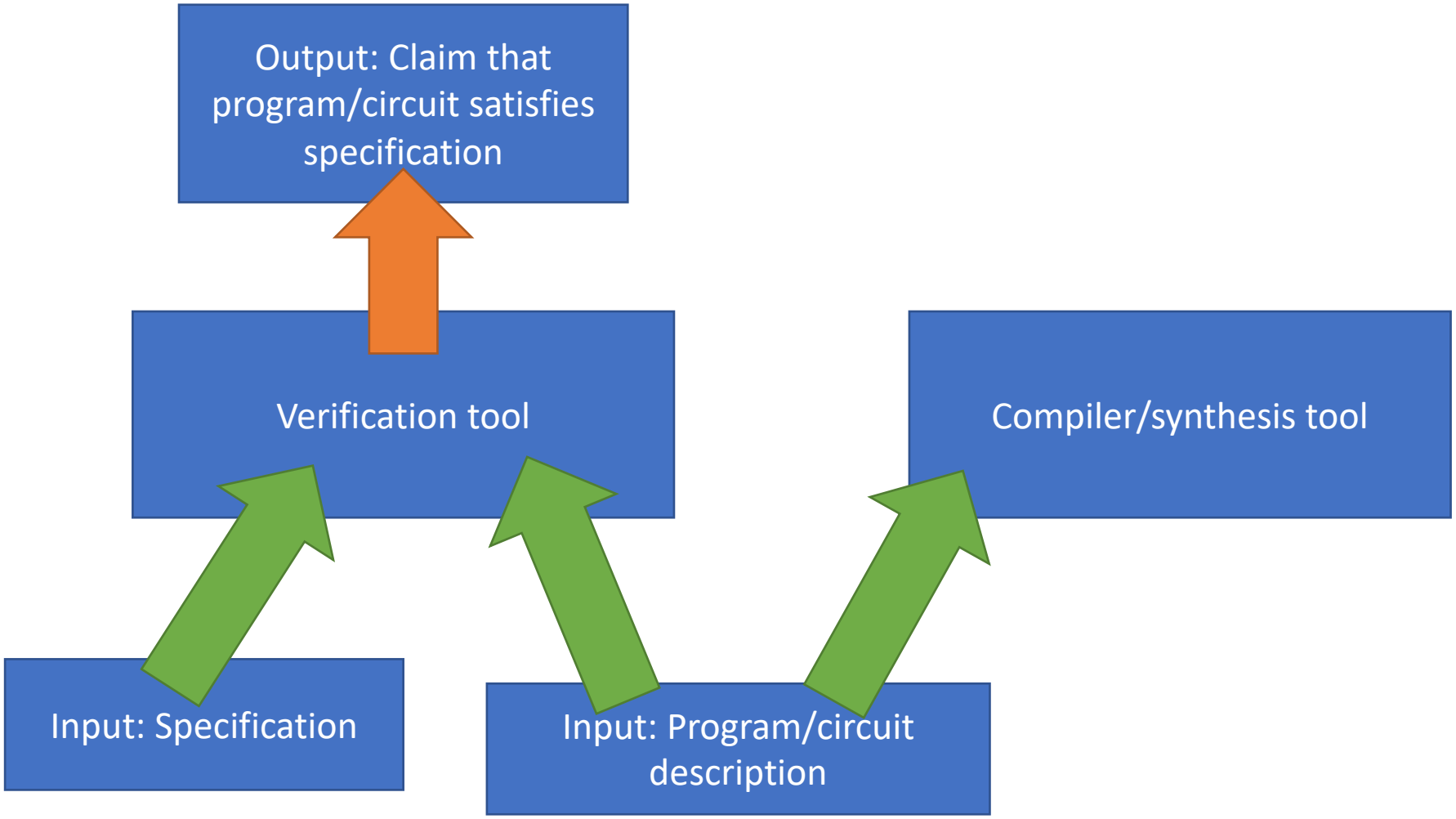
Development flow



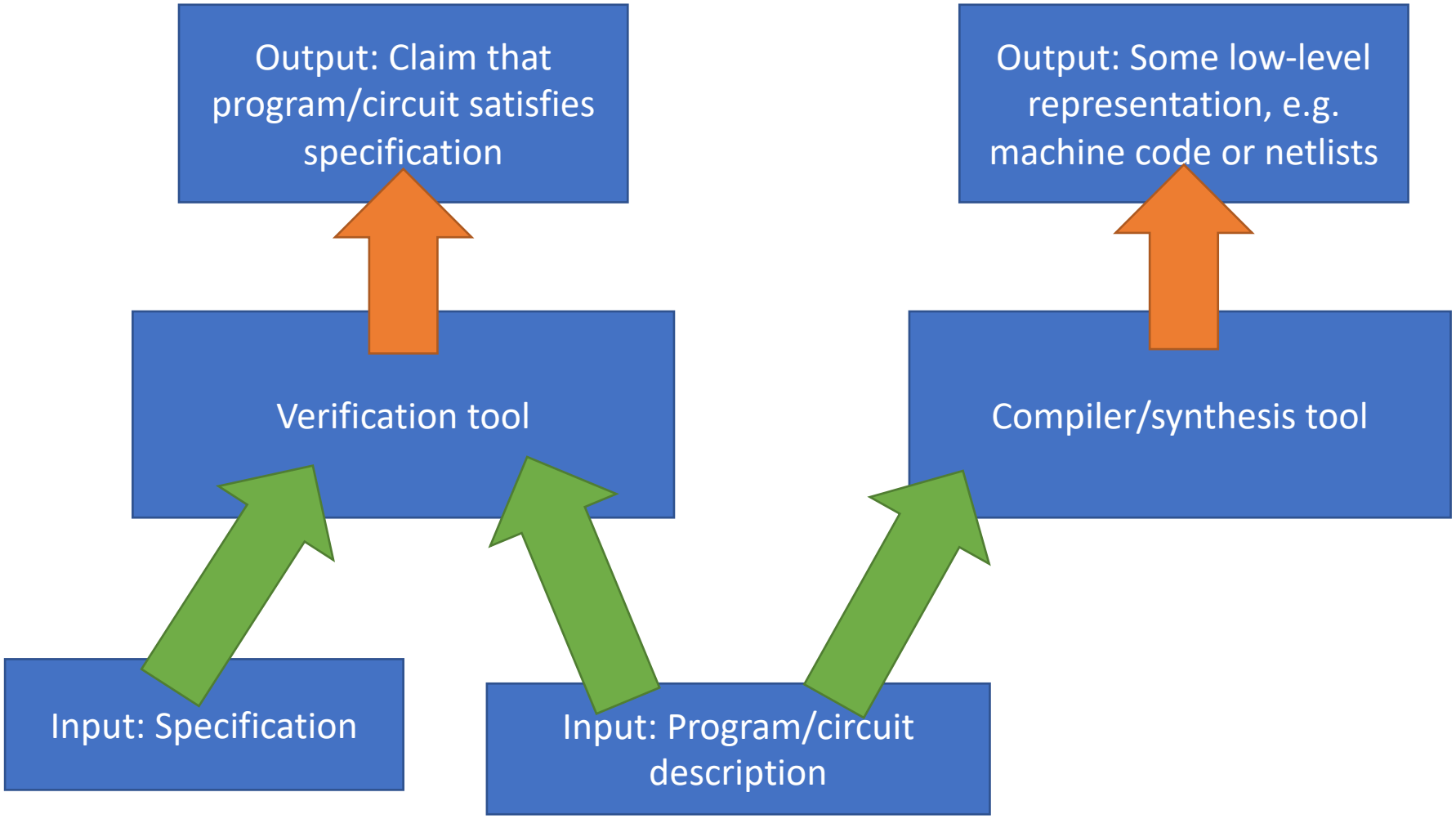
Development flow



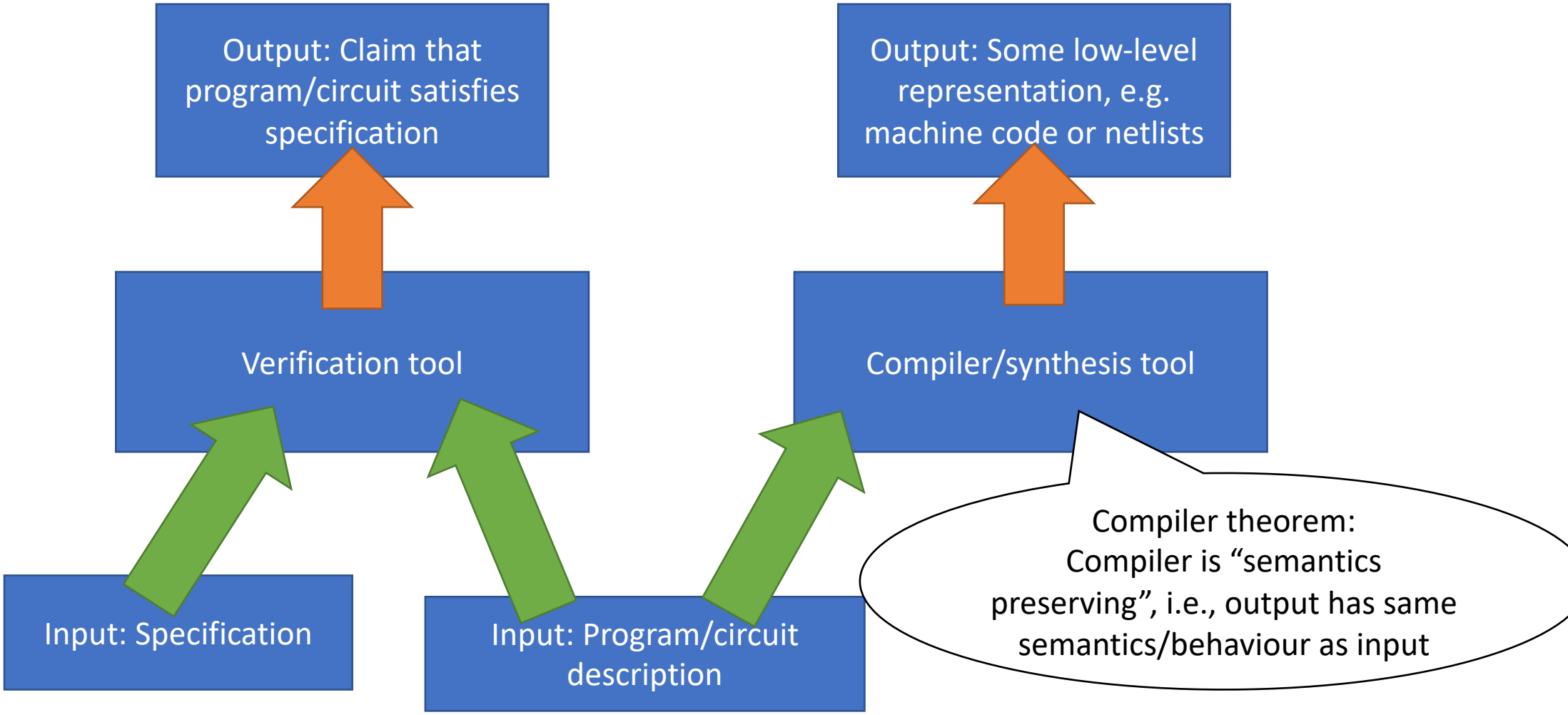
Development flow



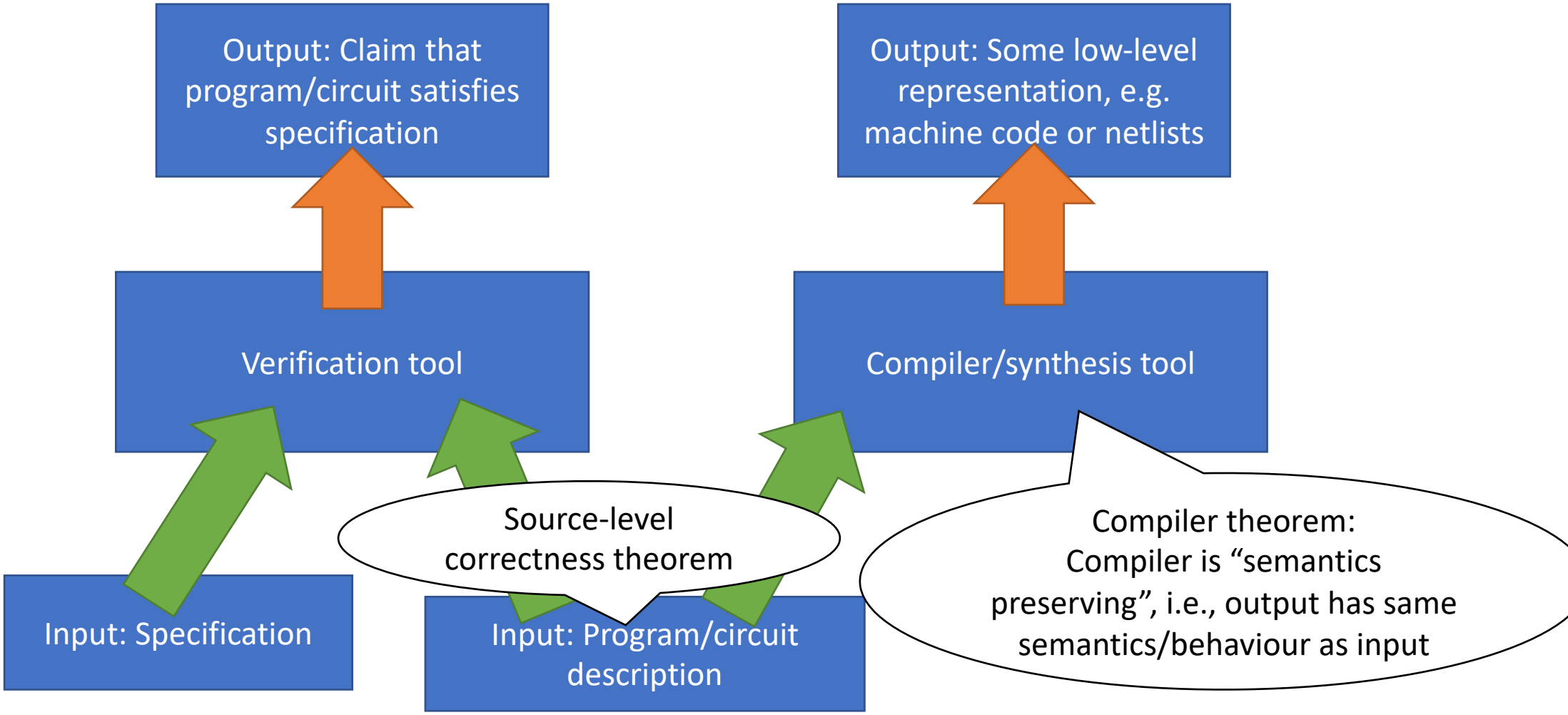
Development flow



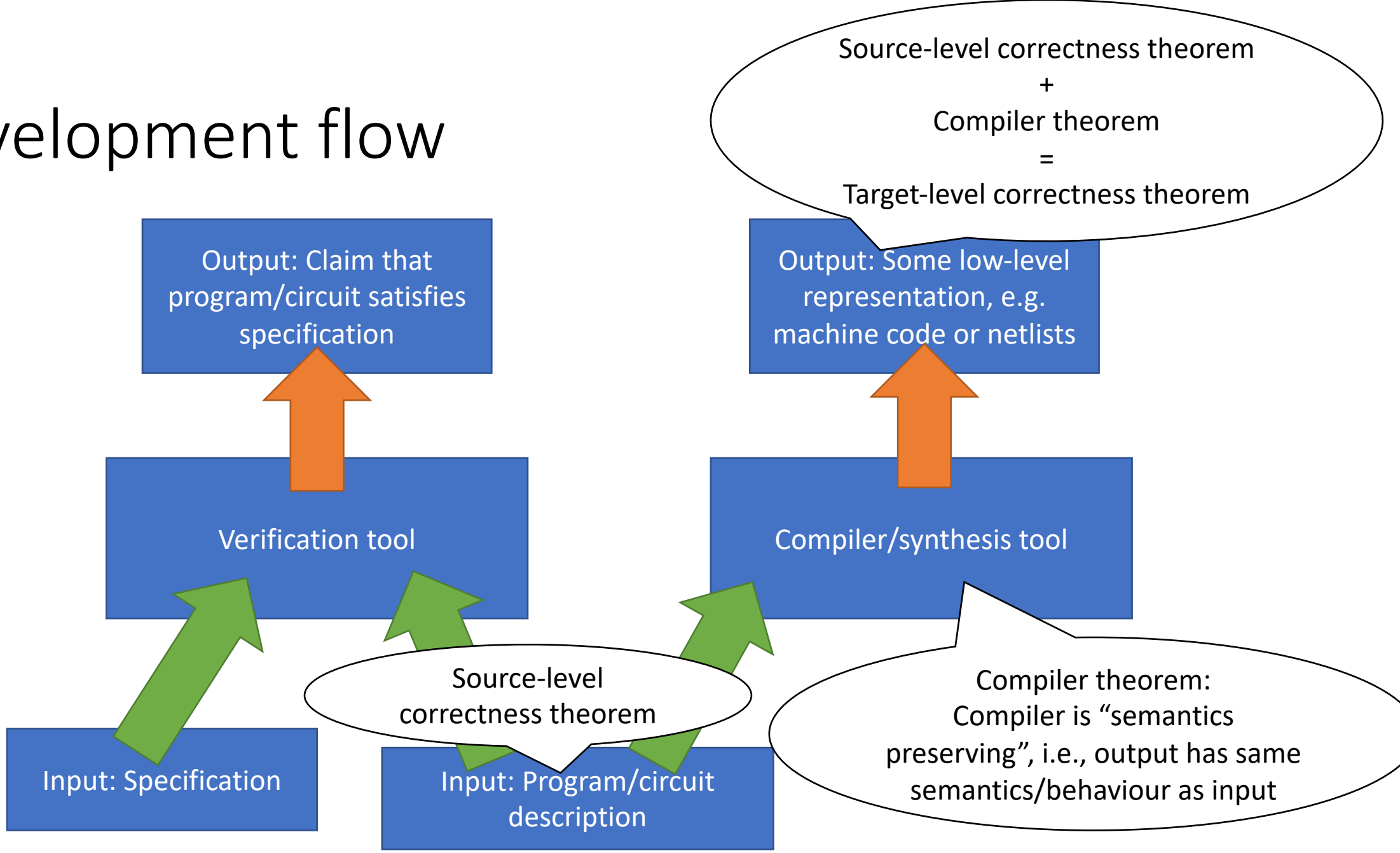
Development flow



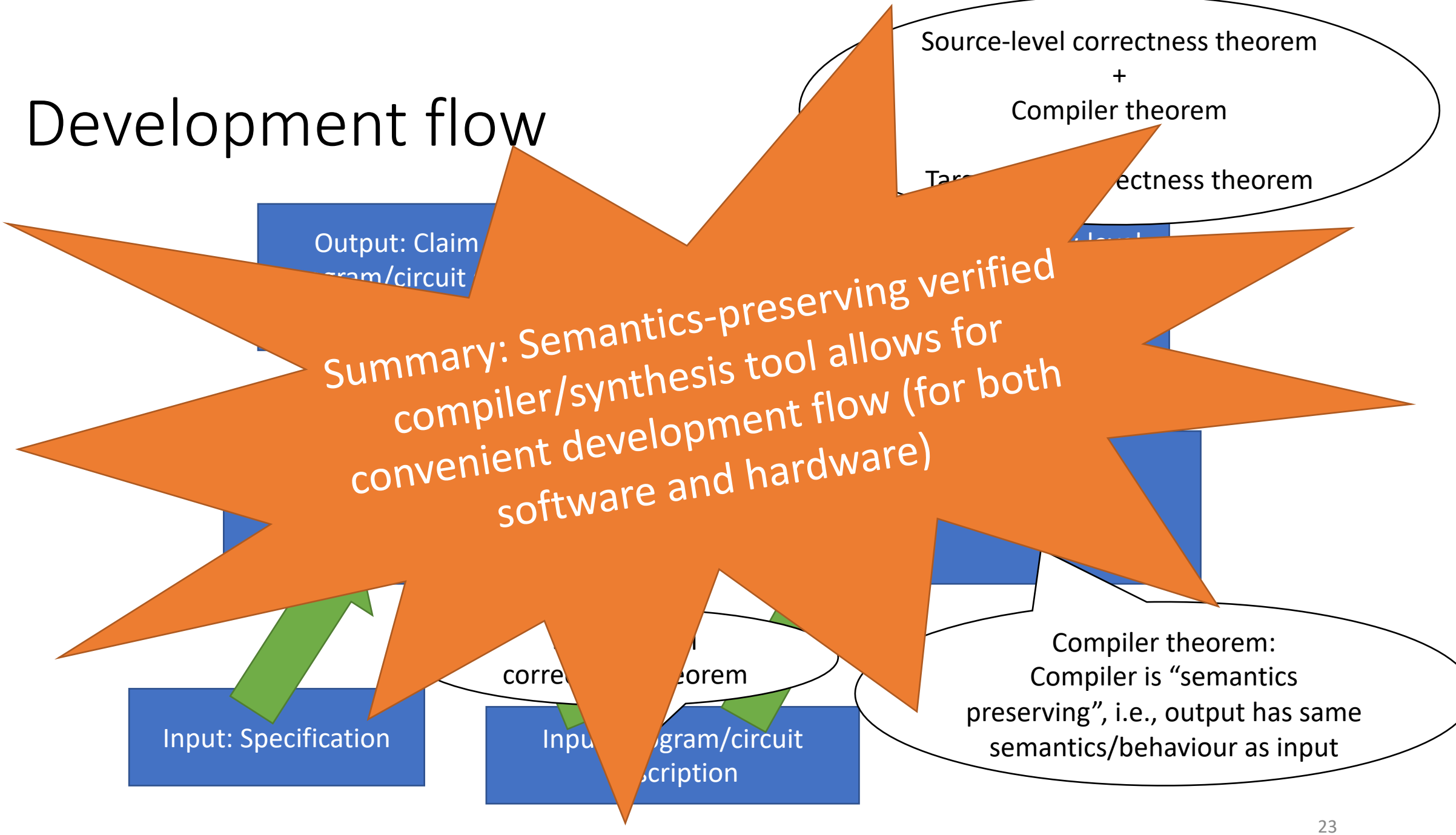
Development flow



Development flow



Development flow



How does verifying a Verilog synthesis tool differ from verifying a compiler for a software language?

Verilog

Gisselquist
Technology, LLC

Main/Blog
About Us
FPGA Hell
Tutorial
Formal training
Quizzes
Projects
Site Index
@zipcpu
Reddit
Support

Reasons why Synthesis might not match Simulation

Aug 4, 2018

When I first learned digital design, I never simulated any of my designs: I just placed them directly onto the hardware and debugged them there.

I've since become convinced in using simulation for several reasons: simulation can be faster than synthesizing a design. Indeed, any time I run Verilator I can find many syntax errors in my design before Vivado fully starts up and shows me one bug. But that's just synthesis. For small designs, simulation is still faster. Of course, ultimately, the hardware is always faster–but in the time it takes to get there, you might manage to get an answer via simulation.

The second reason why I like simulation is that a simulation generated trace will contain *every wire* within the design. For this reason, when something doesn't work in hardware, I'll almost always return to simulation and try to do the same thing in simulation to see if I can come across the same bug. That allows me to be able to turn around quickly and find the bug.

