ELEC 305

Digital System Design Lab

Fall 2024

Lecture 3:

Simulation / Verification

Recap

- At this point, we know how to **<u>describe</u>** a simple circuit using VHDL based on a given set of specifications
- We know how to use Vivado to automatically **<u>synthesize</u>** that circuit and **<u>implement</u>** it on the FPGA
- New → We can also use VHDL to generate test signals for our circuits to **<u>simulate</u>** their behavior rather than testing the system directly on the FPGA hardware
- Today we'll have a look at the simulation (and more generally the verification) aspect, which will guide us when we start working on more complicated systems

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity coffeemaker is
  Port ( clk : in  STD_LOGIC;
         led : out STD_LOGIC;
         sw  : in STD_LOGIC
       );
end coffeemaker;

architecture Behavioral of coffeemaker is
  signal pulse : std_logic := '0';
  signal count : integer range 0 to 199999999 := 0;
begin
  process(clk, sw)
  begin
    if sw = '0' then
      pulse <= '0';
    elsif clk'event and clk = '1' then
      if count = 199999999 then
        count <= 0;
        pulse <= not pulse;
      else
        count <= count + 1;
      end if;
    end if;
  end process;

  led <= pulse;
end Behavioral;
```

Outline

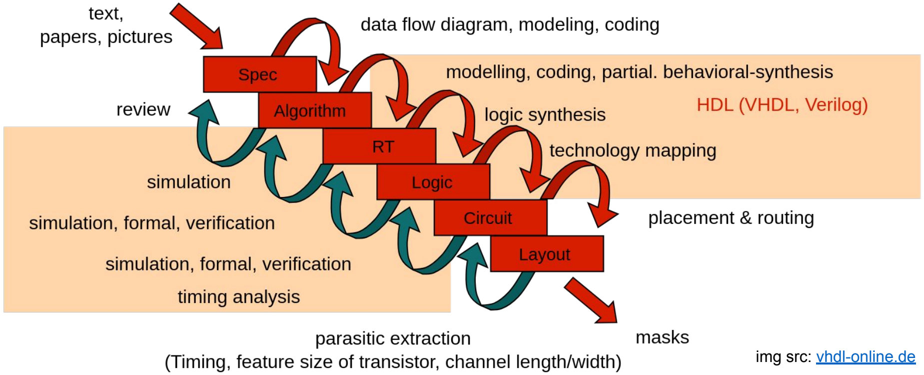
- Intro to simulation and verification in digital circuits
- Verification approaches: why is it a hard problem?
- Using VHDL for simulation
- Vivado’s simulator and open-source options: GHDL + GTKWave

Outline

- Intro to simulation and verification in digital circuits
- Verification approaches: why is it a hard problem?
- Using VHDL for simulation
- Vivado’s simulator and open-source options: GHDL + GTKWave

Simulation and Verification

- Waterfall →→→→→
  The specs dictate how the design should turn out, and simulations (+ other techniques) are used to verify that



img src: vhdl-online.de

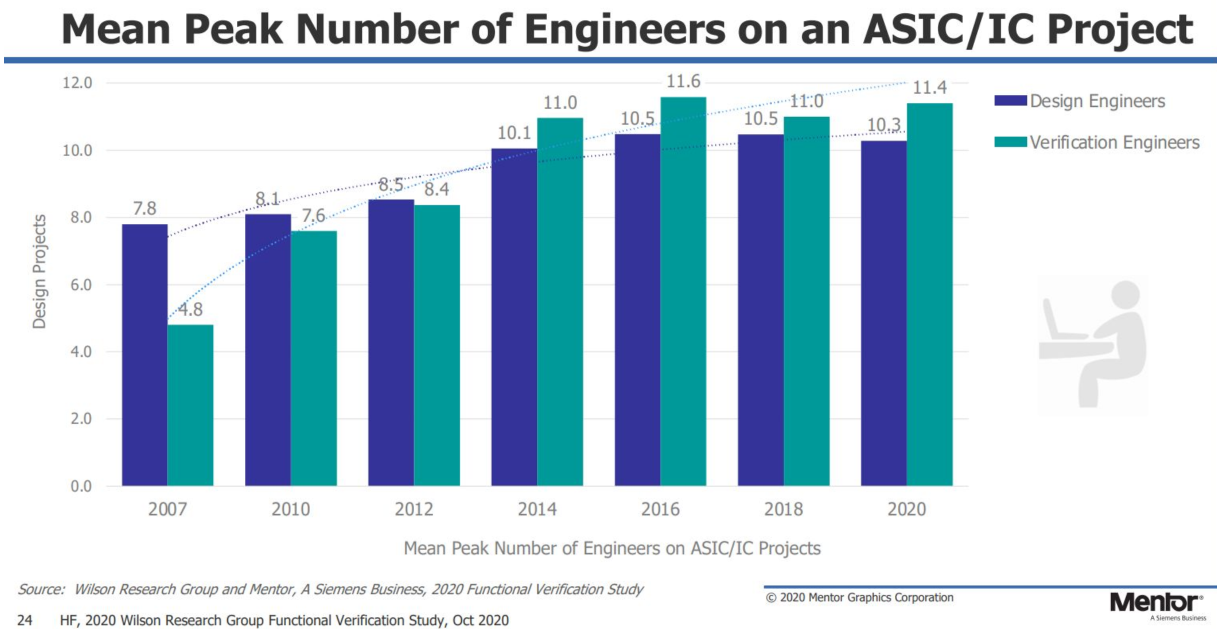
- Verification on hardware (e.g., FPGA + logic analyzer) is also done, but simulation can cover significantly more cases

- Simulation is also sometimes the only feasible option for complicated designs and for debugging internal signals (can't put scope probes on signals inside the FPGA)

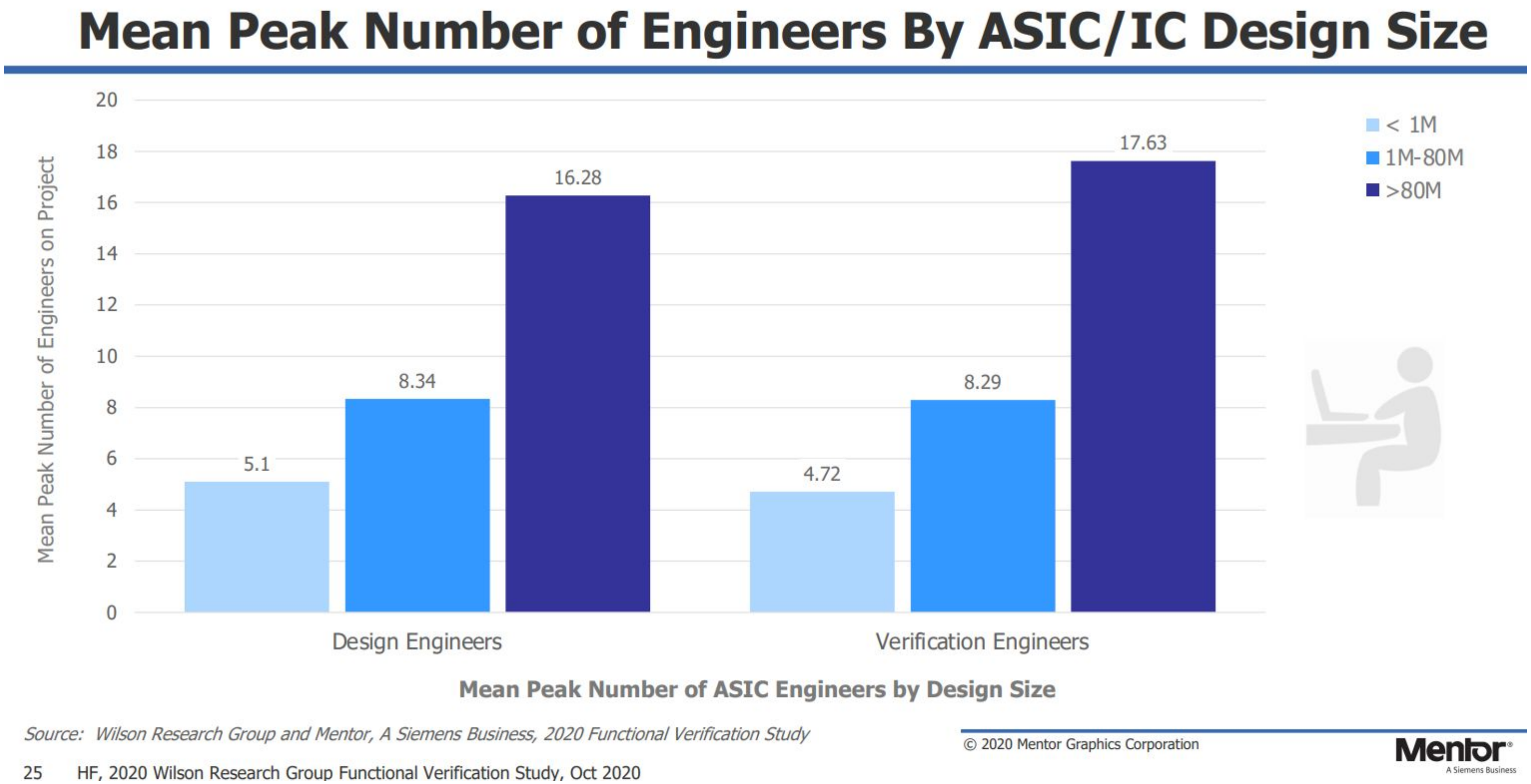
- Most projects spend >50% of the total engineering effort in verification. There are even dedicated verification companies, it's an industry on its own!

Simulation and Verification

- The "Wilson Research Group Functional Verification Study" (WRG-FVS) by Mentor (now part of Siemens) keeps tabs on sector dynamics
- See how the number of verification engineers surpassed the number of design engineers in projects over the years!

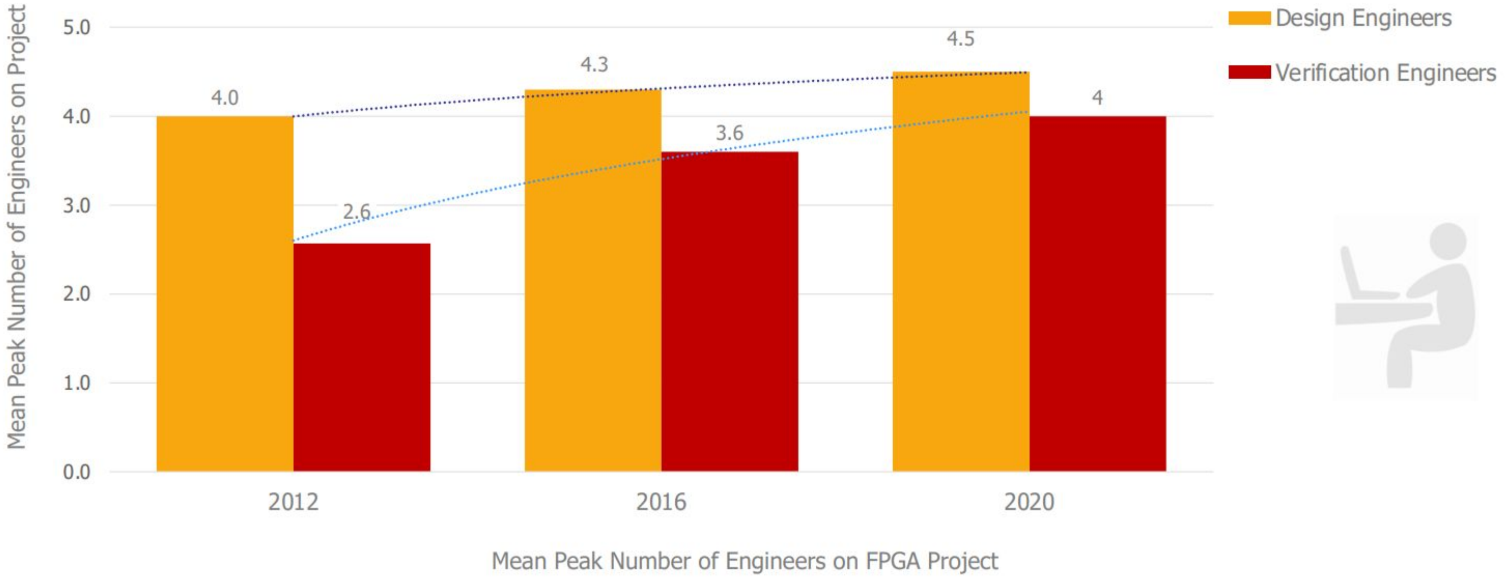


Simulation and Verification



Simulation and Verification

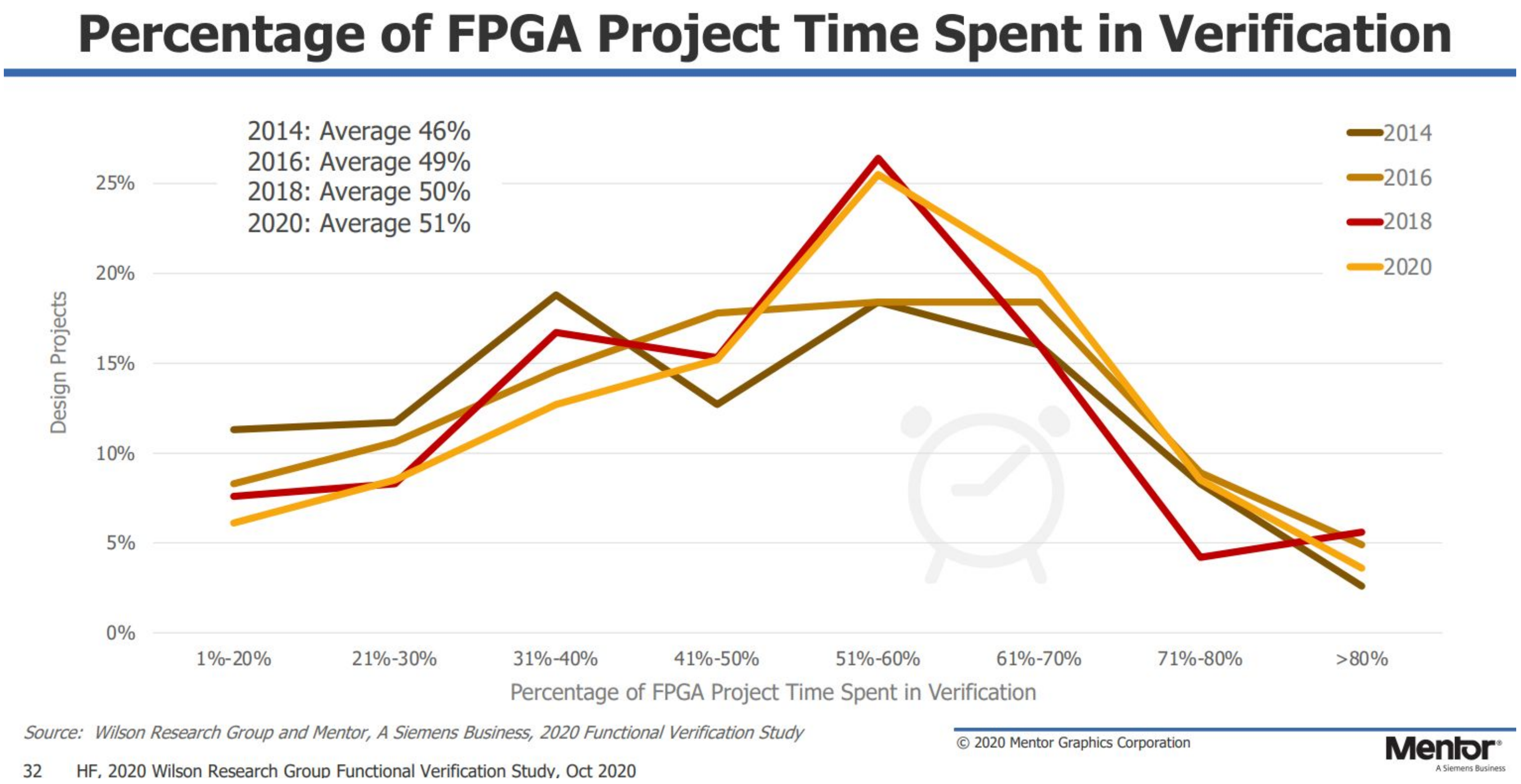
Mean Peak Number of Engineers on a FPGA Project



Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study

26 HF, 2020 Wilson Research Group Functional Verification Study, Oct 2020

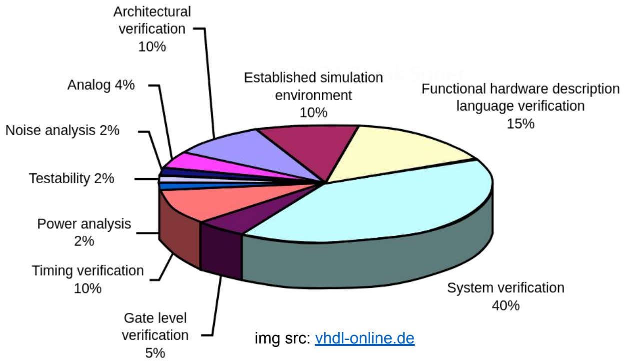
Simulation and Verification



Simulation and Verification

