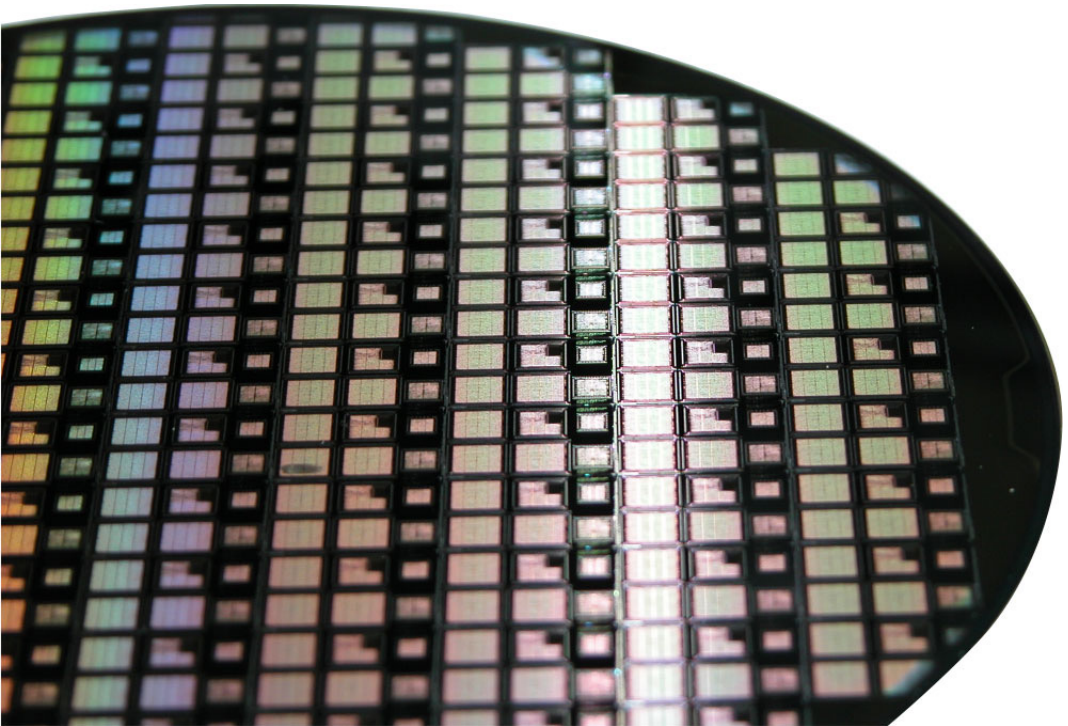
ETH zürich

Need for Verification

Adapted from VLSI1 lecture notes



20 March 2024

Luca Benini

Frank K. Gürkaynak

The story so far

- **We have learned basics of Verilog**
  - It is a way to describe circuit schematics
- **Modules / Hierarchy / Logic Signals**
  - We understand the basic constructs to define combinational circuits
- **We discussed process statements and sequential circuit description**
  - How to model basic sequential blocks (FFs), Finite State Machines
- **We should be able to describe all circuits in Verilog**
  - What we have covered so far is sufficient to design ANY circuit, even the most complex ones
  - You will need experience, and maybe learn a few more tricks, but nothing fundamental is missing.

What will we learn today

- **Understand strategies to find out functional bugs in our circuits**
  - What approaches can we try, how well do they work?
- **Functional simulations and simulation vectors**
  - How can we choose a (limited) set of vectors to uncover bugs in our circuit
- **Using testbenches to simulate our circuits**
  - What is a testbench, and how does it differ from our circuits
  - Understanding a coherent timing schedule for simulation
  - Different options we have for constructing testbenches

Verification can have different motivations

- **During specification:**
  Is the circuit I want to design, really what is needed?

- **During design:**
  Have I indeed designed a circuit that does what I specified?

