Open Source Formal Verification

Introduction to hardware verification

Yann Thoma

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

July 2025

Verification goal

Answer two questions

- Does it work?
- Are you sure?
- Really?
- Well, really really sure?
- No kidding?

Verification : yes, but ...

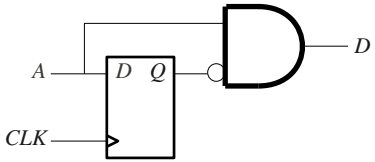
We verify ...

- What?
  - FPGA design
  - Software
  - Both
- Why?
- When?
- How?

First question: What?

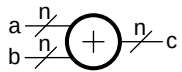
- Verification of the correct behavior of the system
- What does that mean?
- Verificatino of each and every module
  - Is exhaustive verification affordable?

- An edge detector:



  - Exhaustive verification: OK

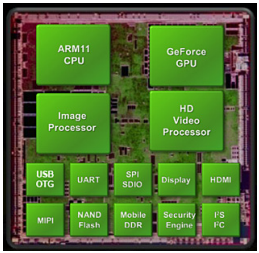
- A 64-bits adder:



  - Exhaustive verification: No
  - $2^{128} = 3.4 \times 10^{38}$ combinations...

First question: What?

- Verification of the correct behavior of the system
- What does that mean?
- Verification of the whole system (System on Chip for instance)
  - Exhaustive verification unfeasible
  - $\Rightarrow$ Choose interesting cases
  - $\Rightarrow$ Random verification



http://www.xbitlabs.com/news/mobile/display/20080603141353_Nvidia_Unleashes_Tegra_System_on_Chip_for_Handheld_Devices.html

Question: Why?

- Patching an FPGA remotely is not always ideal
- Also, it can harm the system (you know that)
- Verification required to validate the system
  - Needs efficient tools
  - A rigorous methodology
- Verification is *the challenge* of digital design

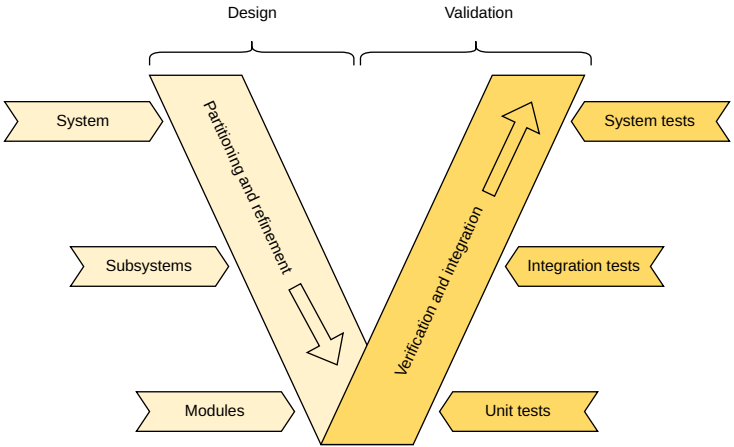
Verification: concept

- Goals
  1. Assess the conformity of the system relative to the specifications
  2. Detect errors as soon as possible
  3. Ensure a correct behavior after synthesis and integration
- Difficulties
  1. To keep the verification cost low
     - Around 50-60% of time spent on verification
  2. To guarantee a correct behavior

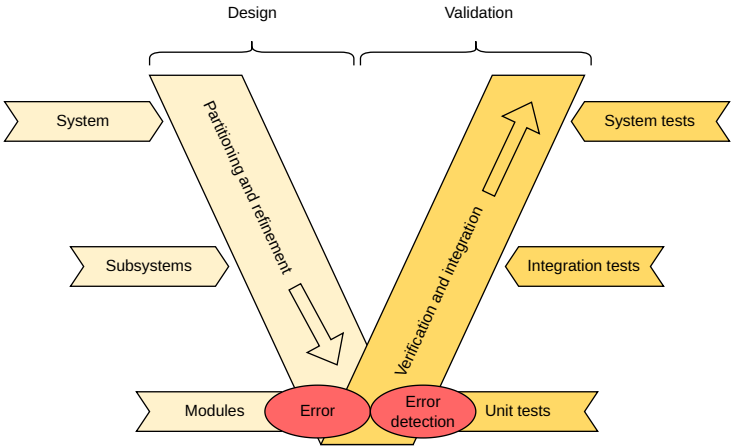
Question: When?

- Obviously as soon as possible
- Not after hours spent trying to spot why the system does not work
- Unit tests on all modules during design and implementation
- System tests or integration tests as soon as possible
  - Harder to setup
  - Huge simulation times
  - When? At night...