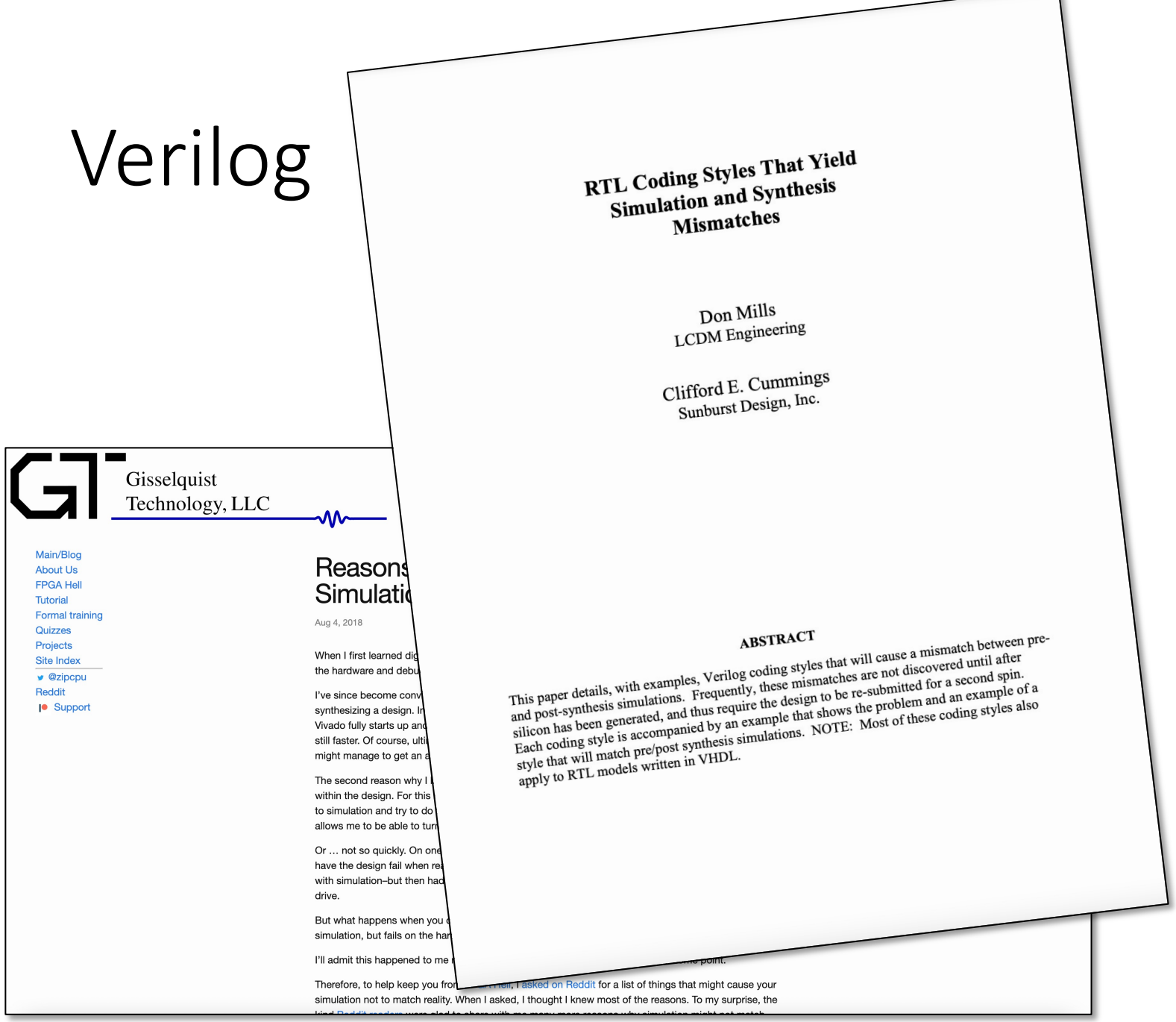
Or … not so quickly. On one recent design, I read the entire 16MB from a SPI flash memory, only to have the design fail when reading the last word from the flash. Not knowing where to start, I started with simulation–but then had to trim down the trace before filling up every bit in my computers disk drive.

But what happens when you cannot simulate the problem? When your design works perfectly in simulation, but fails on the hardware?

I'll admit this happened to me recently as well. I think it happens to everyone at some point.

Therefore, to help keep you from FPGA Hell, I asked on Reddit for a list of things that might cause your simulation not to match reality. When I asked, I thought I knew most of the reasons. To my surprise, the

Verilog



RTL Coding Styles That Yield Simulation and Synthesis Mismatches

Don Mills
LCDM Engineering

Clifford E. Cummings
Sunburst Design, Inc.

ABSTRACT

This paper details, with examples, Verilog coding styles that will cause a mismatch between pre- and post-synthesis simulations. Frequently, these mismatches are not discovered until after silicon has been generated, and thus require the design to be re-submitted for a second spin. Each coding style is accompanied by an example that shows the problem and an example of a style that will match pre/post synthesis simulations. NOTE: Most of these coding styles also apply to RTL models written in VHDL.

Gisselquist Technology, LLC

Main/Blog
About Us
FPGA Hell
Tutorial
Formal training
Quizzes
Projects
Site Index
@zipcpu
Reddit
Support

Reasons
Simulatio

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, ulti
might manage to get an a

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

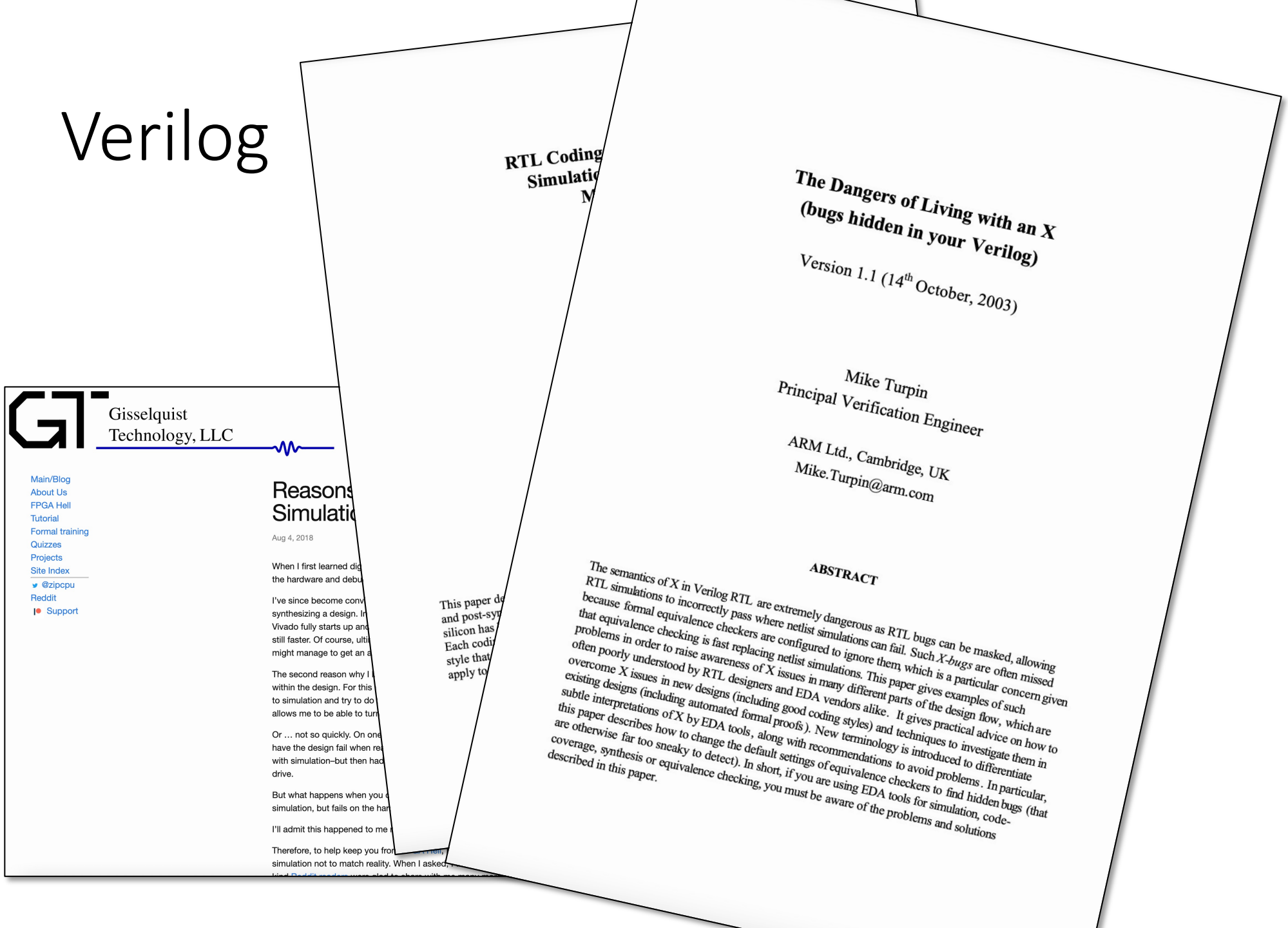
Or … not so quickly. On one
have the design fail when rea
with simulation–but then had
drive.

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

I'll admit this happened to me

Therefore, to help keep you fro
asked on Reddit for a list of things that might cause your simulation not to match reality. When I asked, I thought I knew most of the reasons. To my surprise, the

Verilog



Gisselquist Technology, LLC

Main/Blog
About Us
FPGA Hell
Tutorial
Formal training
Quizzes
Projects
Site Index
@zipcpu
Reddit
Support

Reasons
Simulatio

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, ulti
might manage to get an a

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

Or … not so quickly. On one
have the design fail when rea
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 fro
simulation not to match reality. When I asked,

RTL Coding
Simulatio
M

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

The Dangers of Living with an X (bugs hidden in your Verilog)

Version 1.1 (14th October, 2003)

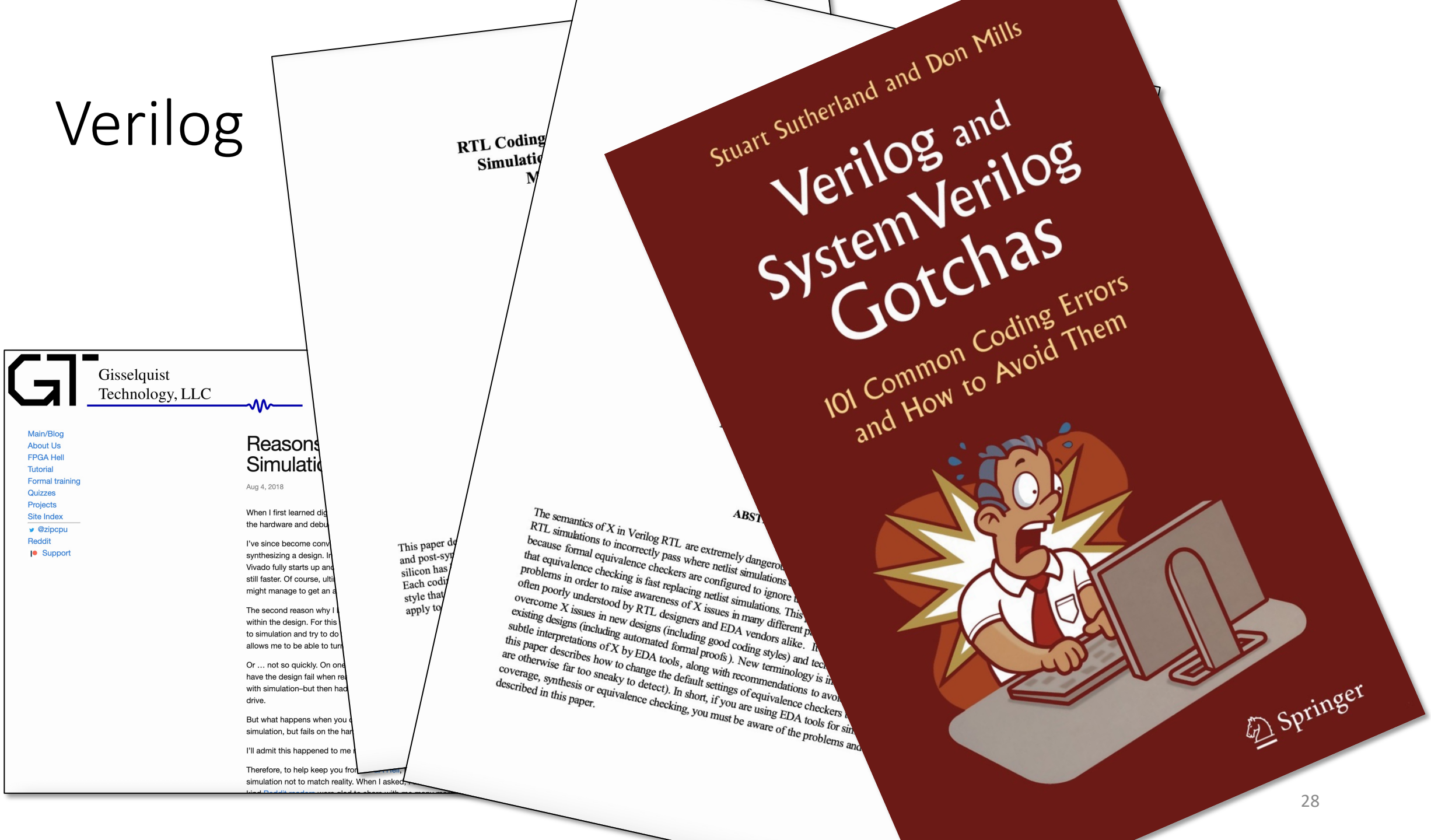
Mike Turpin
Principal Verification Engineer