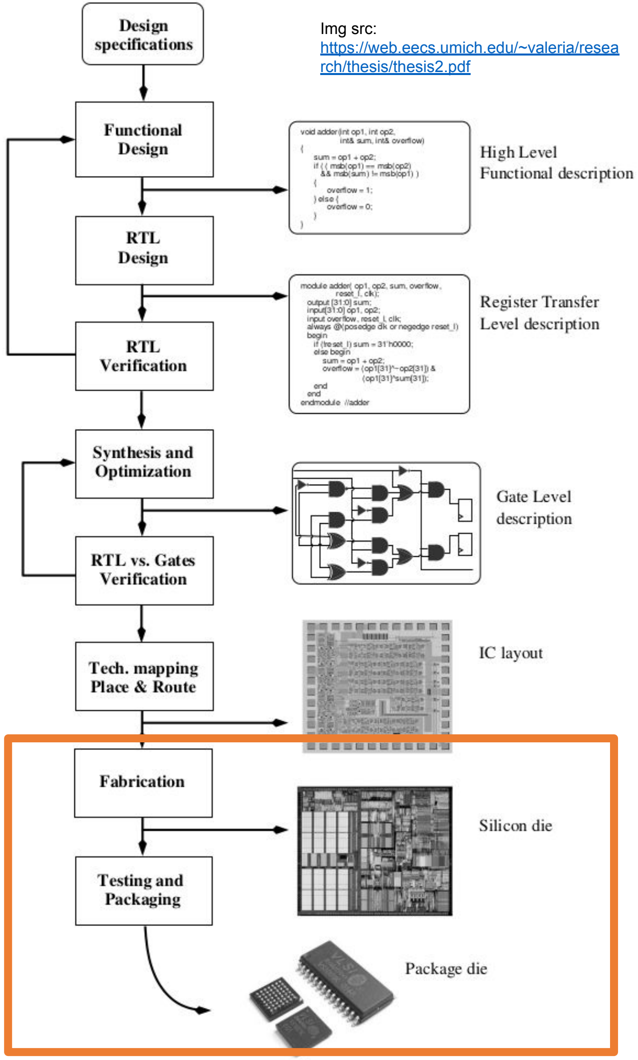
- There are many levels of verification: behavioral sim (no timing) is the first / fastest / simplest and it is inaccurate for timing. Then post-synth and post-impl are more accurate, but they take more time

- If we were to keep going after the FPGA deployment phase and fabricate this circuit (i.e., ASIC), there would be even further testing too.



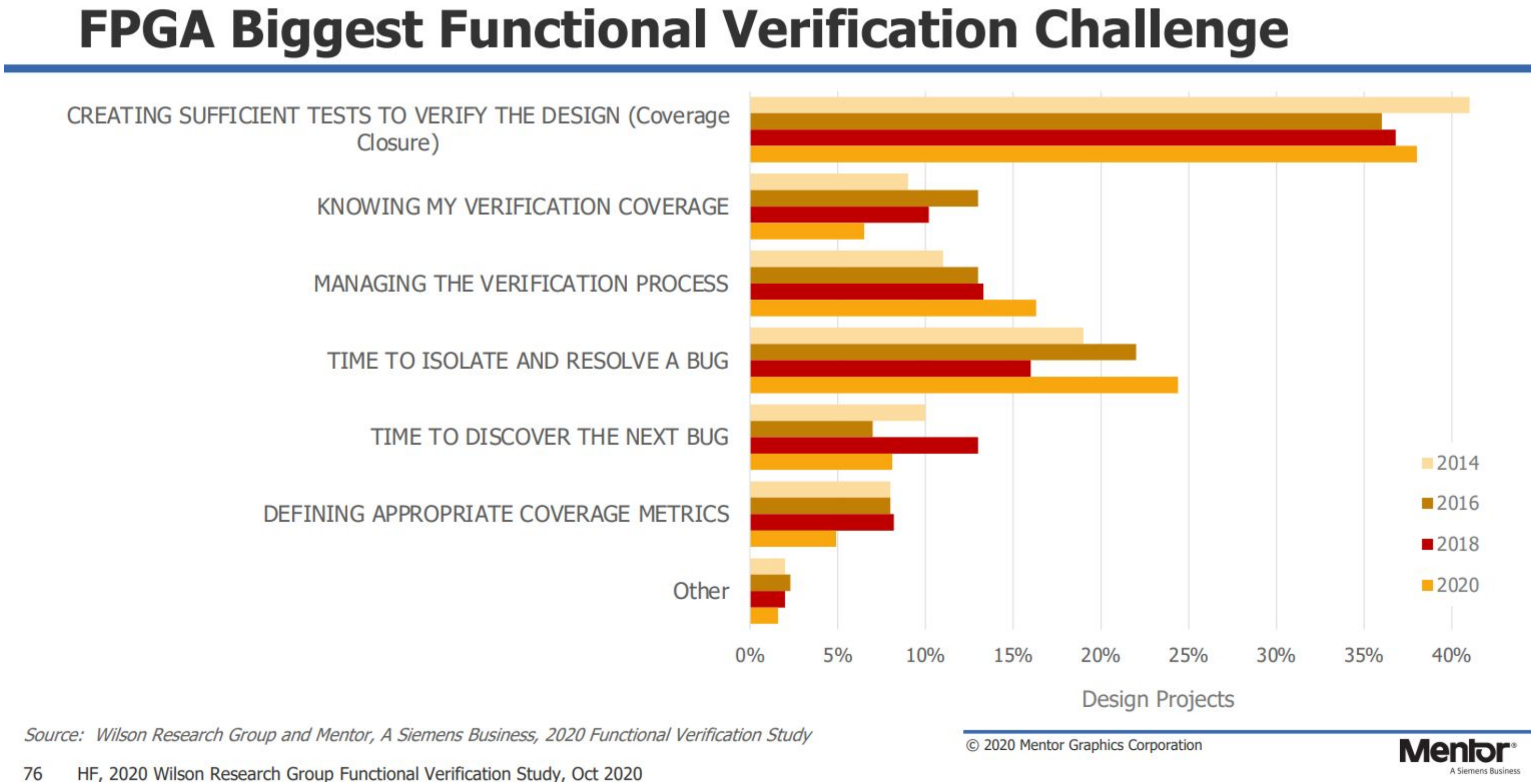
img src: vhdl-online.de

See the "[magic smoke test]()" for fun.

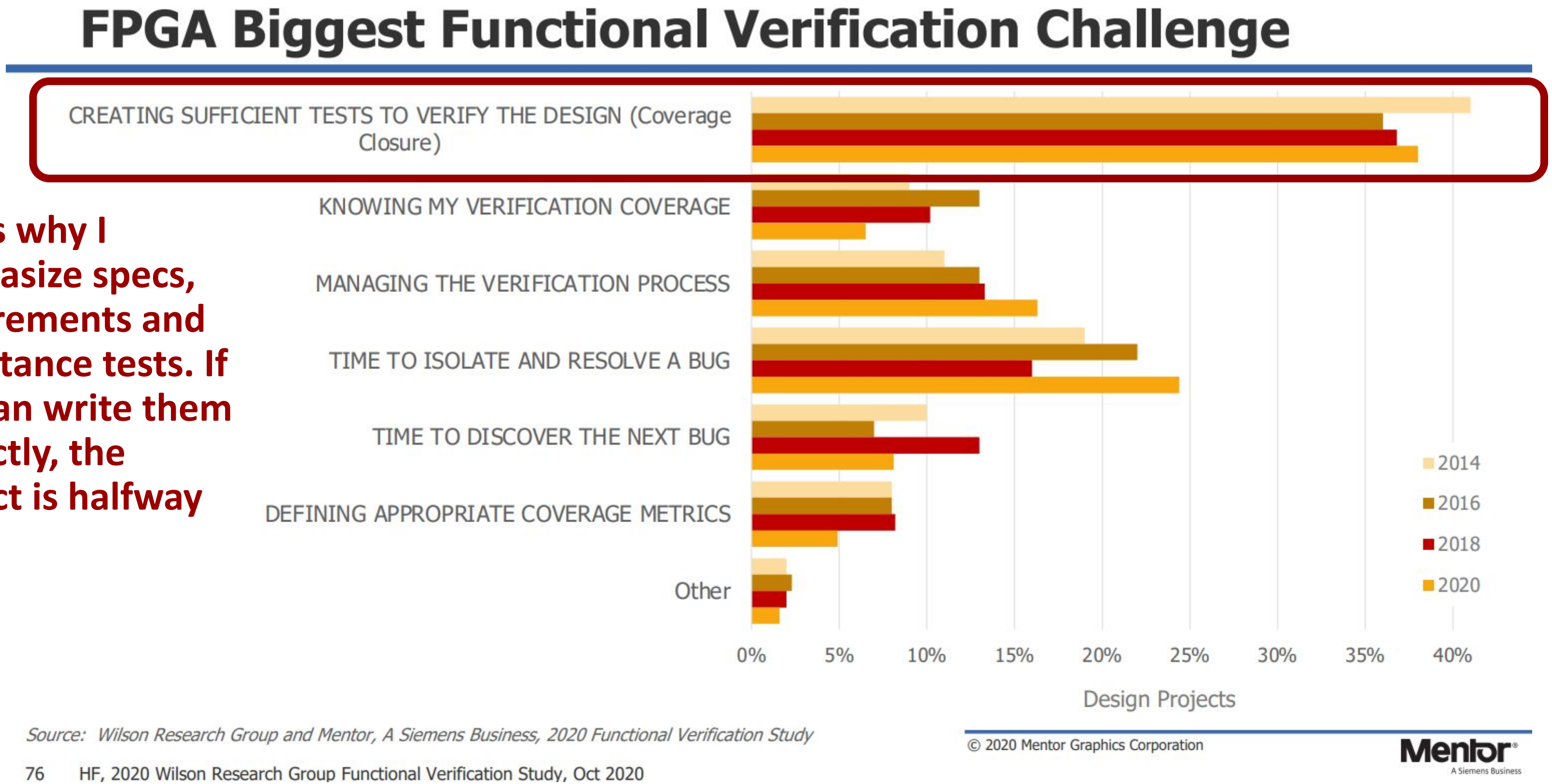


Img src: https://web.eecs.umich.edu/~valeria/research/thesis/thesis2.pdf

Simulation and Verification



Simulation and Verification

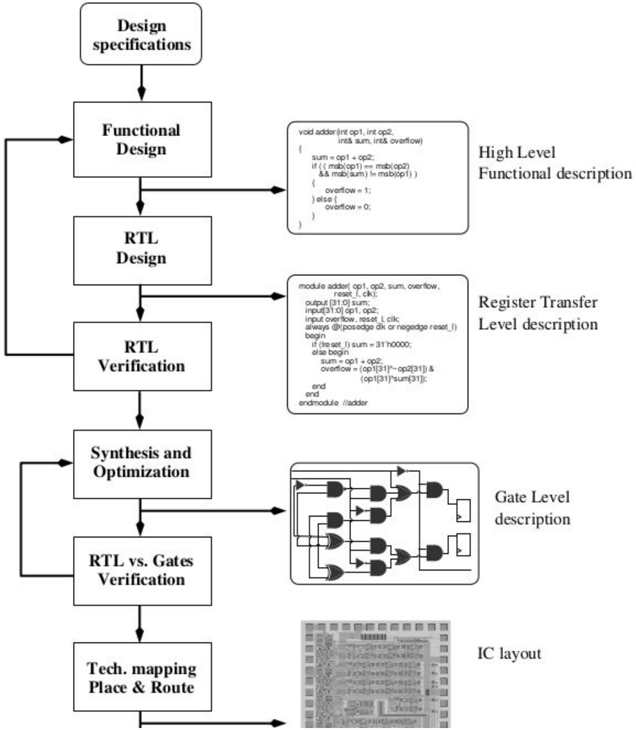


This is why I emphasize specs, requirements and acceptance tests. If you can write them correctly, the project is halfway done!

Simulation and Verification

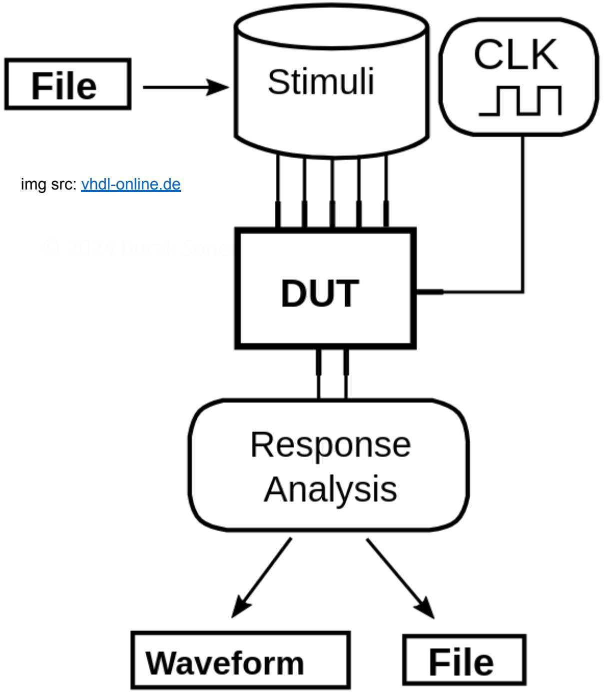
Picture this digital design workflow chart in the following way:

- Design specifications stage → abstract representation of your circuit, in words and numbers (e.g., “LED should blink at 1 Hz”)
- As you go down, you transform that representation,
  - first into functional software (e.g., with C / Python)
  - then into an HDL,
  - then into a gate netlist (synthesis),
  - and finally into an FPGA bitstream (implementation + write_bitstream).
- After each of those steps, you have the option of “running” the circuit with certain stimuli and checking outputs.
- All of these runs would be simulations, we just don’t call the final step that runs on the FPGA (or the ASIC) “simulation” *per se*, because that is the intended outcome of the project.



Simulation and Verification

- We run "testbenches" in simulations
- A testbench consists of…
  - input stimuli for the device under test (DUT)
  - a mapping between simulation signals and the DUT ports
- The simulator tool takes the testbench stimuli, the DUT description (in VHDL / Verilog) and runs something called "discrete event simulation" (DES) to calculate the outputs so you can cross them with specs
- [DES](#) is a generic concept for simulating discontinuous systems, we just employ it here, nothing new.



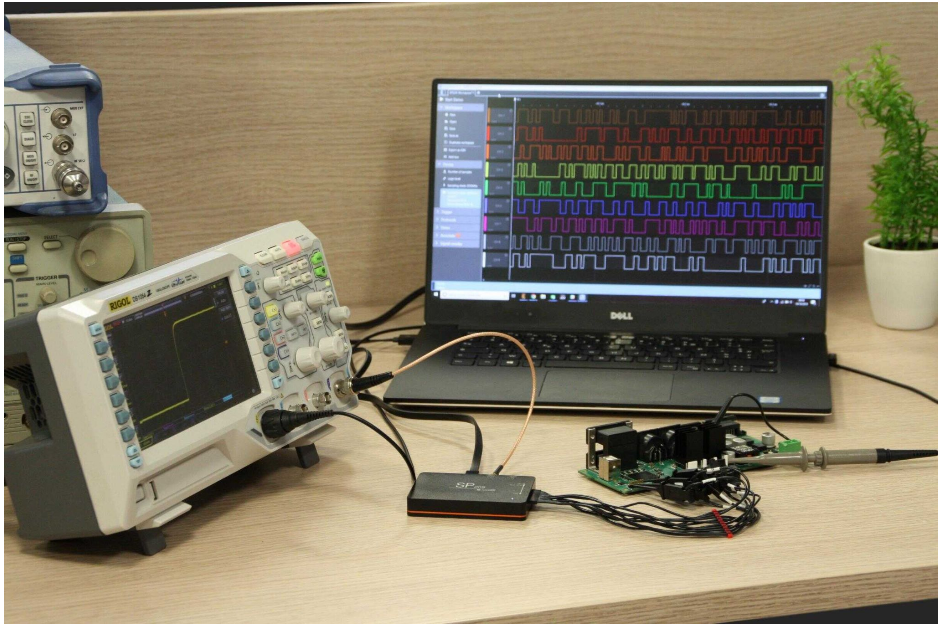
img src: vhdl-online.de

Simulation and Verification

- Why is it called a “testbench”?

- We test final deployed digital hardware (on FPGAs and ASICs) like this (right), with programmed (either via buttons on the tool or via a PC) waveform generators and logic analyzers on a bench.

- The simulator mimics this on earlier stages of the design workflow, so people called it a “testbench”.



img src: https://www.ikalogic.com/assets/images/galleries/sp209/0%20Logic%20analyzer%20usecase.jpg

Simulation and Verification

- For instance let’s consider our blinking LED task from Lab 1, a checkoff list looked like this:

- SW# are the stimuli
- LEDs are the outputs
- Seq1-12 constitute all logical combinations of the inputs
- The simulation testbench for this lab would basically mimic what I did during the lab hours → “stimulate” the switches and record how the LEDs behave

	pwr	E/L	pw3	pw2	pw1	E/L	pw3	pw2	pw1	