- **During testing:**
  Once manufactured, can I tell working circuits from faulty ones
  This topic will be covered in part in VLSI 2 and exclusively in VLSI 4

Functional verification vs parametric verification

- **Functional verification is about what the circuit does**
  - Given a set of inputs, what is the expected behavior/output of the circuit
  - Described using algorithms, equations, stategraphs, truth tables, etc

- **Parametric verification is making sure physical properties are correct**
  - This relates to physically measurable qualities, speed, area, power, throughput..
  - Units like Mbits/s, mA, nW, pF...

- **This lecture is about functional verification**

We need specifications we can depend on

- **How do you know we are not missing anything?**
  - Are the specs complete, correct?
- **Are we describing functionality that is really needed?**
  - Was this what was wanted from us?
- **Do customers, marketing and engineers understand and agree on the same thing?**

- **Natural language and ad-hoc sketches do not work very well**
  - There are too many cases of 'famous' mistakes.

Ideally we would have formal specifications

- **All requirements would be described formally**
  - These could be equations, graphs, transfer functions, properties
- **We could then compare our designs mathematically and prove that they are correct.**
  - This is called **formal verification**
  - By far the most complete way of verification we can have
  - If we can specify the functionality properly, there are tools that can help us.
- **Unfortunately it is not always easy**
  - Usually only possible for smaller circuits
  - Mathematical formalisms are not really suitable for communicating with customers and management.

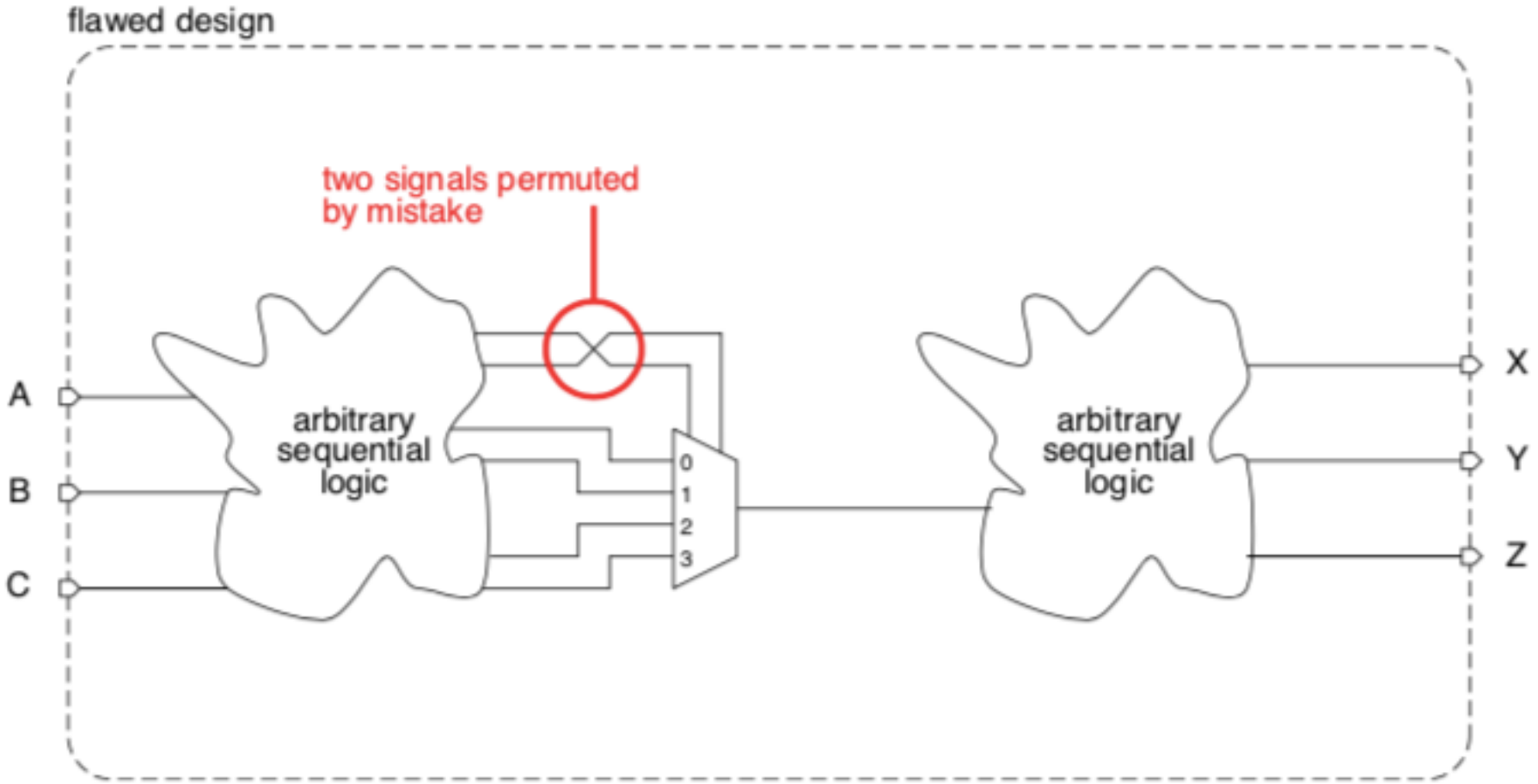
We still rely on a collection of ad-hoc specifications