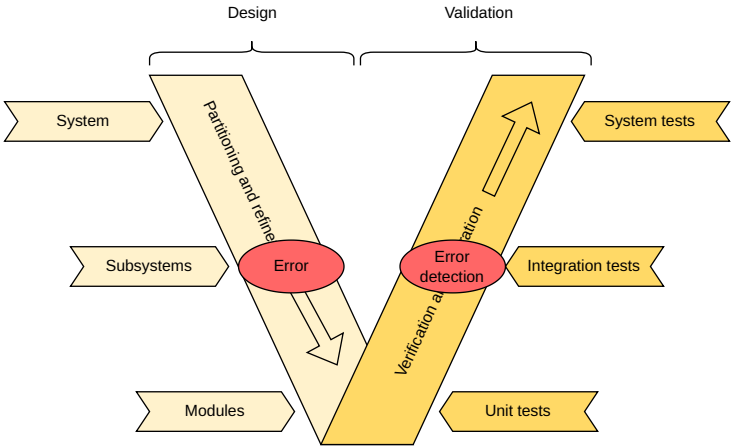
Design/Verification: decomposition



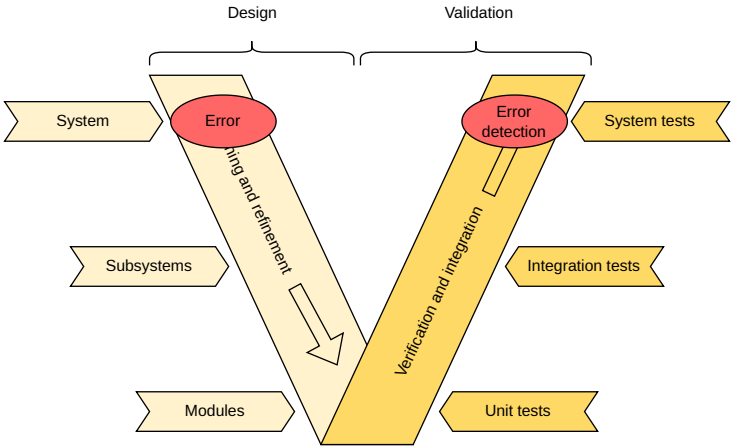
Design/Verification: decomposition



Design/Verification: decomposition



Design/Verification: decomposition



Verification: the 2 questions

- Two big questions during verification
  1. Does it work?
     - Does the system behave as expected?
     - Are errors correctly detected?

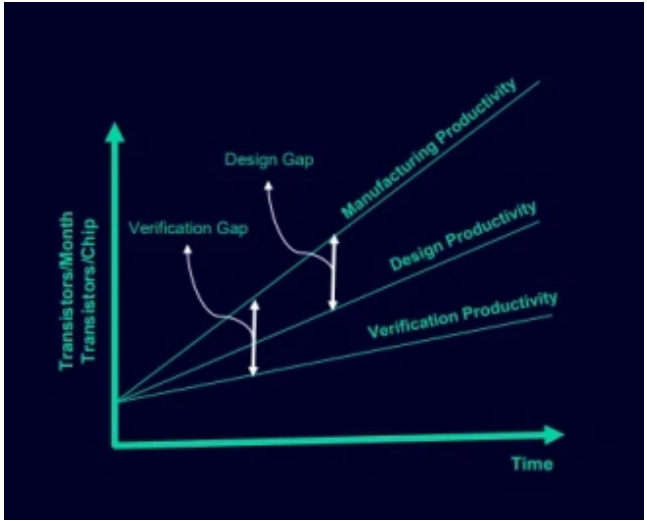
Example of a testbench that passes all tests

```
entity my_great_design_tb is
end my_great_design_tb;

architecture my_great_architecture of my_great_design_tb is
begin
    process is
    begin
        report "Everything is fine";
    end process;
end my_great_architecture;
```

  2. Are we sure? (Are we over?)
     - Did we run enough tests?
     - Easy with a small one, quite difficult with a complex design

Verification: Evolution of Complexity



https://pradeepstechpoints.wordpress.com/2022/06/13/three-pillars-of-intent-focused-insight-verification-futures-2022/

Question: How?

- Language choice
  - VHDL
  - SystemVerilog
  - SystemC
  - Python
  - ...
- Methodology choice
  - UVM
  - OSVMM
  - UVVM
  - ...
- Technological choice
  - Formal verification
  - Simulation-based verification
  - Emulation
- Interactive
  - Manual or automatic

Challenges of verification: methodology choice

- The methodology choice should take into account:
  - Completeness
    - Maximizing the number functionalities (scenarios) tested
  - Reusability
    - Maximizing code reusability for future projects
  - Efficiency
    - Minimizing the effort, and maximizing automation
  - Productivity
    - Maximizing the usage of manual work
  - Code performance
    - Minimizing computing time

Functional verification: approaches

- Simulation-based verification
  - Attempt to prove the system correct behavior by simulating it
- Emulation-based verification
  - Attempt to prove the system correct behavior by emulating it in an FPGA
    - Similar to simulation-based verification
- Formal verification
  - Attempt to *mathematically* prove the system correct behavior

Simulation-based verification: approaches

- Different approaches to simulation-based verification:
  - Directed tests
    - Manually-designed tests
    - For instance: exhaustive tests (!)
  - Coverage-driven random-based
    - Generation of random stimuli
    - Coverage-driven for determining the end of the simulation
  - Assertion-based verification
    - Assertions can be used for formal verification
    - But also used in simulation
- Tools (languages) are more or less suited for one or the other approach

Objectives of simulation

- Verify the deterministic behavior of the module
- Verify the dynamic characteristics of the module (execution time, max frequency, ...)
- In both cases, a written proof should be generated by the tools
  - Result interpretable by CI tools

Verification methodology

- Exhaustive verification
  - NP-hard
  - In general: Impossible to test all possibilities for a combinational system
  - For a sequential system: worse
- *Standard* verification
  - Test of the code (white box)
    - Code coverage
  - Test of the behavior (black box)
    - Verification of the system behavior, without any information on its internal architecture
    - Ideally should be done by another person than the design developer
- Advanced approaches
  - Assertions, random-based coverage-driven, ...