Seq	**SW0**	**SW1**	**SW4**	**SW3**	**SW2**	**LED1**	**LED4**	**LED3**	**LED2**	**LED0**
1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF
2	ON	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF
3	ON	OFF	OFF	OFF	ON	OFF	OFF	OFF	ON	OFF
4	ON	OFF	OFF	ON	OFF	OFF	OFF	ON	OFF	OFF
5	ON	OFF	OFF	ON	ON	OFF	OFF	ON	ON	blinking at 1 Hz
6	ON	OFF	ON	OFF	OFF	OFF	ON	OFF	OFF	blinking at 1 Hz
7	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	blinking at 1 Hz
8	ON	OFF	ON	ON	OFF	OFF	ON	ON	OFF	OFF
9	ON	OFF	ON	ON	ON	OFF	ON	ON	ON	OFF
10	OFF	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF
11	OFF	ON	ON	OFF	ON	ON	ON	OFF	ON	OFF
12	ON	ON	ON	OFF	ON	ON	ON	OFF	ON	blinking at 2 Hz

Simulation and Verification

- Why didn't we simulate Lab 1?

- Simulators try to capture high-resolution timing information and extremely fast transients like gate delays etc., so they work at high time resolutions like 1 ps.

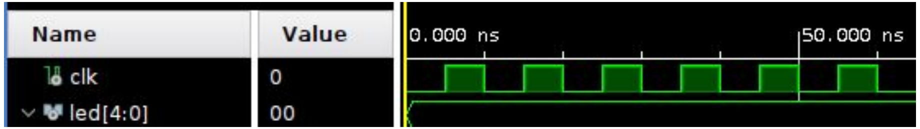
- We're trying to see if an LED blinks at 1 Hz or 2 Hz for ≈10 different switch configurations, that means we need to monitor at least a full period, which means at least ≈10 seconds.
  →That's at least 10^13 simulation steps when the time resolution is 1ps!!
  - Even if the simulation ran in reasonable time, the (uncompressed) simulation record file for this small experiment would be >10 GB !!

- It's possible to enlarge the step time, but convergence issues start after 1ns since the gate models aren't valid for larger steps, so you can't really run the simulation in that scenario
  (you might know "max step size" issues in MATLAB, this is similar, the solver breaks down. See ELEC518 for more on this).

Simulation and Verification

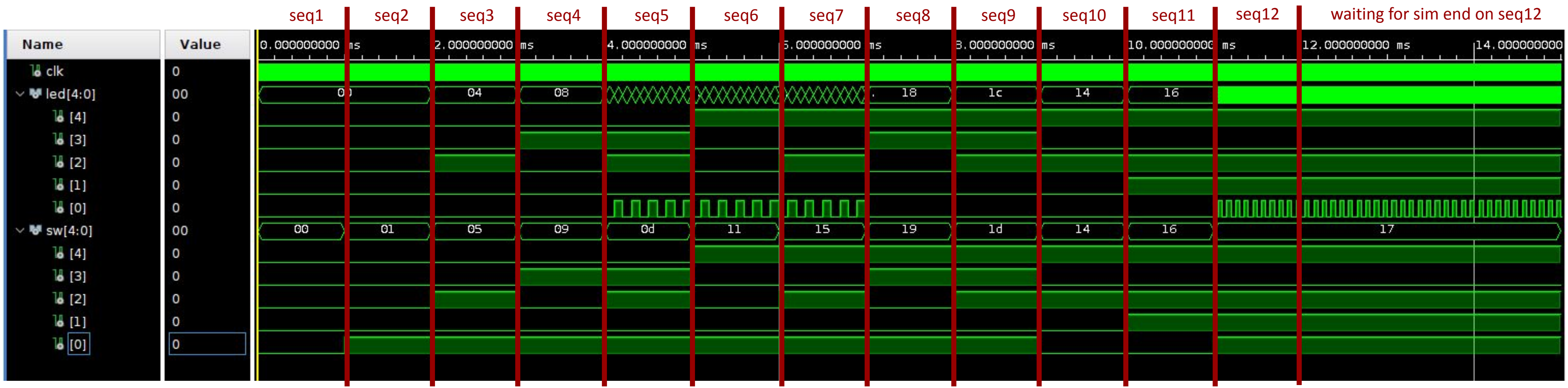
- Let’s change Lab 1 a bit and make it feasible for us to do the checkoff in simulation
  → 5 kHz blink rather than 1 Hz, simulation time of 15 ms (≈10 MB), 1 ms waits between seqs

- The clock is still very fast compared to the rest of the circuit, but now at least we can simulate the desired behavior →



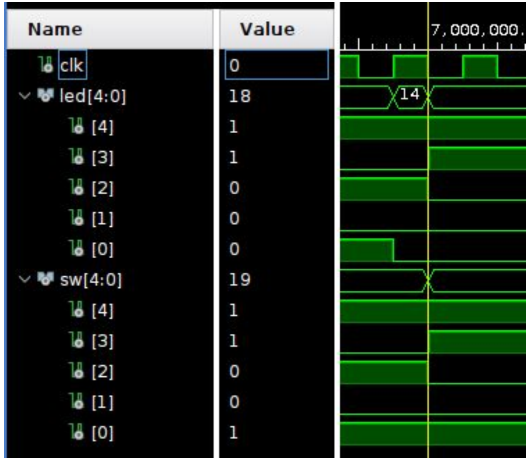
Simulation and Verification

	pwr	E/L	pw3	pw2	pw1	E/L	pw3	pw2	pw1	
Seq	SW0	SW1	SW4	SW3	SW2	LD1	LD4	LD3	LD2	LED0
1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF
2	ON	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF
3	ON	OFF	OFF	OFF	ON	OFF	OFF	OFF	ON	OFF
4	ON	OFF	OFF	ON	OFF	OFF	OFF	ON	OFF	OFF
5	ON	OFF	OFF	ON	ON	OFF	OFF	ON	ON	blinking at 5 kHz
6	ON	OFF	ON	OFF	OFF	OFF	ON	OFF	OFF	blinking at 5 kHz
7	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	blinking at 5 kHz
8	ON	OFF	ON	ON	OFF	OFF	ON	ON	OFF	OFF
9	ON	OFF	ON	ON	ON	OFF	ON	ON	ON	OFF
10	OFF	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF
11	OFF	ON	ON	OFF	ON	ON	ON	OFF	ON	OFF
12	ON	ON	ON	OFF	ON	ON	ON	OFF	ON	blinking at 10 kHz

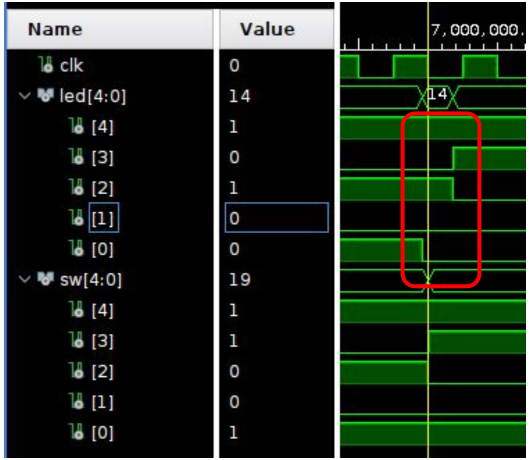


Simulation and Verification

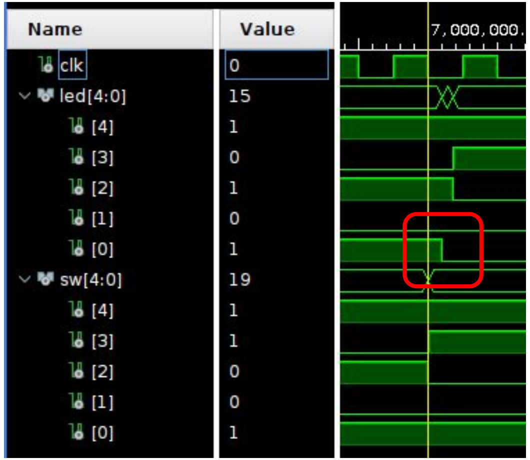
- That was behavioral sim., post-synth and post-impl simulations add more accurate timing info
- Zoom in to the 7ms mark, see how the LED responses are delayed a bit in post-synth, and there's further delay on the LED0 line after implementation. Post-impl is the most accurate.



behavioral