- **Specifications are (generally) a text document that contains**
  - Descriptions in natural language
  - Truth tables
  - Block diagrams
  - Graphs and tables
  - References to known and accepted standards (we will have a USB3 interface)
- **Formal methods still too rigid/cumbersome for most practical circuits**
  - Usually languages that can capture behavior
- **Important unsolved problem in design**
  - Although there are many mistakes, we still believe we can somehow manage

Now that we have (some) specifications, does it work?

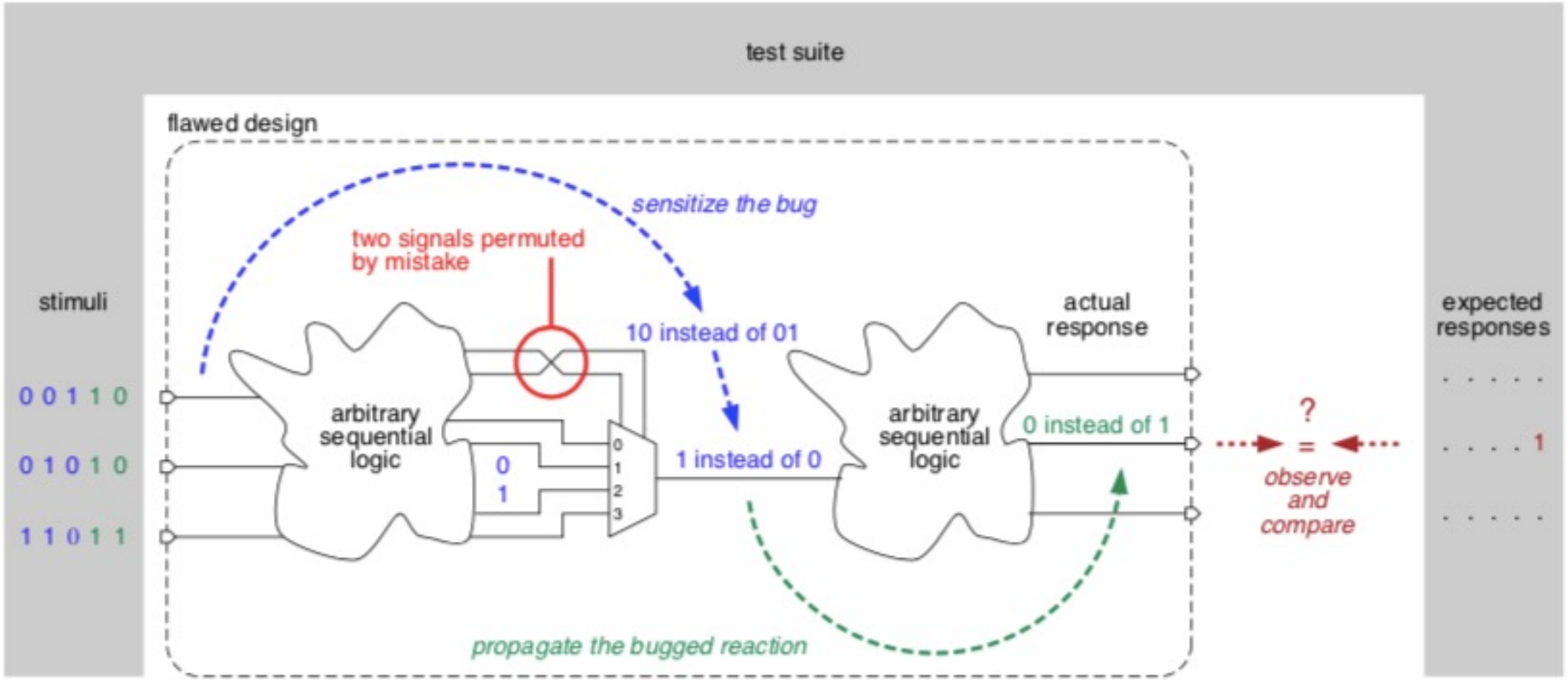
- **We use a formal method to mathematically prove that it is correct**
  - Formal specifications not easy to define
- **Directly build the circuit and use it in the field to see what happens**
  - It may be costly to build the circuit, and it may take a very long time.
- **We can make a prototype of the real circuit and use it**
  - Rapid prototyping solutions (usually on FPGAs) allow you to run the complete system relatively quickly. System will be (much) slower, but possible to get behavior in field
- **We can model the circuit in a simulator and see how it behaves**
  - No need to build a real circuit, can experiment quickly, but simulation speed much much slower than the real circuit (or a prototype)