Verification methodology

- We should avoid testbenches that do not detect design flaws
  - Very easy to write an happy testbench who doesn't detect any error...
  - Rigor is essential in writing a testbench
  - Testbench auto-test is a good idea
    - The testbench modifies the design outputs to check itself

Simulation output validation

- For direct verification, validation is given by:
  - List of the generated stimuli et verifications done (testbench files)
  - Written proof of the verification results (simulator log, text file, HTML, junit XML, ...)
- For random-based verification, validation is given by:
  - List of the tested scenarios et verifications done (testbench files)
  - Coverage
  - Written proof of the verification results (simulator log, text file, HTML, junit XML, ...)

Formal verification

- Checking the system against properties
- Proof of the adequation between the system and its specifications
- Advantages
  - Formal proof, so irrefutable if properties are well written
  - Allows to validate part of a system that can then be used in simulation without further tests
- Disadvantages
  - Maybe not adequate for all systems
  - Better with control than data
  - A specific know-how
- Could need to be complemented by functional verification

Manual vs. automatic

- Manual verification
  - Forcing input signals *by hand*
  - Huge risk of errors
  - Hard to to execute the same scenario twice
  - Visual verification by the developer, thanks to simulation chronograms
  - Slow and error-prone
  - Risk of bad analysis by the developer
- semi-automatic verification
  - Input signals applied automatically
  - Visual verification by the developer, thanks to simulation chronograms
  - Risk of bad analysis by the developer

Automatic verification

- Goal: verification as complete as possible
- Results in a Go/No Go (OK, KO)
- Indicates the time of error
- Enables a complete rerun (Go/No Go) after a modification or correction
- Perfect for continuous integration
- Interactions with humans should be minimized
  - Required to ensure that reruns are similar

Automatic simulation: scripts

- Obviously, scripts are used to start simulations
- Avoid human errors
- Can be shared by members of the team
- Make life much simpler

Example: sim.do for QuestaSim

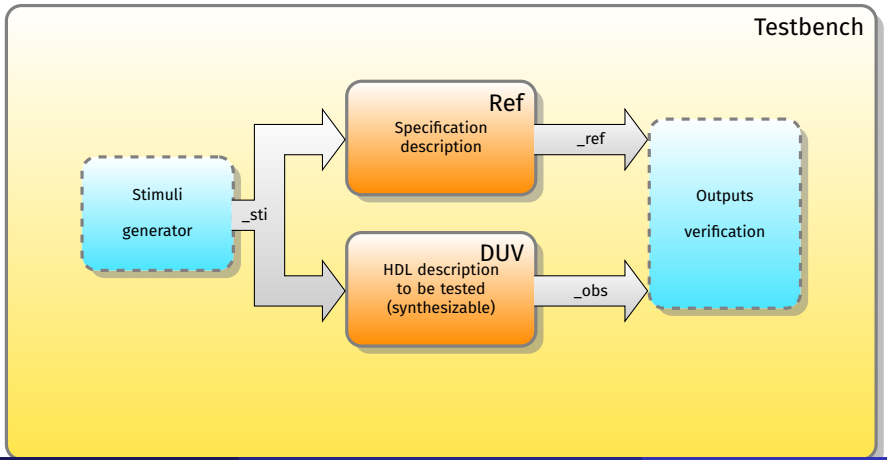
```
# Definition of the work library
vlib work
# compilation of the DUV
vcom counter.vhd
# compilation of the testbench
vcom counter_tb.vhd
# launching of the simulation
vsim work.counter_tb
# Addition of all signals to the chronogram
add wave -r *
# Simulation run
run -all
```

Black/white/gray box verification

- Goal of a design: the outputs respect the specifications
- But... Some internal errors could be untrackable
  - Never-activated errors
  - Errors that do not propagate to the outputs
  - Errors canceling each other
- Black box
  - Only the design outputs are available for verification
  - Typically compared to a reference model

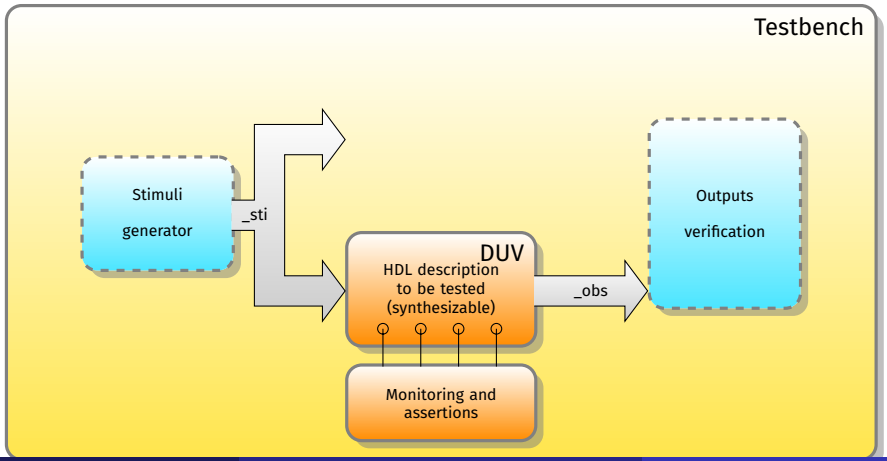
Black box

- Only testing the outputs
  - Hard to find the source of errors
  - Requires a reference model