post-synth



post-impl

Outline

- Intro to simulation and verification in digital circuits
- Verification approaches: why is it a hard problem?
- Using VHDL for simulation
- Vivado’s simulator and open-source options: GHDL + GTKWave

Outline

- Intro to simulation and verification in digital circuits
- Verification approaches: why is it a hard problem?
- Using VHDL for simulation
- Vivado’s simulator and open-source options: GHDL + GTKWave

Verification Approaches

- Lab 1 verification was easy enough just now… so why is verification hard?
- Mainly because verification effort grows fast vs. the increase in design complexity
- E.g., consider the case in which we need a 6-bit password instead of a 3-bit password for Lab 1. We had to test for 8 combinations with a 3-bit password, with 6-bits we need to test for 64.
- Once you start factoring in different aspects, things start getting out of hand if you're planning to continue on this exhaustive testing approach, especially with multiple clock/control paths and complex arithmetic

	pwr	E/L	pw3	pw2	pw1	E/L	pw3	pw2	pw1	
Seq	SW0	SW1	SW4	SW3	SW2	LED1	LED4	LED3	LED2	LED0
1	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF
2	ON	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF
3	ON	OFF	OFF	OFF	ON	OFF	OFF	OFF	ON	OFF
4	ON	OFF	OFF	ON	OFF	OFF	OFF	ON	OFF	OFF
5	ON	OFF	OFF	ON	ON	OFF	OFF	ON	ON	blinking at 1 Hz
6	ON	OFF	ON	OFF	OFF	OFF	ON	OFF	OFF	blinking at 1 Hz
7	ON	OFF	ON	OFF	ON	OFF	ON	OFF	ON	blinking at 1 Hz
8	ON	OFF	ON	ON	OFF	OFF	ON	ON	OFF	OFF
9	ON	OFF	ON	ON	ON	OFF	ON	ON	ON	OFF
10	OFF	OFF	ON	OFF	ON	OFF	ON	OFF	ON	OFF
11	OFF	ON	ON	OFF	ON	ON	ON	OFF	ON	OFF
12	ON	ON	ON	OFF	ON	ON	ON	OFF	ON	blinking at 2 Hz

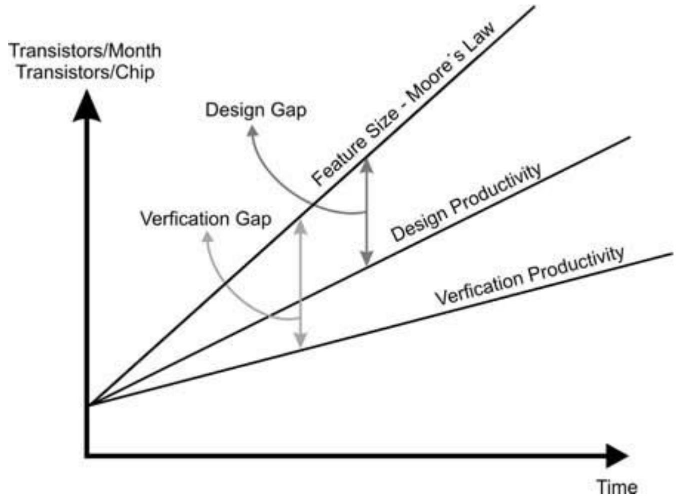
Verification Approaches

- In more formal terms, verifying a complex design (similar to verifying a software program) is a problem that is "NP-hard"

- While it is certainly not unsolvable, sometimes exhaustive testing is infeasible since total runtime can be as much as **years** even on supercomputers

- Therefore, clever approaches are always sought for

- These range from simply designing a good simulation testbench that represents the verification space well and finds possible bugs, to more complicated algorithmic solutions that can augment or even replace such brute force simulation-testing



img src: http://www.scielo.org.ar/scielo.php?script=sci_arttext&pid=S0327-07932007000100013

Verification Approaches

- So we've talked about two somewhat naive approaches:
  - 1) exhaustive testing of "all possible scenarios" in the verification space
  - 2) coming up with fewer clever tests that represent the whole space those scenarios cover

- There are at least two other prominent options:
  - *Intelligent verification:* can be briefly summarized as an adaptive version of (2), where an algorithm searches or optimizes for representative tests **as** the tests are running and the outputs are analyzed. For instance, you do 1 test, see the results, design the next test so that it tests a maximally different part of the verification space, and so on and so forth until you cover as much of the space as possible with as few tests as possible.