ARM Ltd., Cambridge, UK
Mike.Turpin@arm.com

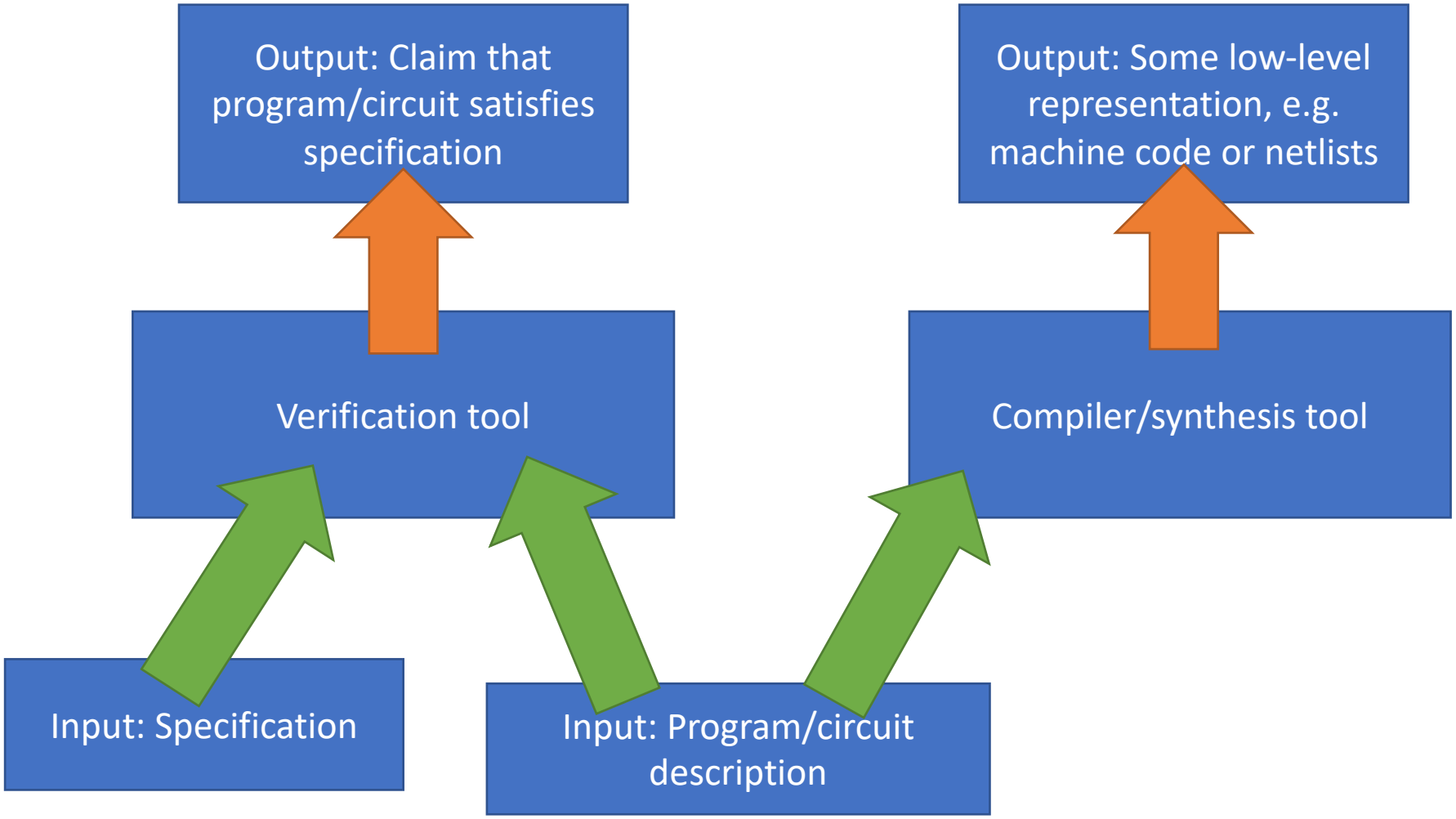
ABSTRACT

The semantics of X in Verilog RTL are extremely dangerous as RTL bugs can be masked, allowing RTL simulations to incorrectly pass where netlist simulations can fail. Such *X-bugs* are often missed because formal equivalence checkers are configured to ignore them, which is a particular concern given that equivalence checking is fast replacing netlist simulations. This paper gives examples of such problems in order to raise awareness of X issues in many different parts of the design flow, which are often poorly understood by RTL designers and EDA vendors alike. It gives practical advice on how to overcome X issues in new designs (including good coding styles) and techniques to investigate them in existing designs (including automated formal proofs). New terminology is introduced to differentiate subtle interpretations of X by EDA tools, along with recommendations to avoid problems. In particular, this paper describes how to change the default settings of equivalence checkers to find hidden bugs (that are otherwise far too sneaky to detect). In short, if you are using EDA tools for simulation, code-coverage, synthesis or equivalence checking, you must be aware of the problems and solutions described in this paper.

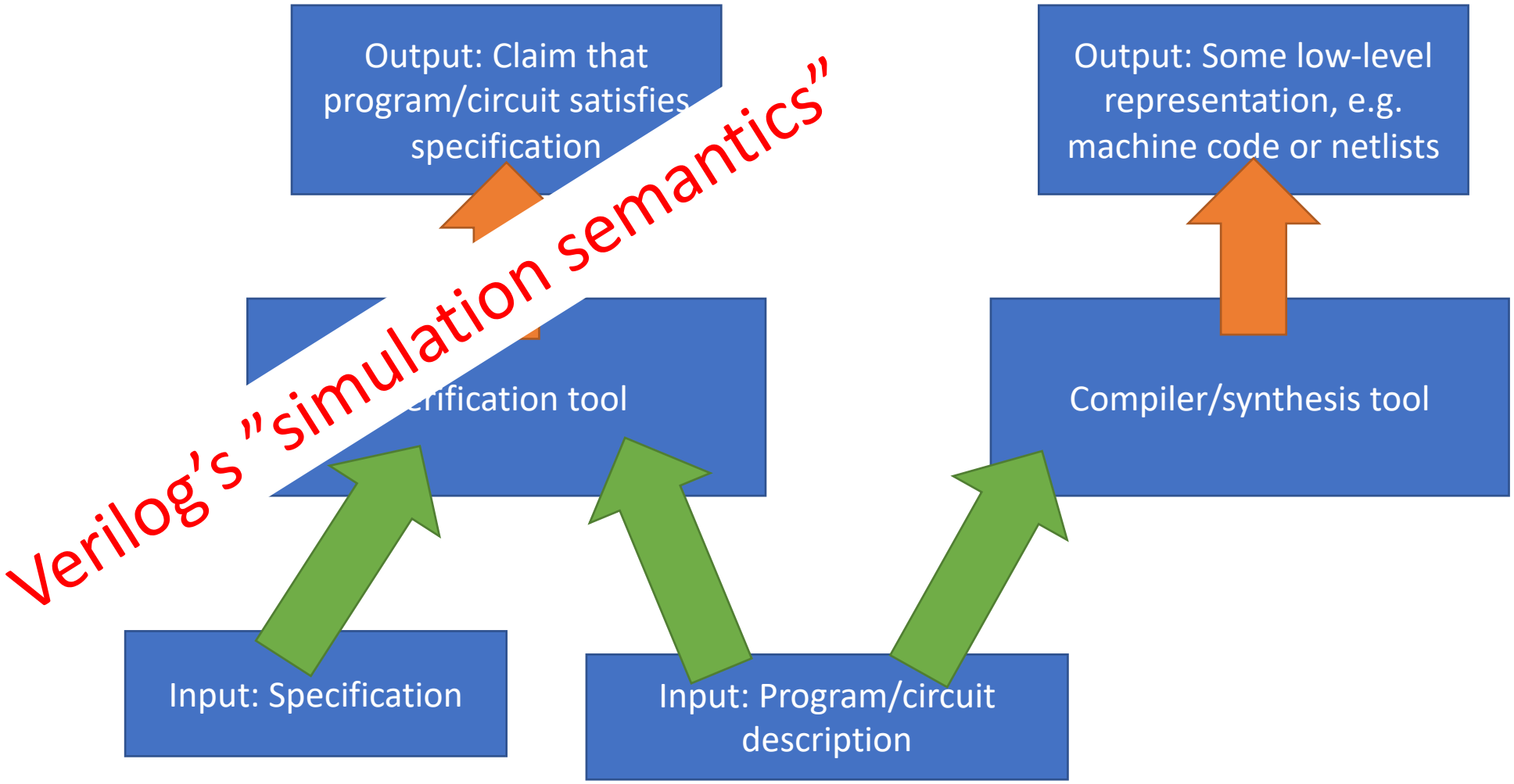
Verilog


Gisselquist
Technology, LLC
Main/Blog
About Us
FPGA Hell
Tutorial
Formal training
Quizzes
Projects
Site Index
@zipcpu
Reddit
Support
Aug 4, 2018
RTL Coding
Stuart Sutherland and Don Mills
Verilog and
SystemVerilog
Gotchas
101 Common Coding Errors
and How to Avoid Them
Springer