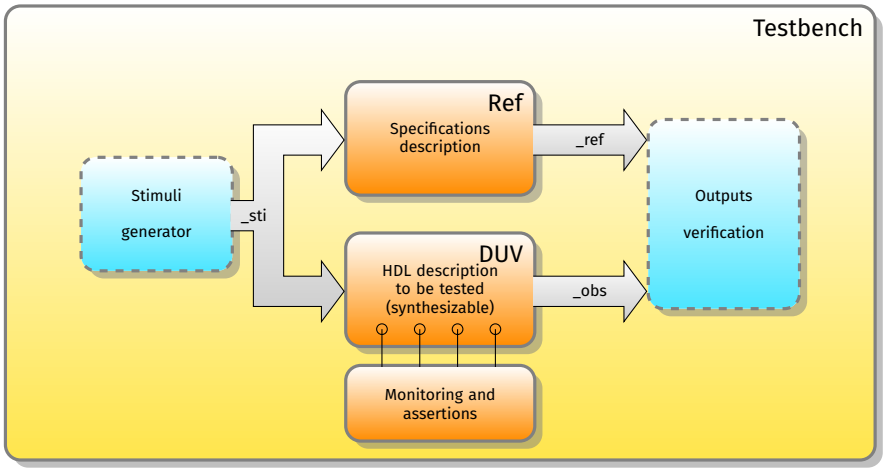
White box

- The module architecture is analyzed
  - Easier to look for errors
  - Can be limited for complex systems



Grey box

- The output and internal architecture can be analyzed
  - Compromise between both approaches



White/grey/black box: Effort

Challenge	**Black**	**White**	**Grey**
Creation of a reference model	High	No	Middle
Addition of monitors and assertions	No	High	Low
Bug tracking from output to source	High	Low	Low
Verification of features implementation	High	Low	Low

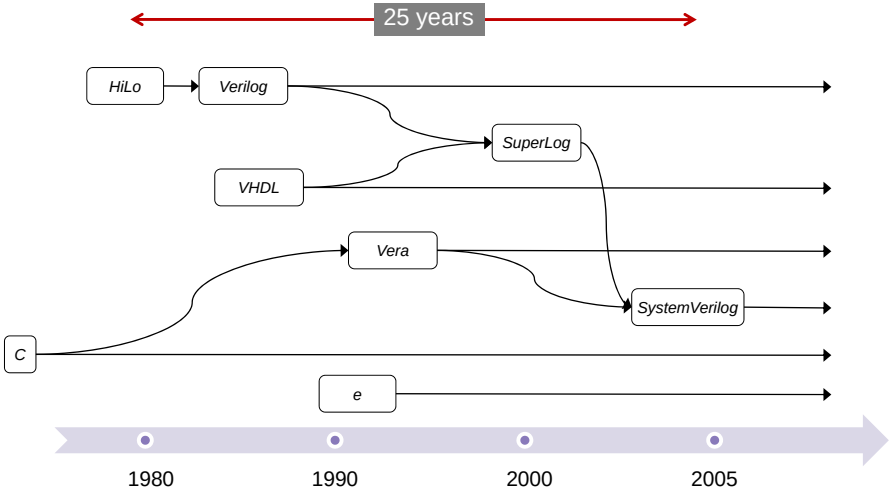
Emulation-based verification

- Issue with simulation: execution time can be huge
- Solution: Exploit FPGAs to accelerate the process
- Difficulties:
  - Placement and route computing time
  - Observability (number and time of windows)
  - Assertions
  - Quite well for black box verification
- Commercial solutions:
  - Veloce (Siemens)
  - ZeBu (Synopsis)
  - Palladium (Cadence)