How do we detect a fault using simulation?



Figures taken from Hubert Kaeslin, "Top Down Digital VLSI Design: from Architectures to Gate-Level Circuits and FPGAs"

Sensitize/activate -> propagate -> observe/compare



Figures taken from Hubert Kaeslin, "Top Down Digital VLSI Design: from Architectures to Gate-Level Circuits and FPGAs"

Basic steps to uncover a bug

- **Activate/sensitize**
  - Set of inputs that will trigger a situation in which circuit will misbehave is needed
- **Propagate**
  - Make sure the faulty behavior reaches an observable point
  - Real circuits have only limited number of I/Os and are more limited
  - Simulators will be able to allow you to peek into the circuit
- **Observe/compare**
  - The activated fault must generate a different output than an expected response
- **Need simulation vectors (input) and expected responses (outputs)**

Let's not inspect and compare outputs manually

- **The temptation might be high to manually inspect waveforms**

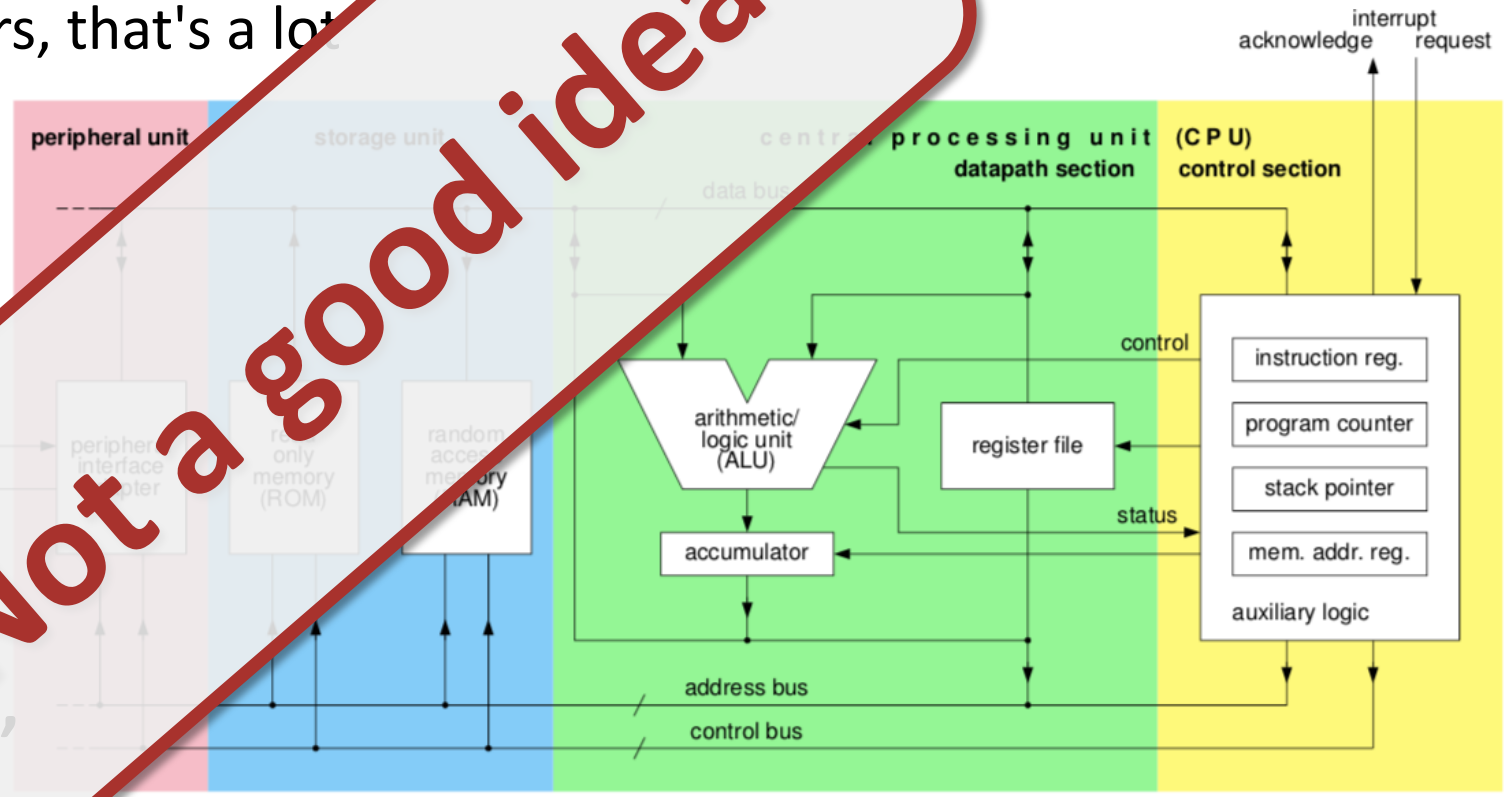
DON'T

- **You could only manage this for very small circuits**
  - Practically anything worth designing will be too complex to do this
- **Humans make way too many mistakes**
  - Why were there bugs to start with?
- **Automated methods to check outputs is a must!**

Exhaustive methods, check every possible input

- **For every state a circuit can be you need to test every possible input**
- **If you have N inputs and S state bits**
  - You need $2^N \times 2^S = 2^{N+S}$ vectors, that's a lot
- **Intel 8080 from 1974**
  - 8 data, 4 control inputs
  - 8x 8-bit registers
  - 2x 16-bit registers
  - 5 flag bits
  - $2^{12} \times 2^{101} = 2^{113} \approx 10^{34}$
- **Even if you run at 100MHz, billions x billions of years**

Not a good idea !



Figures taken from Hubert Kaeslin, "Top Down Digital VLSI Design: from Architectures to Gate-Level Circuits and FPGAs"

So we can not test for every input, how do we choose?