  - *Formal verification:* rather than simulating possible scenarios and interpreting them, you try to model the system you designed mathematically so that you can try to rigorously prove that the system works as intended via assertions.

Verification Approaches

- Intelligent verification is a growing field, but it's relatively new right now so it's not prevalent in the industry

- **Formal verification** is motivated by the following idea: "You will not be able to run a truly exhaustive test for most practical designs, and an incomplete exhaustive test can be misleading (see the example described on the right), so you need rigorous **proof** to truly verify that your design works as intended".

- This is a very promising field that has made it into standard practices (Vivado supports one method called "Equivalency Checking"), but it's very specialized work since the methods typically have constraints that need to be "tuned" for the design. See this reddit thread for formal verification "lore" in the industry.

Formal Verification In Industry (I) 3

Exhaustiveness

In mathematics, a general proposition can't be *proved* by testing many possible cases. A rigorous proof is something different.

Sometimes even a huge weight of numerical evidence can be misleading. For example, Littlewood proved in 1914 that $\pi(n) - li(n)$ changes sign infinitely often, where $\pi(n)$ is the number of primes $\leq n$ and

$$li(n) = \int_0^n du/ln(u)$$

This came as a surprise since not a single sign change had been found despite extensive testing of values up to 10^{10}. (In the days before computers.)

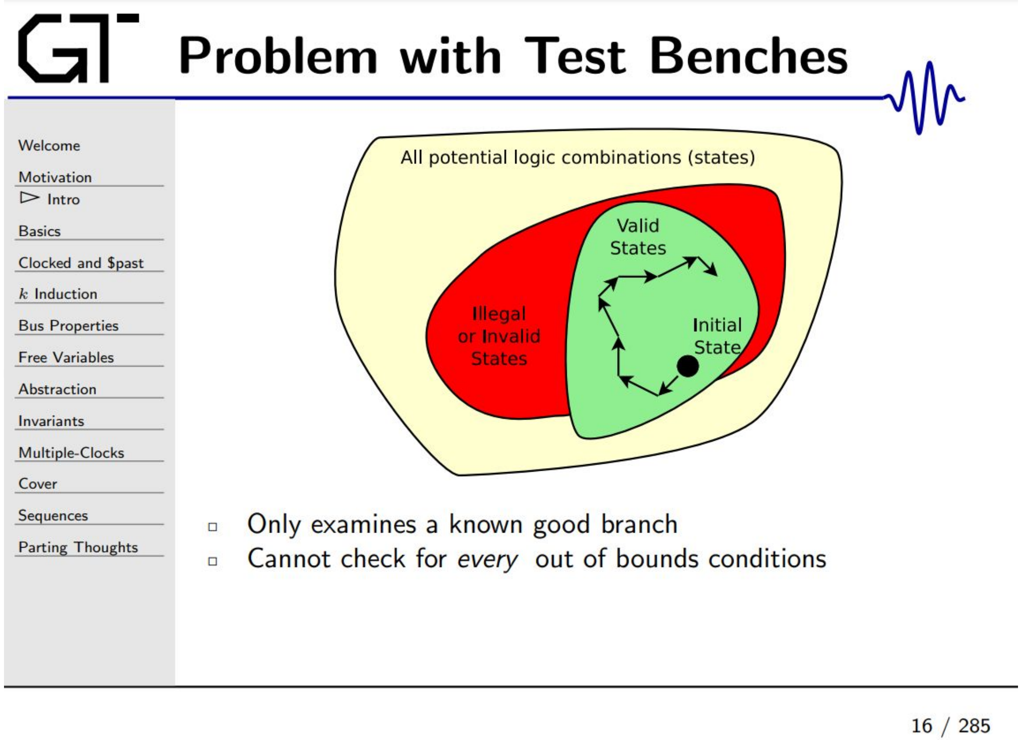
Similarly, extensive testing of hardware or software may still miss errors that would be revealed by a formal proof.

John Harrison Intel Corporation, 4 September 1999

Verification Approaches

- Good formal verification methods usually saves people A LOT of time and money since "exhaustive testbenching" is extremely infeasible in some complex projects (e.g., think of a Pentium CPU project)

- However coming up with such methods for general use cases is also very hard. Typically experts get contracted specifically for a project and devises / tunes methods accordingly. See one such expert here (GT LLC) →→→→→→→→→



Outline

- Intro to simulation and verification in digital circuits
- Verification approaches: why is it a hard problem?
- Using VHDL for simulation
- Vivado’s simulator and open-source options: GHDL + GTKWave

Outline

- Intro to simulation and verification in digital circuits
- Verification approaches: why is it a hard problem?
- Using VHDL for simulation
- Vivado's simulator and open-source options: GHDL + GTKWave

Using VHDL for Simulation (alongside description and synthesis)

- VHDL was originally devised for just *describing* circuits, people *read* VHDL descriptions to verify circuit functionality *on paper*, so VHDL was like a documentation format.

- Naturally, two additional uses emerged for VHDL to augment this workflow:

  - *Automatic Synthesis:* Given the VHDL description, generate a circuit design in terms of known components (e.g., the CLBs on the FPGA, stuff that we see on the schematic after implementation)