Languages for verification

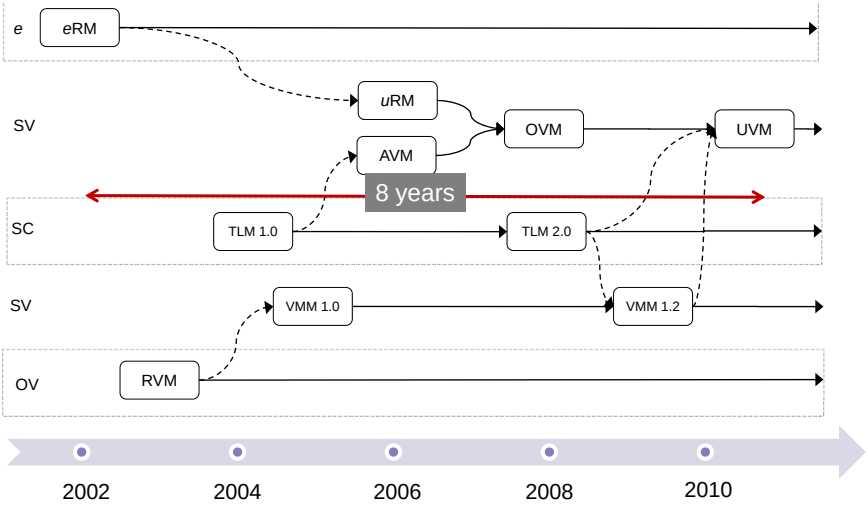
- Small to middle-size projects
  - VHDL
- Middle to high-size projects
  - Needs for:
    - High-level models (transaction)
    - Random stimuli generation
    - Assertion-based verification
  - Current options
    - SystemVerilog
    - SystemC
    - Python
    - Assertions: PSL (*Property Specification Language*), or SVA

Origin of languages



Functional Verification of Today's and Tomorrow's SoCs, Janick Bergeron, Synopsys, 2011

Origin of methodologies

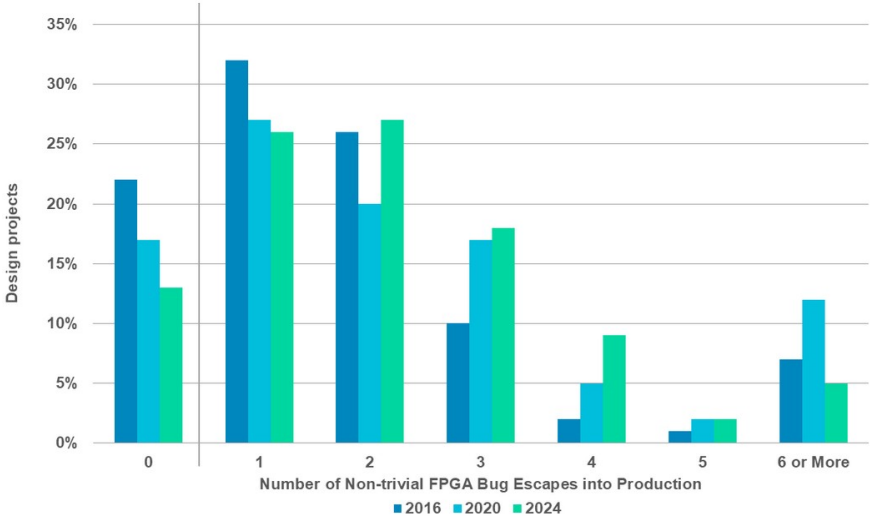


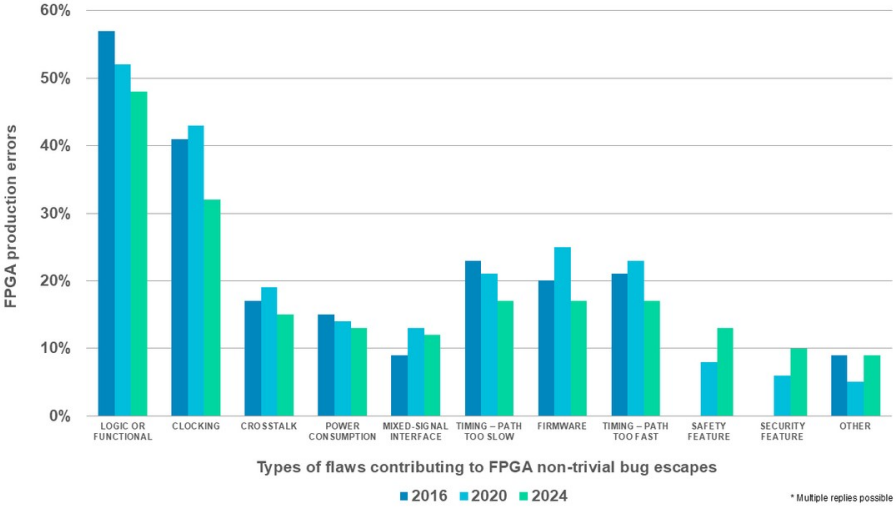
Functional Verification of Today's and Tomorrow's SoCs, Janick Bergeron, Synopsys, 2011

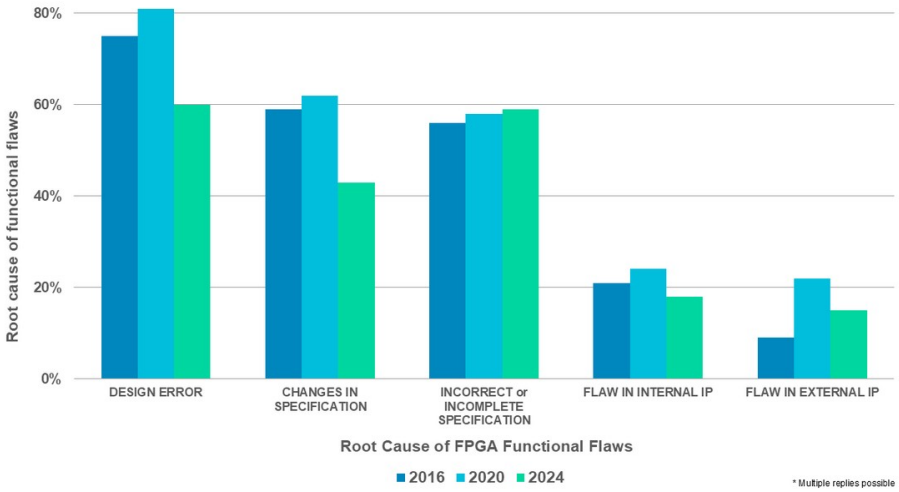
Study on worldwide usages for design and verification