- **We can only afford to test for a tiny fraction of possible inputs**
  - Somehow we need to select a suitable set of input vectors that gives us sufficient coverage of the functionality
- **This is called directed testing**
  - Vectors are chosen according to ***some*** method
  - There are ***some*** methods to help us determine if these vectors are sufficient
  - More vectors added if needed
- **How to determine the best subset of all possible inputs is a fundamental and open problem of verification**

Some methods to detect if we have enough vectors

- **Code coverage: check if all lines of code have been simulated**
  - Need support from simulator
  - Important to check all conditions in statements
  - Can be used to guide generation of vectors to increase coverage
- **Monitoring toggle counts of all nodes in circuit**
  - See if everything has toggled at least once
  - Can identify regions that are not sufficiently tested
  - Works better with gate-level netlists rather than RTL description

So we have some inputs, how do we get correct outputs

- **We need to check if the outputs of the circuit matches expectations**
  - We call these vectors **expected responses**
- **How do we get these expected responses?**
  - We need a trusted model of the circuit we want to build. This is a **golden model**.
- **If we already have a golden model why do we have a problem?**
  - Golden model could be written anything:
    - Matlab, Python. C, Java..
  - Not every code can be directly mapped to HDL/hardware
- **Verification identifies mismatch between golden model and circuit**
  - (In the beginning) you find just as many errors in the golden model as in the circuit
  - Model stays stable while you are improving your circuit/architecture

More about the golden model

- **The circuit specifications are our starting point**
  - What do we want the circuit to do
- **For well defined mathematical functions, easy to do**
  - There might already be an implementation in a computer language, Matlab
- **Standard interfaces (USB, AXI, SPI..) have an agreed upon behavior**
  - There are pre-defined models for such interfaces that can be used
- **Otherwise a model has to be first constructed**
  - Using a convenient language
- **Sometimes we need to adapt the model to a bit-true model**
  - In hardware we deal with ones and zeroes. Some high-level models use computer language specific data types (int, float..) these may need conversions