    - "Logic compilers" were developed: Took VHDL circuit descriptions + component libraries as input, generating FPGA-deployable circuits as output

  - *Simulation*: Given a VHDL description, verify the performance of the circuits that is represented by that description

    - "Logic simulators" were developed: Took VHDL circuit descriptions (DUT) + input stimulus vectors as input, generating DUT responses as output

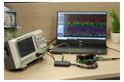
Using VHDL for Simulation (alongside description and synthesis)

- Component libraries for synthesis are developed “offline” by FPGA manufacturers, so synthesis is covered. How do we generate the input stimuli for simulation?

- We can of course manually write vectors of signals for each test case in simulation via some sort of “waveform writing GUI”, but it would be great if we programmatically generate these

- Well, we already know of a “tool” that allows us to programmatically describe something that generates digital signals at its output → VHDL !!

- This is where it gets confusing → **we use VHDL to describe a circuit that generates stimulus signals for the simulation of a DUT that we also described in VHDL** (different source files of course)**.**

- Digital systems naturally have circularities like this but once you get past it you see why this makes sense → by writing the stimulus in VHDL, you are practically generating something like a smaller version of the waveform generator you use on the physical testbench →→ 

Using VHDL for Simulation (alongside description and synthesis)

- For instance the testbench I showed the results for earlier looks like this (Lab 1 but with 5-10 kHz)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity coffeemaker_tb is
-- Port ( );
end coffeemaker_tb;

architecture Behavioral of coffeemaker_tb is
    component coffeemaker_pwd
        Port ( clk : in  STD_LOGIC;
               led  : out STD_LOGIC_VECTOR (4 downto 0);
                sw   : in STD_LOGIC_VECTOR (4 downto 0)
               );
    end component;
    signal clk : STD_LOGIC;
    signal led : STD_LOGIC_VECTOR (4 downto 0);
    signal sw  : STD_LOGIC_VECTOR (4 downto 0);
begin

    dut: entity work.coffeemaker port map (clk => clk, led => led, sw => sw);

    clk_process :process
    begin
        clk <= '0';
        wait for 5 ns;
        clk <= '1';
        wait for 5 ns;
    end process;

    sim_process : process
    begin
        sw <= "00000"; -- seq 1
        wait for 1 ms; -- arbitrary wait.
        sw <= "00001"; -- seq 2
        wait for 1 ms;
        sw <= "00101"; -- seq 3
        wait for 1 ms;
        sw <= "01001"; -- seq 4
        wait for 1 ms;
        sw <= "01101"; -- seq 5
        wait for 1 ms;
        sw <= "10001"; -- seq 6
        wait for 1 ms;
        sw <= "10101"; -- seq 7
        wait for 1 ms;
        sw <= "11001"; -- seq 8
        wait for 1 ms;
        sw <= "11101"; -- seq 9
        wait for 1 ms;
        sw <= "10100"; -- seq 10
        wait for 1 ms;
        sw <= "10110"; -- seq 11
        wait for 1 ms;
        sw <= "10111"; -- seq 12
        wait;
    end process;
end Behavioral;
```