- *2024 Wilson Research Group FPGA functional verification trend report*
- Published by Siemens
- Comparison between 2016 et 2024 (started in 2012)
- Designs FPGA and ASIC (but let's focus on FPGA)

https://resources.sw.siemens.com/en-US/white-paper-2024-wilson-research-group-fpga-functional-verification-trend-report/


35%
30%
25%
20%
15%
10%
5%
0%
Design projects
0
1
2
3
4
5
6 or More
Number of Non-trivial FPGA Bug Escapes into Production
2016
2020
2024


60%
50%
40%
30%
20%
10%
0%
FPGA production errors
LOGIC OR FUNCTIONAL
CLOCKING
CROSSTALK
POWER CONSUMPTION
MIXED-SIGNAL INTERFACE
TIMING – PATH TOO SLOW
FIRMWARE
TIMING – PATH TOO FAST
SAFETY FEATURE
SECURITY FEATURE
OTHER
Types of flaws contributing to FPGA non-trivial bug escapes
2016
2020
2024
* Multiple replies possible

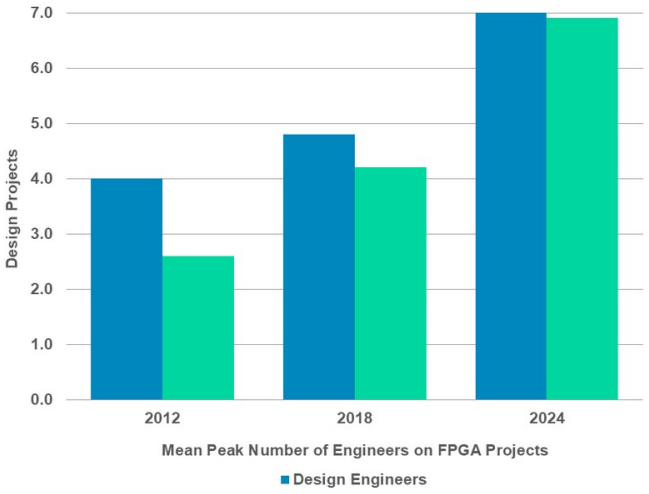

80%
60%
40%
20%
0%
Root cause of functional flaws
DESIGN ERROR
CHANGES IN SPECIFICATION
INCORRECT or INCOMPLETE SPECIFICATION
FLAW IN INTERNAL IP
FLAW IN EXTERNAL IP
Root Cause of FPGA Functional Flaws
2016
2020
2024
* Multiple replies possible