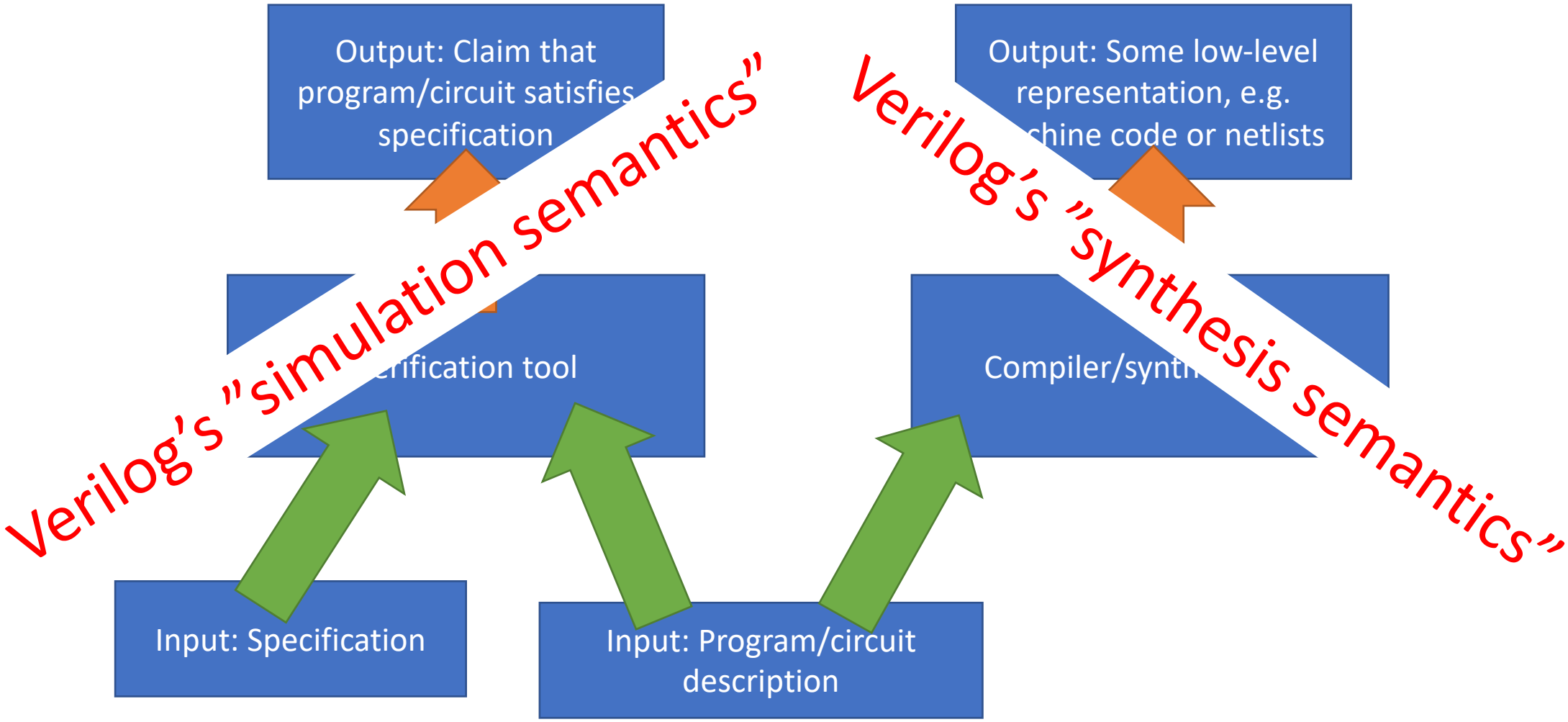
Simulation-and-synthesis mismatches



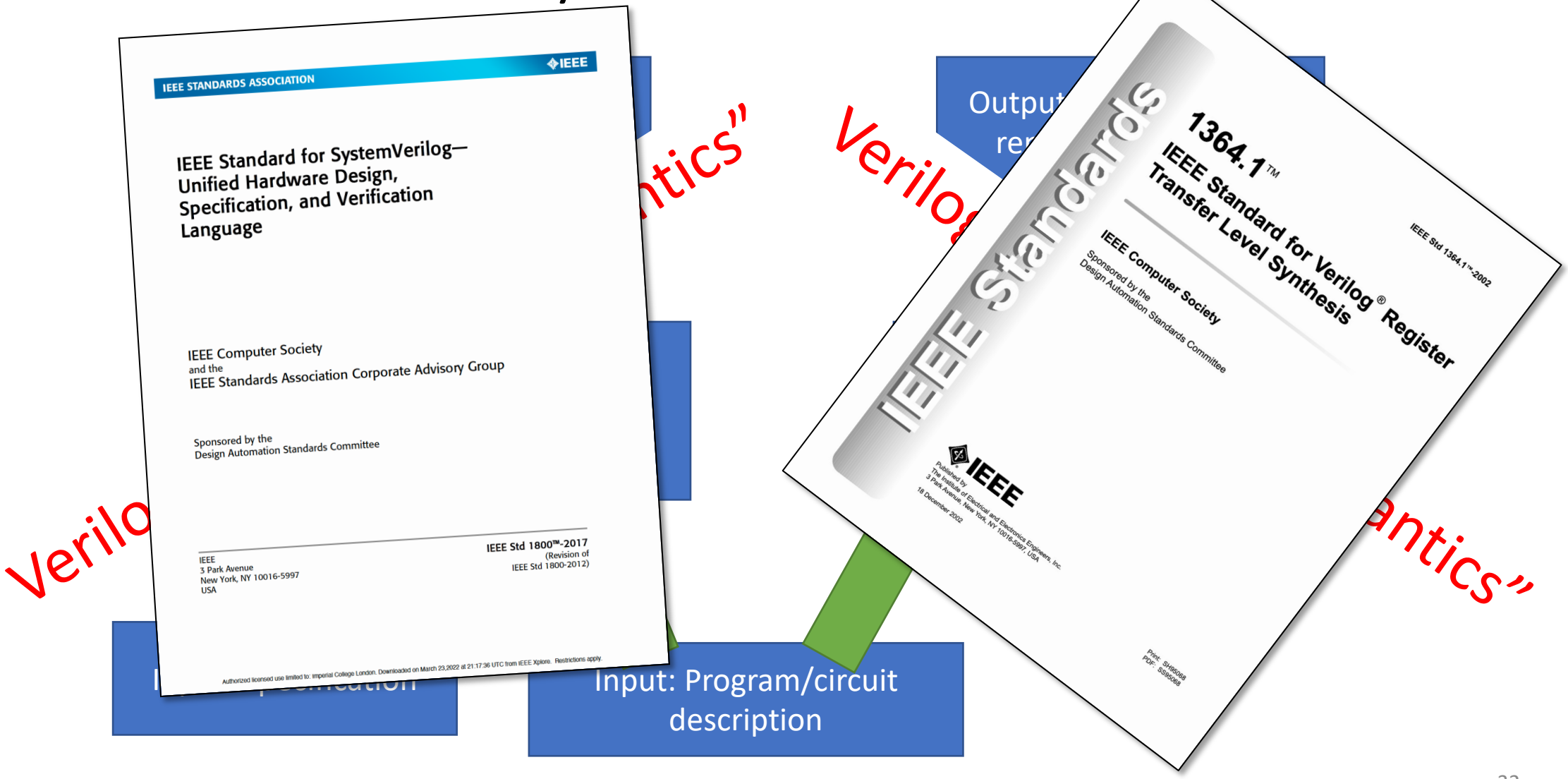
Simulation-and-synthesis mismatches



Simulation-and-synthesis mismatches



Simulation-and-synthesis mismatches



Illustrative example of the clash between the two semantics: Combinational logic

“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” assignment

Example from the “synthesis standard”

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
```

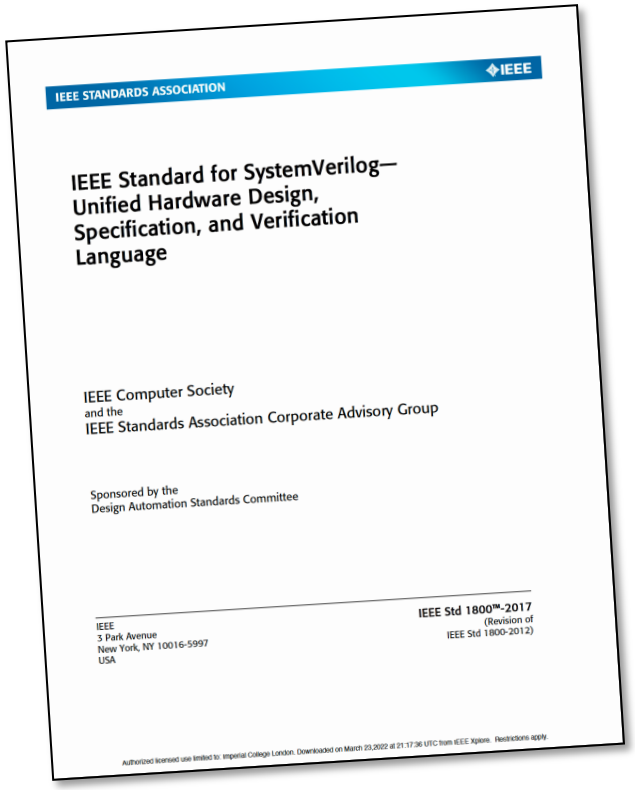
“Mis-ordered” assignments

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


IEEE STANDARDS ASSOCIATION

IEEE

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

IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Sponsored by the
Design Automation Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1800™-2017
(Revision of
IEEE Std 1800-2012)

Authorized licensed use limited to: Imperial College London. Downloaded on March 23,2022 at 21:17:36 UTC from IEEE Xplore. Restrictions apply.

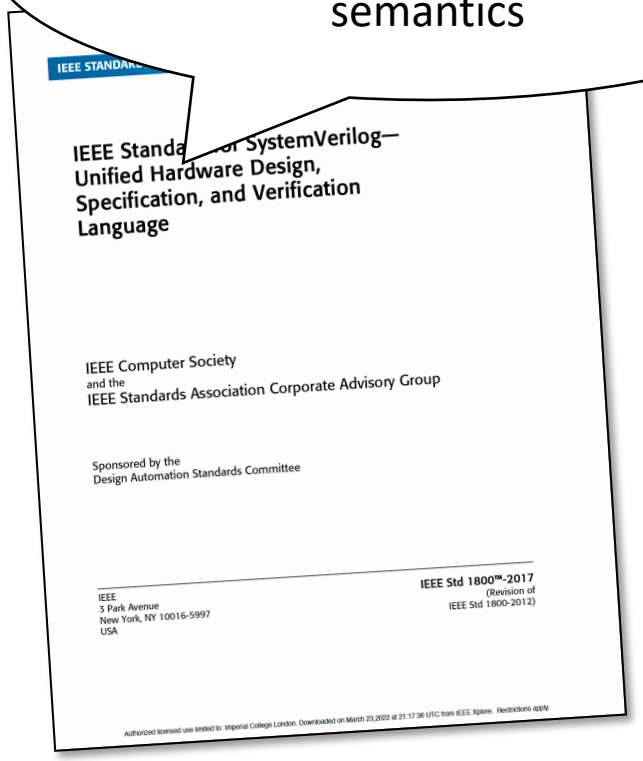
B.5 Assignment statements mis-ordered

```
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

Essentially, a prose-specified event-driven operational semantics


IEEE STANDA

IEEE Standa SystemVerilog—
Unified Hardware Design,
Specification, and Verification
Language

IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Sponsored by the
Design Automation Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1800™-2017
(Revision of
IEEE Std 1800-2012)

Authorized licensed use limited to: Imperial College London. Downloaded on March 23,2022 at 21:17:36 UTC from IEEE Xplore. Restrictions apply.

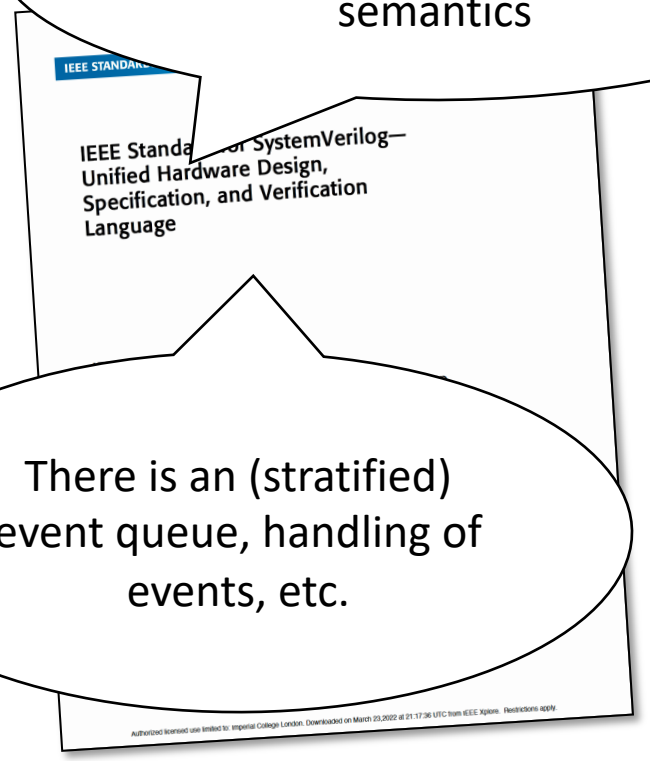
nment statements mis-ordered

```
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

Essentially, a prose-specified event-driven operational semantics



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

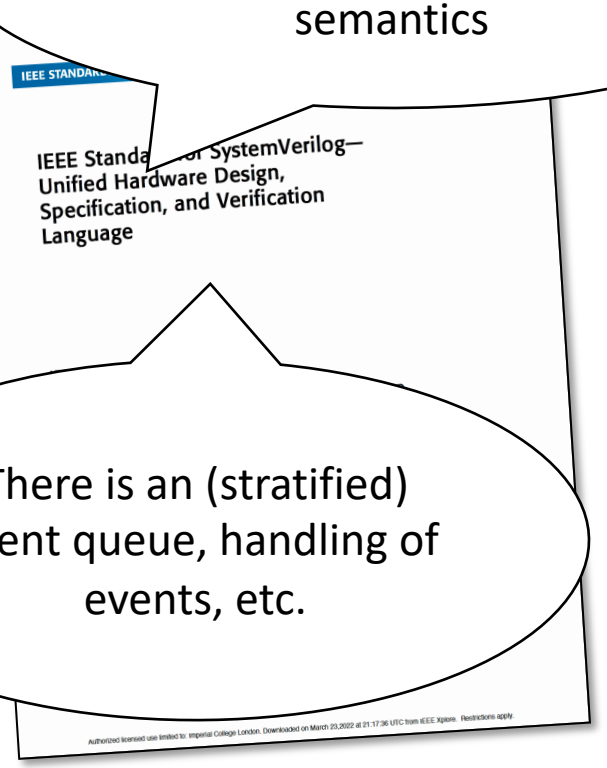
nment statements mis-ordered