Using VHDL for Simulation (alongside description and synthesis)

This "circuit" that generates signals for simulation will not get synthesized and make it out to the FPGA, so it doesn't need ports

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity coffeemaker_tb is
-- Port ( );
end coffeemaker_tb;

architecture Behavioral of coffeemaker_tb is
    component coffeemaker_pwd
        Port ( clk : in  STD_LOGIC;
               led  : out STD_LOGIC_VECTOR (4 downto 0);
                sw   : in STD_LOGIC_VECTOR (4 downto 0)
              );
    end component;
    signal clk : STD_LOGIC;
    signal led : STD_LOGIC_VECTOR (4 downto 0);
    signal sw  : STD_LOGIC_VECTOR (4 downto 0);
begin

    dut: entity work.coffeemaker port map (clk => clk, led => led, sw => sw);

    clk_process :process
    begin
        clk <= '0';
        wait for 5 ns;
        clk <= '1';
        wait for 5 ns;
    end process;

    sim_process : process
    begin
        sw <= "00000"; -- seq 1
        wait for 1 ms; -- arbitrary wait.
        sw <= "00001"; -- seq 2
        wait for 1 ms;
        sw <= "00101"; -- seq 3
        wait for 1 ms;
        sw <= "01001"; -- seq 4
        wait for 1 ms;
        sw <= "01101"; -- seq 5
        wait for 1 ms;
        sw <= "10001"; -- seq 6
        wait for 1 ms;
        sw <= "10101"; -- seq 7
        wait for 1 ms;
        sw <= "11001"; -- seq 8
        wait for 1 ms;
        sw <= "11101"; -- seq 9
        wait for 1 ms;
        sw <= "10100"; -- seq 10
        wait for 1 ms;
        sw <= "10110"; -- seq 11
        wait for 1 ms;
        sw <= "10111"; -- seq 12
        wait;
    end process;
end Behavioral;
```

Using VHDL for Simulation (alongside description and synthesis)

However, it will connect to the DUT in some way, so it needs its own internal signals (you can picture this like the testbench circuit "hugging" the DUT)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity coffeemaker_tb is
-- Port ( );
end coffeemaker_tb;

architecture Behavioral of coffeemaker_tb is
    component coffeemaker_pwd
        Port ( clk : in STD_LOGIC;
               led : out STD_LOGIC_VECTOR (4 downto 0);
               sw  : in STD_LOGIC_VECTOR (4 downto 0)
             );
    end component;
    signal clk : STD_LOGIC;
    signal led : STD_LOGIC_VECTOR (4 downto 0);
    signal sw  : STD_LOGIC_VECTOR (4 downto 0);
begin

    dut: entity work.coffeemaker port map (clk => clk, led => led, sw =>
sw);

    clk_process :process
    begin
        clk <= '0';
        wait for 5 ns;
        clk <= '1';
        wait for 5 ns;
    end process;

    sim_process : process
    begin
        sw <= "00000"; -- seq 1
        wait for 1 ms; -- arbitrary wait.
        sw <= "00001"; -- seq 2
        wait for 1 ms;
        sw <= "00101"; -- seq 3
        wait for 1 ms;
        sw <= "01001"; -- seq 4
        wait for 1 ms;
        sw <= "01101"; -- seq 5
        wait for 1 ms;
        sw <= "10001"; -- seq 6
        wait for 1 ms;
        sw <= "10101"; -- seq 7
        wait for 1 ms;
        sw <= "11001"; -- seq 8
        wait for 1 ms;
        sw <= "11101"; -- seq 9
        wait for 1 ms;
        sw <= "10100"; -- seq 10
        wait for 1 ms;
        sw <= "10110"; -- seq 11
        wait for 1 ms;
        sw <= "10111"; -- seq 12
        wait;
    end process;
end Behavioral;
```