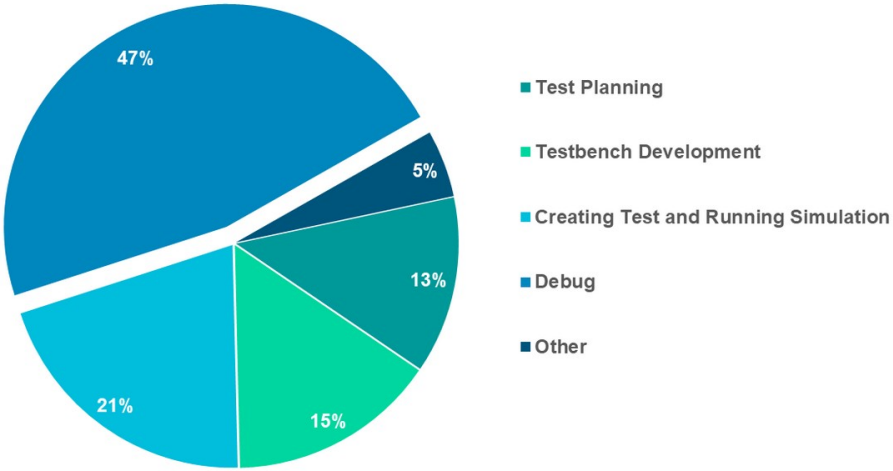
4.7%

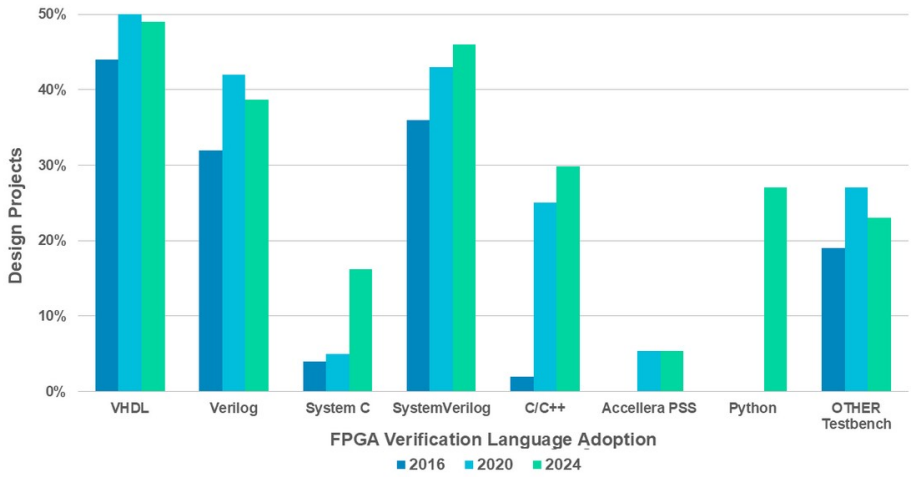
Increase in design engineers since 2012

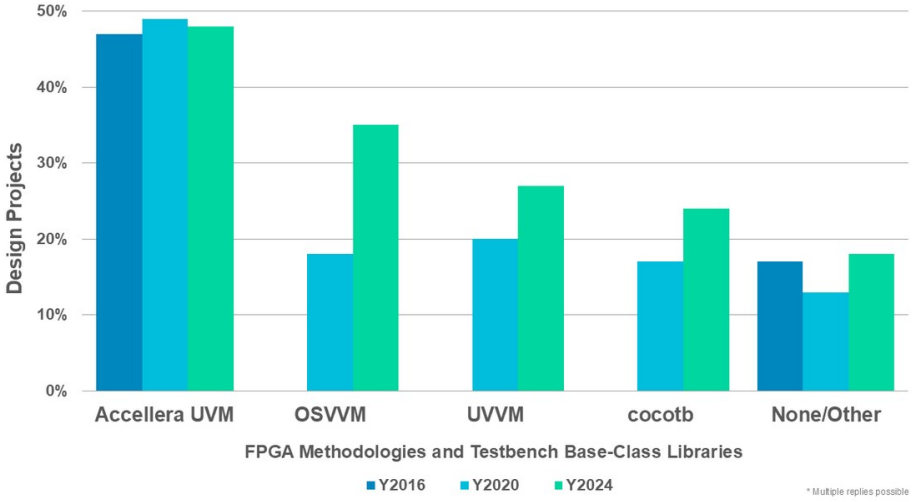
8.5%

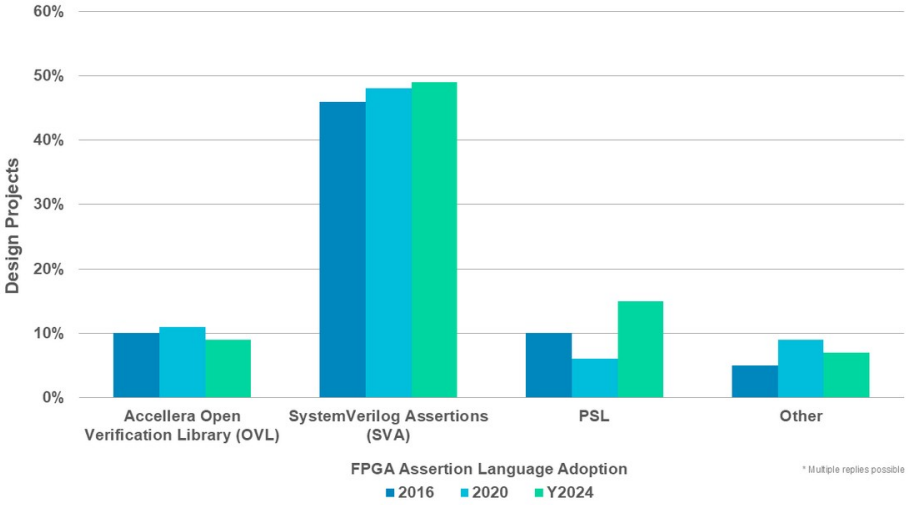
Increase in verification engineers since 2012