```
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

Essentially, a prose-specified event-driven operational semantics



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

nment statements mis-ordered

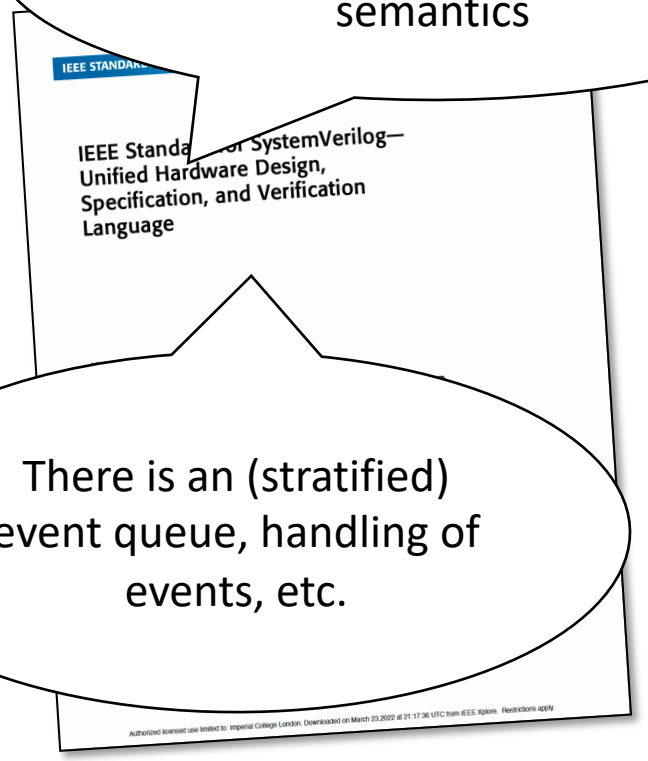
```
module andor1
 output
 input
logic tmp

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

“Mis-ordered” assignments

Essentially, a prose-specified event-driven operational semantics



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

nment statements mis-ordered

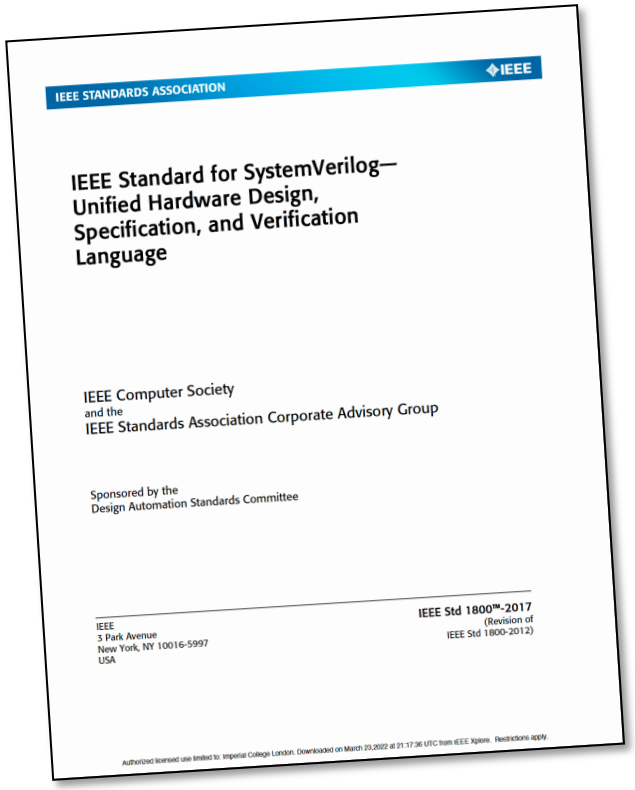
```
module andor1
 output
 input
logic tmp

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

The statements run in the given order

“Mis-ordered” assignments


IEEE STANDARDS ASSOCIATION

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

IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Sponsored by the
Design Automation Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1800™-2017
(Revision of
IEEE Std 1800-2012)

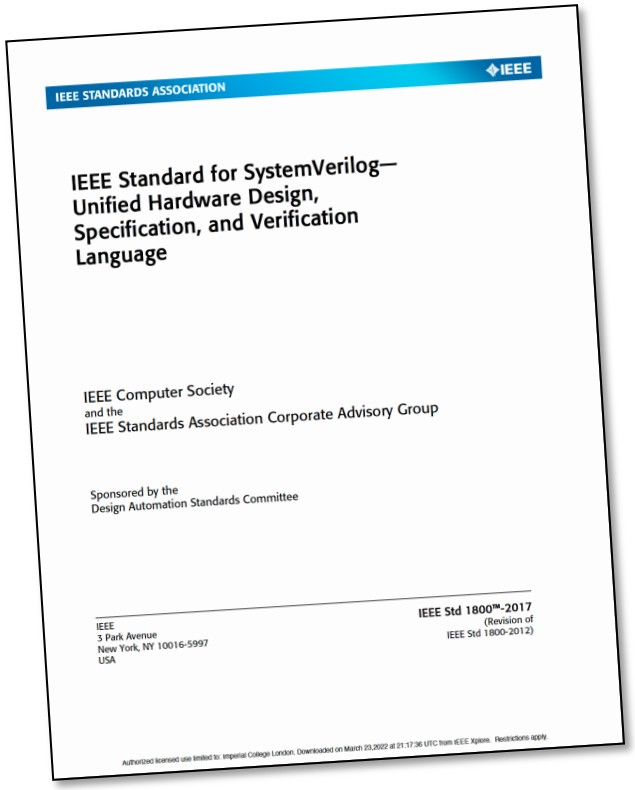
Authorized licensed use limited to: Imperial College London. Downloaded on March 23,2022 at 21:17:36 UTC from IEEE Xplore. Restrictions apply.

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


IEEE STANDARDS ASSOCIATION

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

IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Sponsored by the
Design Automation Standards Committee

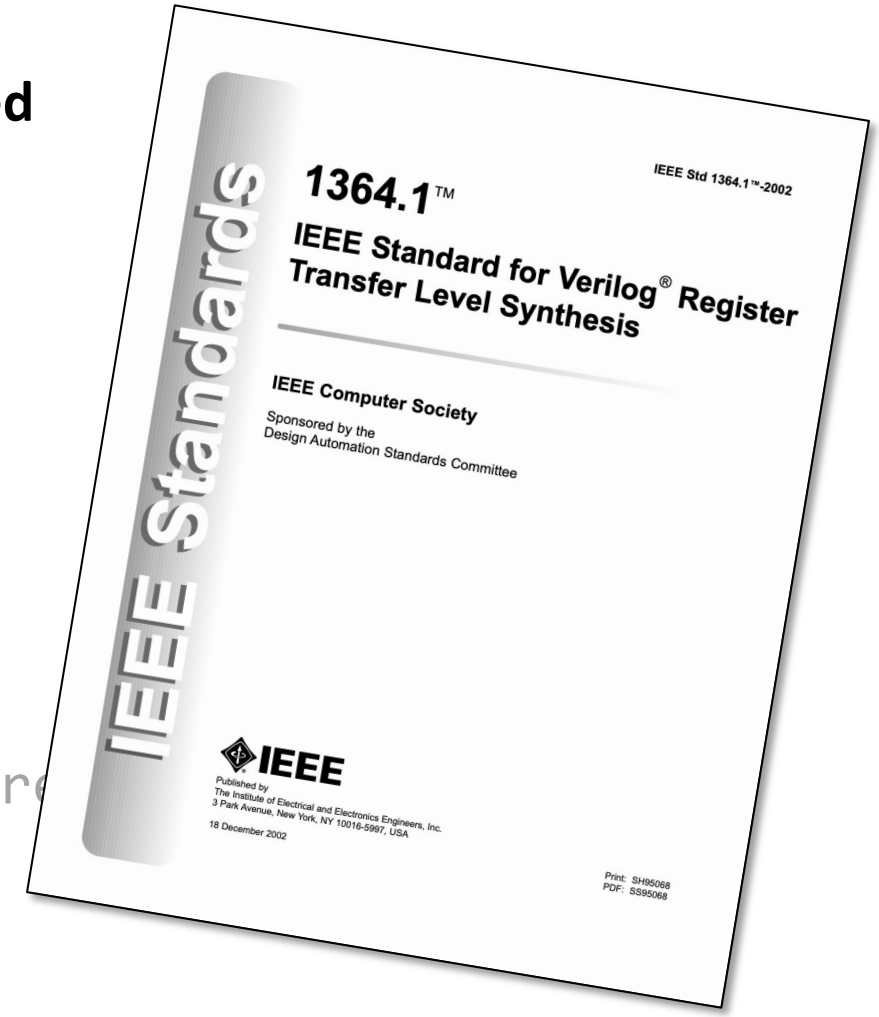
IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1800™-2017
(Revision of
IEEE Std 1800-2012)