Using VHDL for Simulation (alongside description and synthesis)

```
library IEEE;
use IEEE.STD_LOGIC_1164 .ALL;

entity coffeemaker_tb  is
--  Port ( );
end coffeemaker_tb ;

architecture Behavioral  of coffeemaker_tb  is
    component coffeemaker_pwd
       Port ( clk  : in   STD_LOGIC;
              led  : out STD_LOGIC_VECTOR  (4 downto 0);
               sw   : in STD_LOGIC_VECTOR  (4 downto 0)
             );
    end component;
    signal clk  : STD_LOGIC;
    signal led  : STD_LOGIC_VECTOR  (4 downto 0);
    signal sw   : STD_LOGIC_VECTOR  (4 downto 0);
begin

    dut: entity work.coffeemaker  port map (clk => clk, led => led, sw =>
sw);

    clk_process  :process
    begin
        clk  <= '0';
        wait for 5 ns;
        clk  <= '1';
        wait for 5 ns;
    end process;

    sim_process : process
    begin
        sw  <= "00000"; -- seq 1
        wait for 1 ms;  -- arbitrary wait.
        sw  <= "00001"; -- seq 2
        wait for 1 ms;
        sw  <= "00101"; -- seq 3
        wait for 1 ms;
        sw  <= "01001"; -- seq 4
        wait for 1 ms;
        sw  <= "01101"; -- seq 5
        wait for 1 ms;
        sw  <= "10001"; -- seq 6
        wait for 1 ms;
        sw  <= "10101"; -- seq 7
        wait for 1 ms;
        sw  <= "11001"; -- seq 8
        wait for 1 ms;
        sw  <= "11101"; -- seq 9
        wait for 1 ms;
        sw  <= "10100"; -- seq 10
        wait for 1 ms;
        sw  <= "10110"; -- seq 11
        wait for 1 ms;
        sw  <= "10111"; -- seq 12
        wait;
    end process;
end Behavioral ;
```

Define DUT and map testbench signals to DUT ports (names can be arbitrary, they don't have to match)

Using VHDL for Simulation (alongside description and synthesis)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164 .ALL;

entity coffeemaker_tb  is
--  Port  ( );
end coffeemaker_tb ;

architecture Behavioral  of coffeemaker_tb  is
    component coffeemaker_pwd
        Port  ( clk  : in   STD_LOGIC ;
                 led  : out STD_LOGIC_VECTOR  (4 downto  0);
                  sw   : in STD_LOGIC_VECTOR  (4 downto  0)
                  );
    end component;
    signal clk  : STD_LOGIC ;
    signal led  : STD_LOGIC_VECTOR  (4  downto  0);
    signal sw   : STD_LOGIC_VECTOR  (4  downto  0);
begin

    dut: entity work.coffeemaker  port map (clk => clk, led  => led, sw  =>
sw);

    clk_process  :process
    begin
        clk  <= '0';
        wait for  5 ns;
        clk  <= '1';
        wait for  5 ns;
    end process;
        sim_process : process
        begin
            sw  <= "00000"; -- seq 1
            wait for 1 ms;  -- arbitrary wait.
            sw  <= "00001"; -- seq 2
            wait for 1 ms;
            sw  <= "00101"; -- seq 3
            wait for 1 ms;
            sw  <= "01001"; -- seq 4
            wait for 1 ms;
            sw  <= "01101"; -- seq 5
            wait for 1 ms;
            sw  <= "10001"; -- seq 6
            wait for 1 ms;
            sw  <= "10101"; -- seq 7
            wait for 1 ms;
            sw  <= "11001"; -- seq 8
            wait for 1 ms;
            sw  <= "11101"; -- seq 9
            wait for 1 ms;
            sw  <= "10100"; -- seq 10
            wait for 1 ms;
            sw  <= "10110"; -- seq 11
            wait for 1 ms;
            sw  <= "10111"; -- seq 12
            wait;
        end process;
    end Behavioral ;
```

Define the clock signal as a sequential process (we can't use the clk on the XCD here, we're not on the FPGA!!)

Using VHDL for Simulation (alongside description and synthesis)

Define the stimuli (i.e., switch positions for seq1,2,...,12)

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity coffeemaker_tb is
-- Port ( );
end coffeemaker_tb;

architecture Behavioral of coffeemaker_tb is
    component coffeemaker_pwd
       Port ( clk : in  STD_LOGIC;
                 led  : out STD_LOGIC_VECTOR (4 downto 0);
                  sw   : in STD_LOGIC_VECTOR (4 downto 0)
              );
    end component;
    signal clk : STD_LOGIC;
    signal led : STD_LOGIC_VECTOR (4 downto 0);
    signal sw  : STD_LOGIC_VECTOR (4 downto 0);
begin

    dut: entity work.coffeemaker port map (clk => clk, led => led, sw => sw);

    clk_process :process
    begin
        clk <= '0';
        wait for 5 ns;
        clk <= '1';
        wait for 5 ns;
    end process;

    sim_process : process
    begin
        sw <= "00000"; -- seq 1
        wait for 1 ms; -- arbitrary wait.
        sw <= "00001"; -- seq 2
        wait for 1 ms;
        sw <= "00101"; -- seq 3
        wait for 1 ms;
        sw <= "01001"; -- seq 4
        wait for 1 ms;
        sw <= "01101"; -- seq 5
        wait for 1 ms;
        sw <= "10001"; -- seq 6
        wait for 1 ms;
        sw <= "10101"; -- seq 7
        wait for 1 ms;
        sw <= "11001"; -- seq 8
        wait for 1 ms;
        sw <= "11101"; -- seq 9
        wait for 1 ms;
        sw <= "10100"; -- seq 10
        wait for 1 ms;
        sw <= "10110"; -- seq 11
        wait for 1 ms;
        sw <= "10111"; -- seq 12
        wait;
    end process;
end Behavioral;
```

Using VHDL for Simulation (alongside description and synthesis)

- You can use practically any language to generate testbench stimuli like this, some industries in which FPGA implementations come at later stages of the project workflow (i.e., starting with software implementations) use C++ / C# / Python for compatibility with software tests.

- you just need to save the output waveform that you generate into a file that's readable by your simulator which will run the simulation on your VHDL-described circuit (the DUT).

- Also, VHDL is not the most prominent testbench language, people generally use SystemVerilog (SV) for that purpose these days. However running a VHDL DUT through an SV testbench is not straightforward in most simulators.

- Vivado does allow this by simply writing a Verilog wrapper around your VHDL DUT and running the SV testbench on it, but for our simple simulations VHDL will be more than enough

Outline

- Intro to simulation and verification in digital circuits
- Verification approaches: why is it a hard problem?
- Using VHDL for simulation
- Vivado's simulator and open-source options: GHDL + GTKWave

Outline

- Intro to simulation and verification in digital circuits
- Verification approaches: why is it a hard problem?
- Using VHDL for simulation
- Vivado’s simulator and free open-source options: GHDL + GTKWave

Vivado vs. FOSS options

- Free and open-source software (FOSS) tools for simulation are lighter and cheaper (free!) compared to Vivado, meaning you can run more tests in less time with less resources
- In terms of simulating logic behavior FOSS tools rarely make errors and when they do they typically have workarounds (ref)
- However, when you want to go beyond behavioral simulation and synthesize + implement designs on FPGAs, FOSS options start drying up.
- Currently the only FOSS-friendly path that I'm aware of is Lattice FPGAs (instead of Xilinx) with the Yosys toolkits, but those are also not "battle-tested" like Vivado and Quartus (Intel/Altera)
- For lightweight behavioral simulation on your VHDL designs and testbenches, you can try out **GHDL, which mimics Vivado's simulator + GTKWave to view waveforms.**
- Let me know if you want to try these out and I'll try to help you with the installations

Vivado vs. FOSS options

- For post-synthesis and post-implementation simulation our Xilinx FPGAs, there are no FOSS alternatives, Vivado is the only option.
- Once you add your VHDL testbench to your VHDL design project and successfully connect the DUT to the testbench, the vivado simulator is pretty straightforward to use.
- You just hit the “Run Simulation” button and choose what type of simulation you want to use
- I’m skipping the details of how these simulators work, but the Xilinx User Guides and application notes have a great level of detail about those aspects
- We will see how to use this tool in more detail in the next lab (FSM)

KOÇ UNIVERSITY

next → HW 2 + Lab 2 (FSMs)

🙋🏽‍♂️