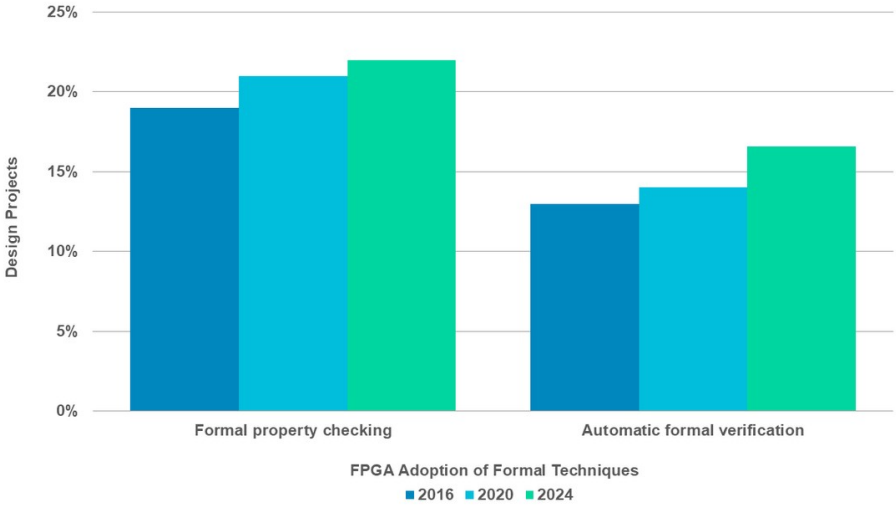


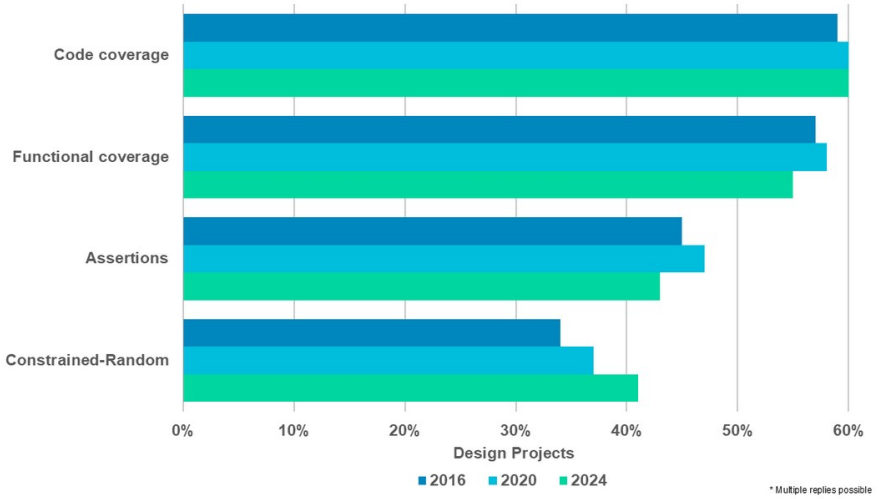

47%
5%
13%
15%
21%
Test Planning
Testbench Development
Creating Test and Running Simulation
Debug
Other


50%
40%
30%
20%
10%
0%
Design Projects
VHDL
Verilog
System C
SystemVerilog
C/C++
Accellera PSS
Python
OTHER Testbench
FPGA Verification Language Adoption
2016
2020
2024

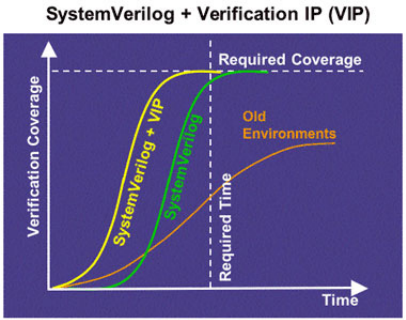

50%
40%
30%
20%
10%
0%
Design Projects
Accellera UVM
OSVVM
UVVM
cocotb
None/Other
FPGA Methodologies and Testbench Base-Class Libraries
Y2016
Y2020
Y2024
* Multiple replies possible


60%
50%
40%
30%
20%
10%
0%
Design Projects
Accellera Open Verification Library (OVL)
SystemVerilog Assertions (SVA)
PSL
Other
FPGA Assertion Language Adoption
2016
2020
Y2024
* Multiple replies possible


25%
20%
15%
10%
5%
0%
Design Projects
Formal property checking
Automatic formal verification
FPGA Adoption of Formal Techniques
2016
2020
2024


Code coverage
Functional coverage
Assertions
Constrained-Random
0%
10%
20%
30%
40%
50%
60%
Design Projects
2016
2020
2024
* Multiple replies possible

SystemVerilog vs. VHDL



Source: http://www.design-reuse.com/articles/7844/verification-ip-verification-ip--part-2-by-dr-aart-de-geus.html

SystemVerilog

- Object-oriented
  - Inheritance
  - Dynamic creation of objects
- Interfaces
- Reusability
- Random generation
  - Unconstrained
  - Constrained
- Assertions
- Coverage
- Ideal for random-based coverage-driven verification
- A set of methodologies

SystemVerilog Methodologies

- Synopsys: VMM (Verification Methodology Manual)
  - Version 1.2
  - `https://www.synopsys.com/verification/vmm-central.html`
- Mentor: AVM (Advanced Verification Methodology)
  - Version 3.0
  - Terminated
- Ensemble: OVM (Open Verification Methodology)
  - Version 2.1.2 $\rightarrow$ UVM
  - `http://www.ovmworld.org/`
- UVM (Universal Verification Methodology)
  - Unification de VMM et OVM
  - Accellera standard!
  - Version 2017-1.0
  - `https://www.accellera.org/downloads/standards/uvm`

Abstraction levels

Abstraction levels

TLM methodology

- For complex systems, the *Transaction Level Modeling* offers:
  - Better modularity
  - Better reusability
  - Decoupling of RTL against other abstraction levels
- Concept:
  - Go to higher abstraction levels as possible
  - Be as much as possible in a software world
- SystemVerilog is well-suited
- VHDL approaches that with UVVM and OSVVM
- If formal is too complex, then switch to TLM