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 re
end
endmodule
```


IEEE Standards

1364.1™

IEEE Std 1364.1™-2002

IEEE Standard for Verilog® Register Transfer Level Synthesis

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

IEEE

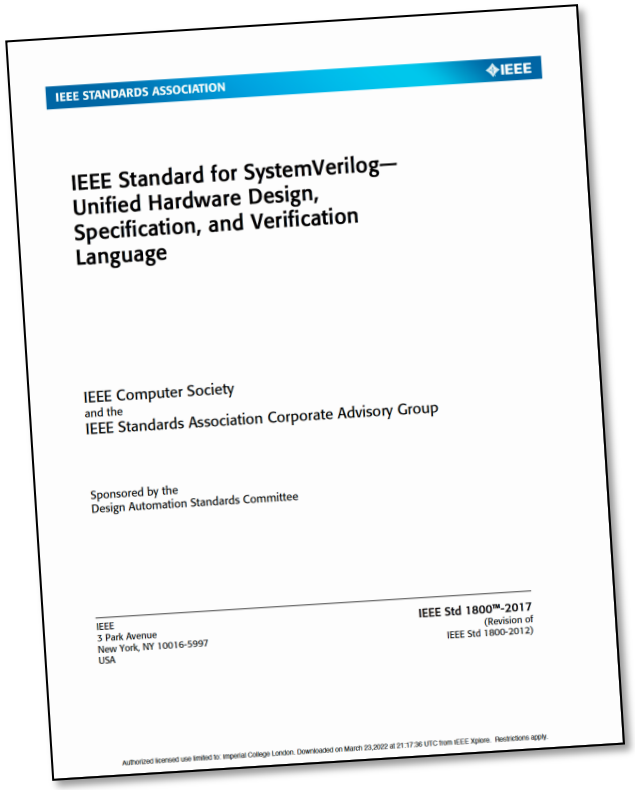
Published by
The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

18 December 2002

Print: SH95068
PDF: SS95068

“Mis-ordered” a

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


IEEE STANDARDS ASSOCIATION

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

IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Sponsored by the
Design Automation Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

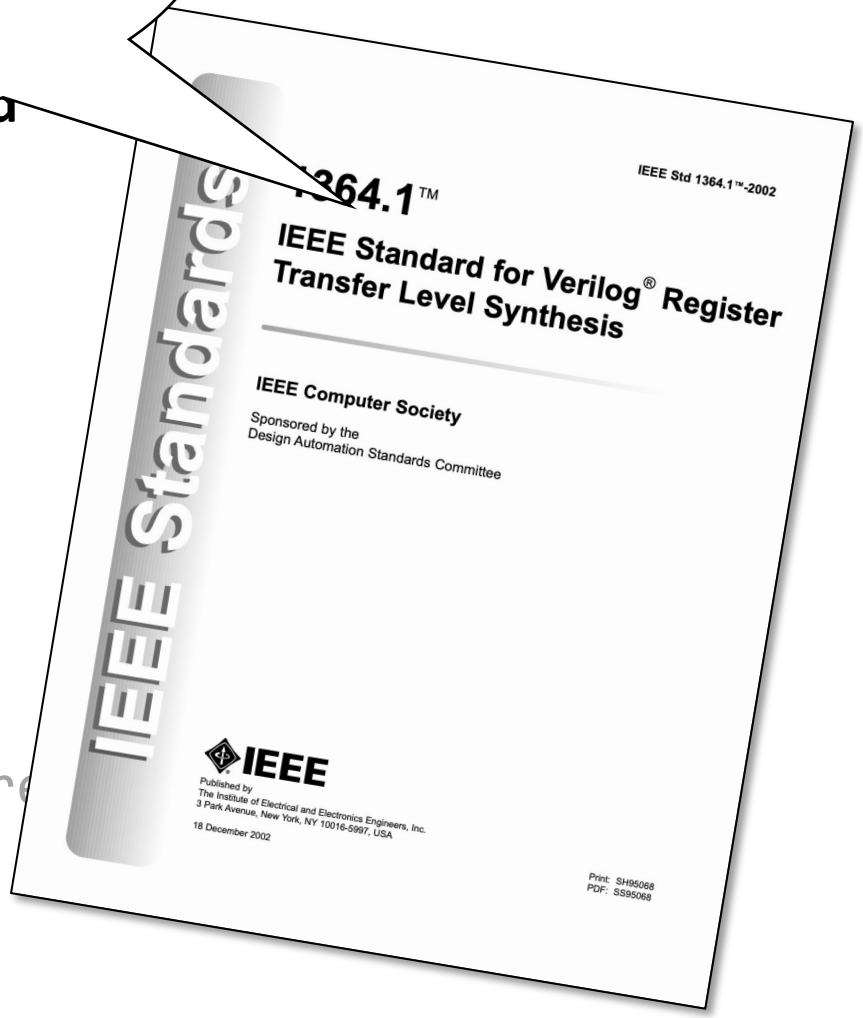
IEEE Std 1800™-2017
(Revision of
IEEE Std 1800-2012)

Authorized licensed use limited to: Imperial College London. Downloaded on March 23,2022 at 21:17:36 UTC from IEEE Xplore. Restrictions apply.

B.5 Assignment stat

```
module andor1a(
 output logic y,
 input logic a, b, c);
logic tmp;

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


IEEE Standards

1364.1™

IEEE Std 1364.1™-2002

IEEE Standard for Verilog® Register Transfer Level Synthesis

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

IEEE

Published by
The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

18 December 2002

Print: SH95068
PDF: SS95068

“Mis-ordered” a

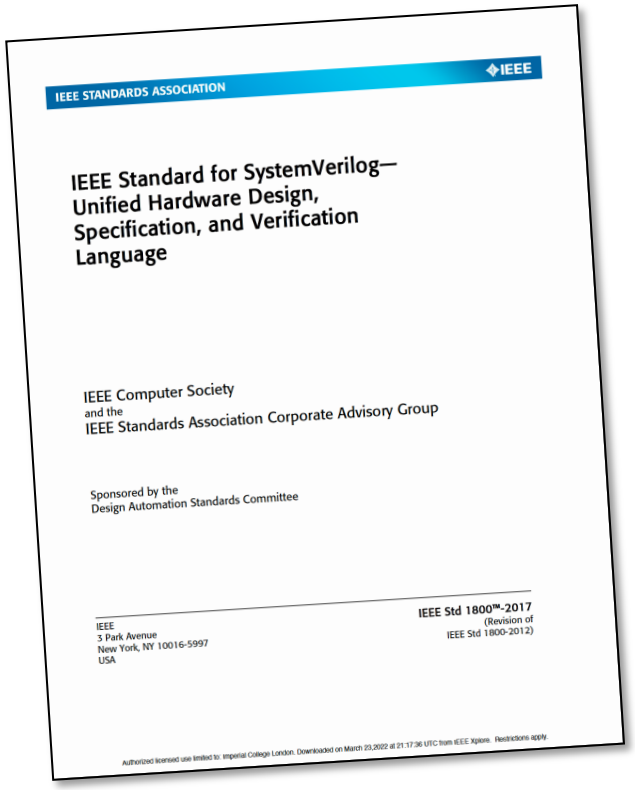
“This standard defines a set of modeling rules for writing Verilog HDL descriptions for synthesis.”

B.5 Assignment stat

```
module a
 output
 input
logic tmp,

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

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


IEEE STANDARDS ASSOCIATION

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

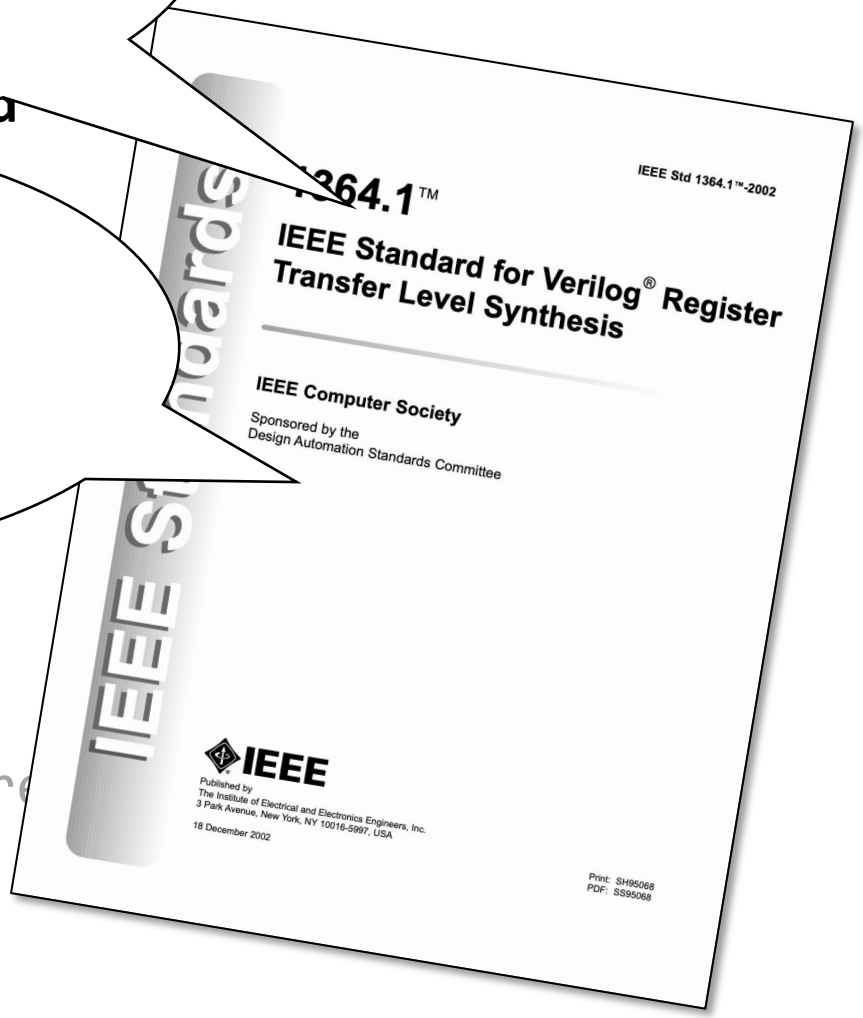
IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Sponsored by the
Design Automation Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1800™-2017
(Revision of
IEEE Std 1800-2012)

Authorized licensed use limited to: Imperial College London. Downloaded on March 23,2022 at 21:17:36 UTC from IEEE Xplore. Restrictions apply.


IEEE Std 1364.1™-2002

IEEE Standard for Verilog® Register Transfer Level Synthesis

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

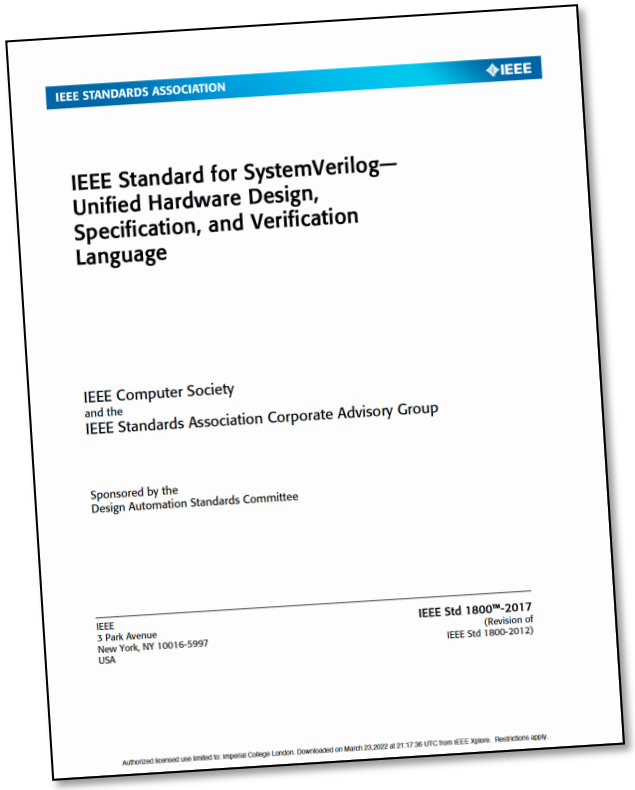
IEEE

Published by
The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

18 December 2002

Print: SH95068
PDF: SS95068

“Mis-ordered” assignments


IEEE STANDARDS ASSOCIATION

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

IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Sponsored by the
Design Automation Standards Committee

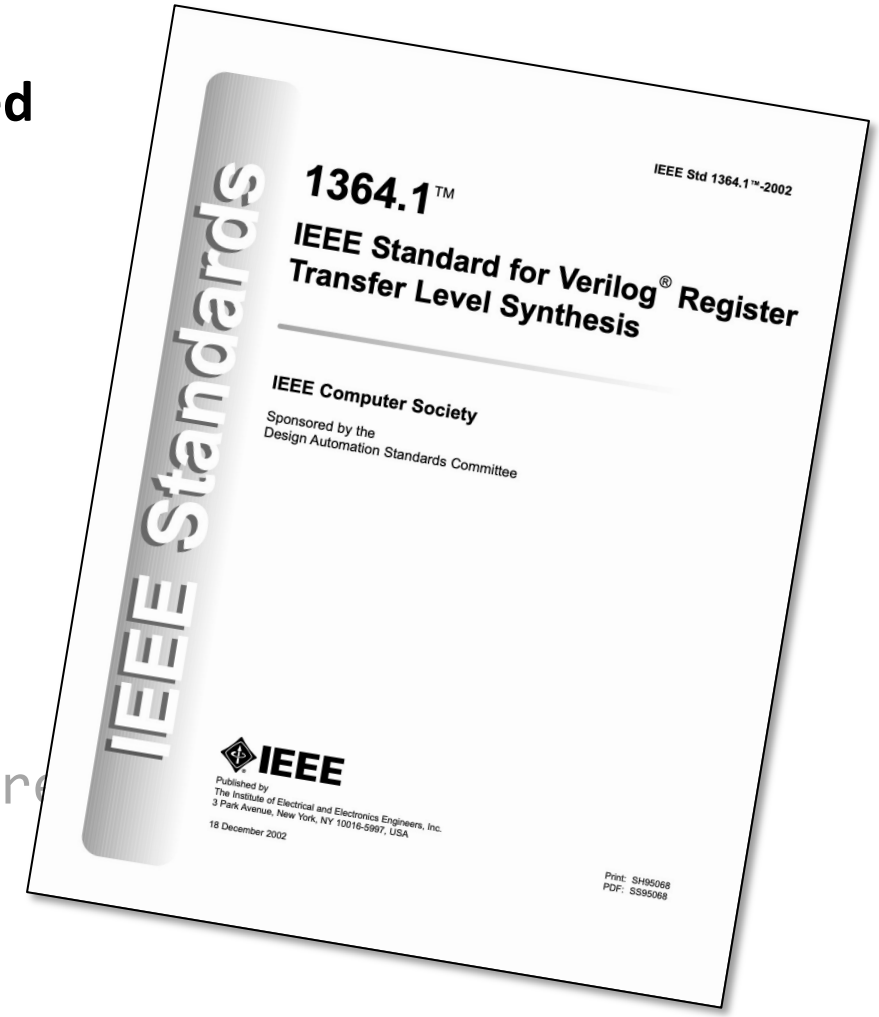
IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 1800™-2017
(Revision of
IEEE Std 1800-2012)

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 re
end
endmodule
```


IEEE Standards

1364.1™

IEEE Std 1364.1™-2002

IEEE Standard for Verilog® Register Transfer Level Synthesis

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

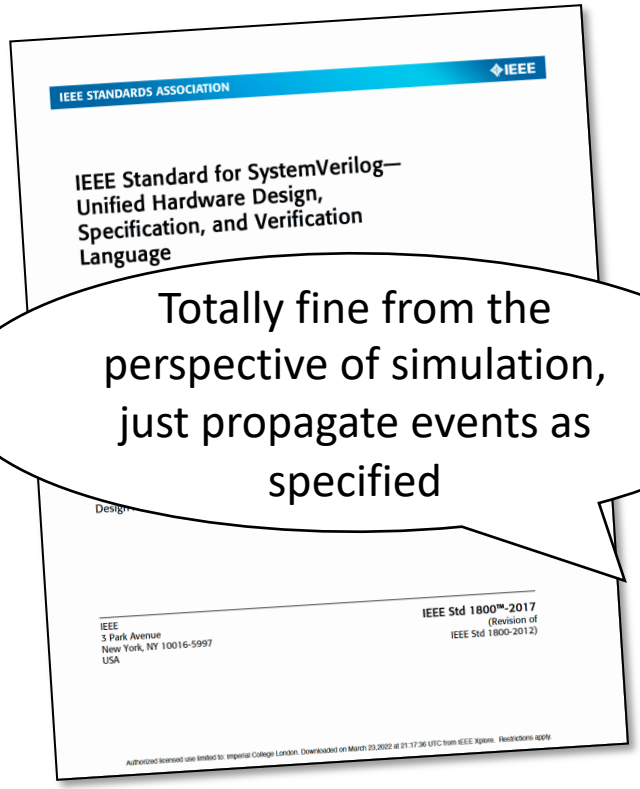
IEEE

Published by
The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

18 December 2002

Print: SH95068
PDF: SS95068

“Mis-ordered” assignments

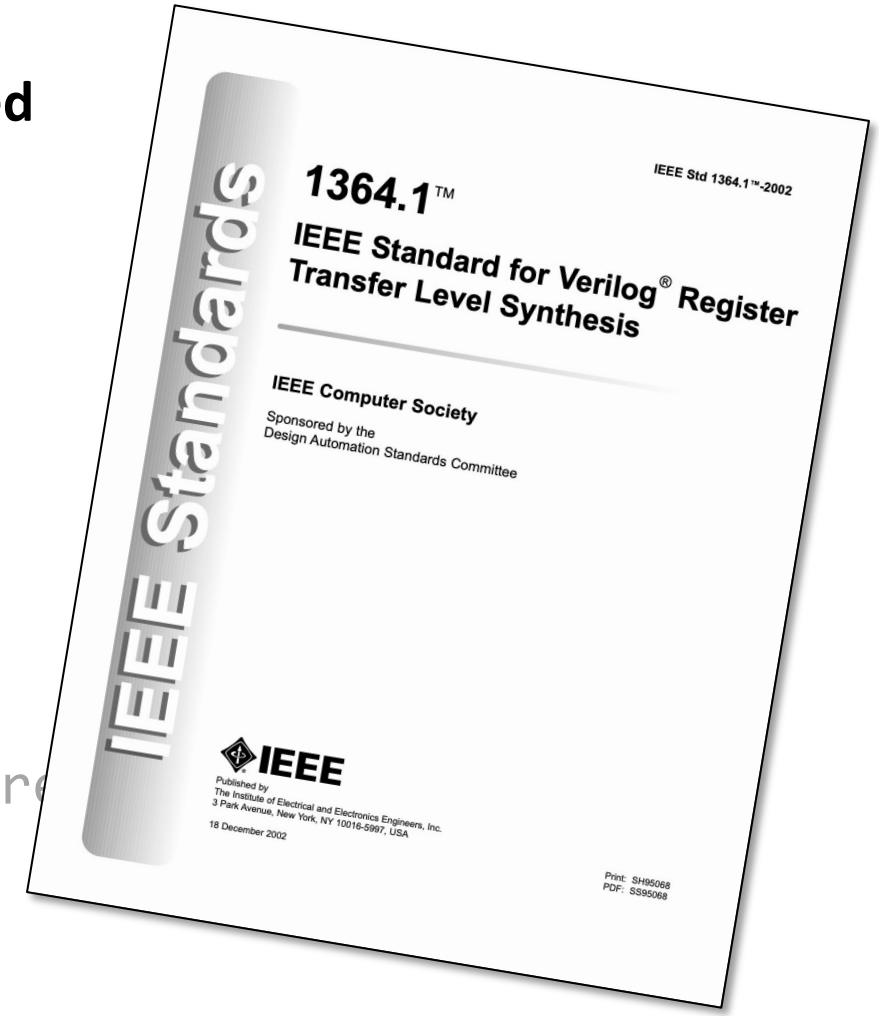


Totally fine from the perspective of simulation, just propagate events as specified

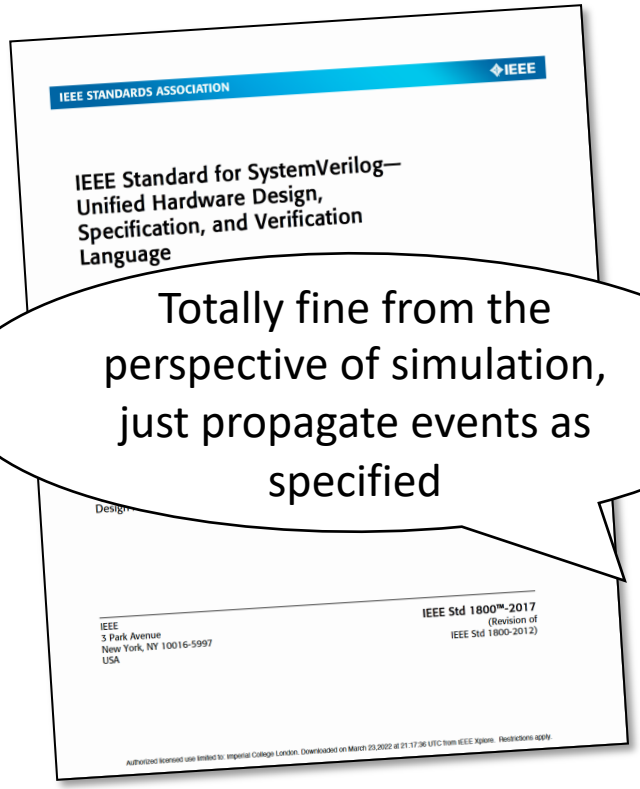
B.5 Assignment statements mis-ordered

```
module andor1a(
 output logic y,
 input logic a, b, c);
   ic tmp;

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



“Mis-ordered” assignments



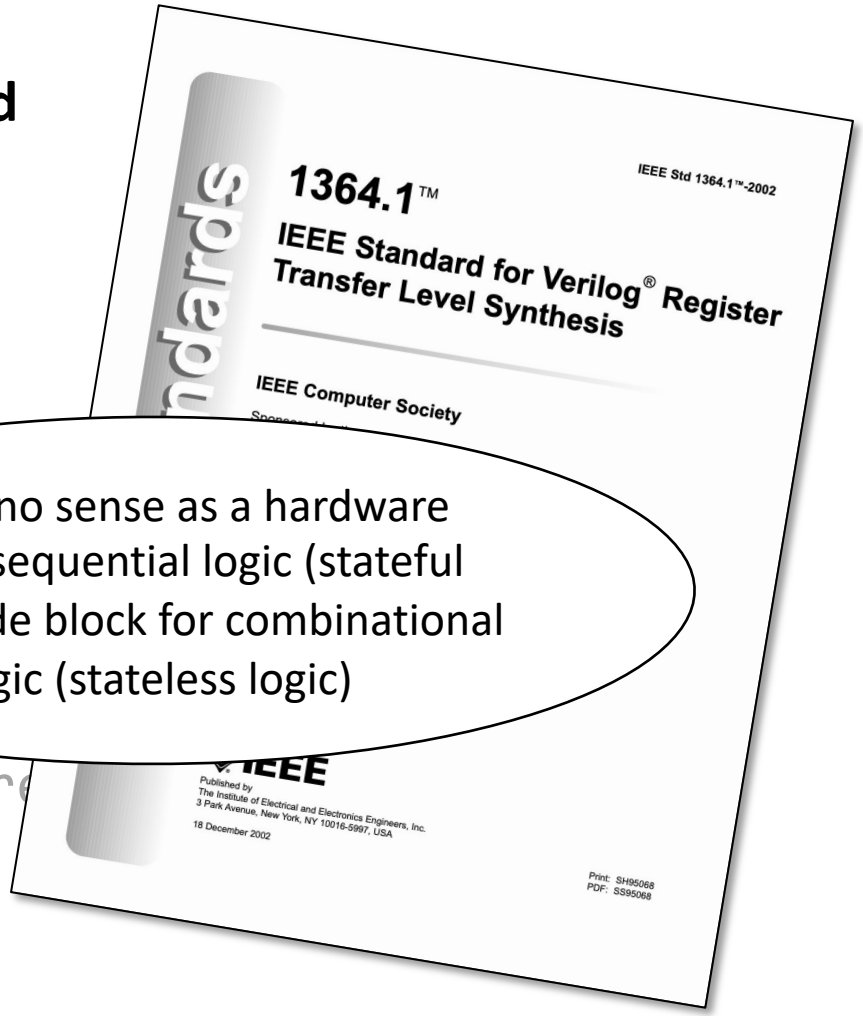
Totally fine from the perspective of simulation, just propagate events as specified

B.5 Assignment statements mis-ordered

```
module andor1a(
 output logic y,
 input logic a, b, c);
   ic tmp;

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

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)

In other words, such synthesis tools are **not** semantics preserving

Lutsig's solution

- Need both semantics
  - Simulation semantics for circuit-correctness theorem transportation
  - Synthesis semantics for actually describing hardware, not just behaviour

- Informally: Lutsig is forced, as we will see, to abort if there's a mismatch between the two

- Formally: There are two theorems…

Lutsig's correctness theorems (simplified)

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

Lutsig(D) = OK(N) ==> forall n, run_verilog(D, n) = run_netlist(N, n)

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

Correctness w.r.t. modelling rules for always_comb:

Lutsig(D) = OK(N) ==>
forall Verilog variables v in D,
if v written to by always_comb block ==>
no register with name v in netlist N

Lutsig in practice

- 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 (i.e., you broke some of the "modelling rules"), 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

- Combinational blocks (always_comb):
  - 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

Some other sources of mismatches to think about

- First version of Lutsig: X values – too broken to use standard semantics

- First version of Lutsig: Correct blocking and nonblocking assignments usage

- Other modelling rules, e.g., block RAM inference should be similar to how combinational logic is handled in Lutsig

Conclusion

- Verilog is a… tricky language…
- (Although, in Verilog's defence, difficult to avoid this when modelling hardware behaviourally.)
- Nevertheless, this new version of Lutsig is one attempt at doing formal hardware development using Verilog