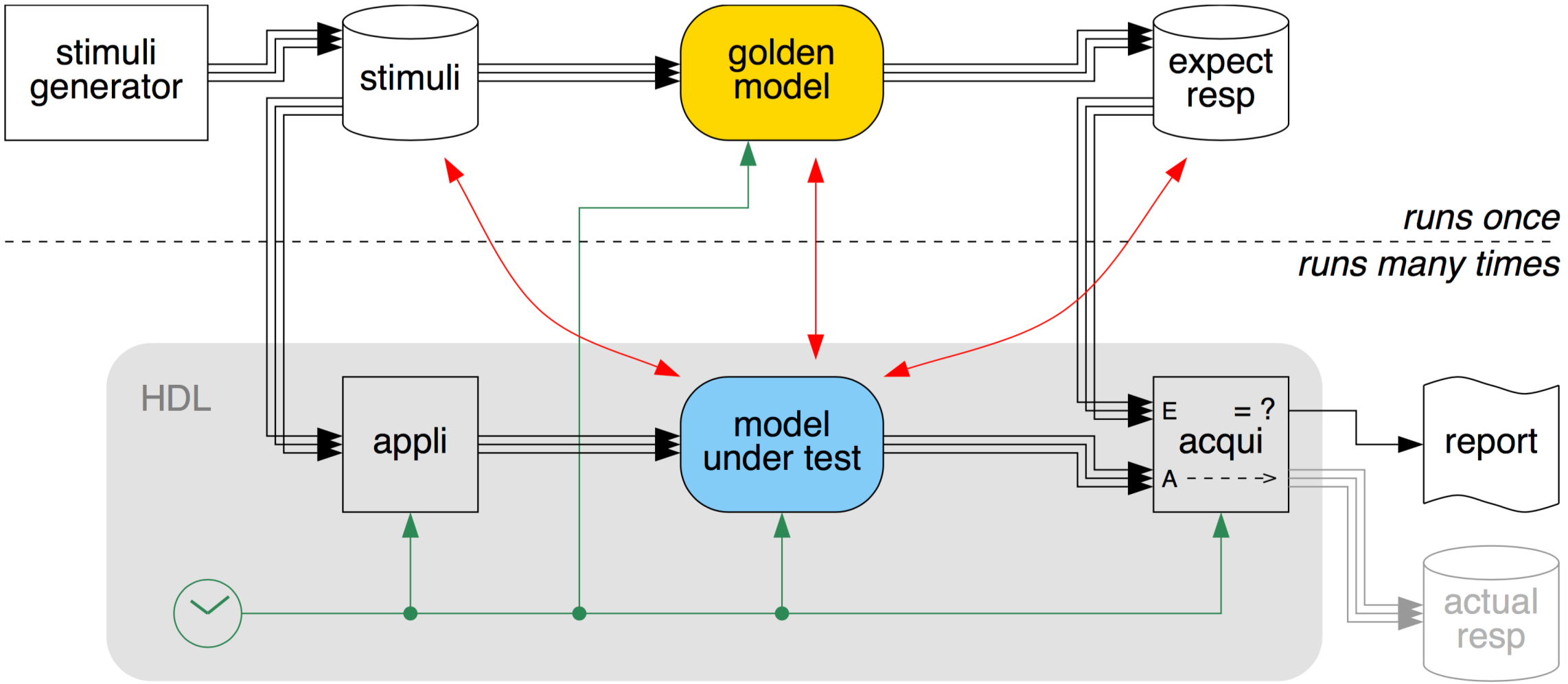
Wish list for simulation

- **We need a notion of time**
  - So far we modelled behavior, we did not care about what happens when
- **We need stimulus (inputs) for our circuit**
  - Generate the inputs, read them from a file, get them from a running program
- **We have to collect the actual outputs of our circuit**
  - Record what the circuit has generated so that we can compare later on
- **Compare the outputs to expected responses**
  - We have to know what the circuit was supposed to generate
  - A report at the end of the simulation is essential, we will not check manually
    - How many vectors were tested
    - What was the result, were there errors, when

Testbench is a virtual environment to test circuits

- **It is (mostly) written using HDL, we will see next lectures**
  - Special constructs in Verilog will be needed for the functionality
- **Allows the circuit/module under test to be instantiated**
  - Your circuit is instantiated inside the testbench.
- **Testbench is self sufficient:**
  - Normally there are no inputs/outputs to the testbench
  - Provides clock, reset and other control signals
  - Supplies the inputs
  - Collects outputs and compares the results
- **Some of the testbench functionality can be provided by other tools**
  - There is a lot of variation for this

A typical testbench for a digital circuit



Figures taken from Hubert Kaeslin, "Top Down Digital VLSI Design: from Architectures to Gate-Level Circuits and FPGAs"

Not everything in a simulation has to be in HDL

- **Modern simulators (i.e. Mentor Questasim) provide a rich environment**
  - Offer basic programming over a TCL interface
  - Can instantiate HDL circuits
  - Generate stimuli: Read in files, connect to running programs
  - Record outputs: dump waveforms and/or lists to files
- **Pre/post-processing using your favorite program/language**
  - Perl, Python, Matlab.. can be used to generate stimuli and/or reports
- **Our job: find an efficient combination**
  - There is no universal answer, different applications, different solutions
  - Personal preferences (and beliefs) play a big role
  - Some of it is '*altlast*' (legacy): The solution we use in exercises has a long history

There are many alternative solutions

Figures taken from Hubert Kaeslin, "Top Down Digital VLSI Design: from Architectures to Gate-Level Circuits and FPGAs"

We spend a lot of time on verification

- **The engineering effort of verification is quoted to be 50%-95%**

  "Verification is actually 100% of our work, as in modern SoC design we do not design anything new, we just combine pre-designed blocks and try to make sure that they work!"
  (In a discussion in 2019 Week of Open Source HW)

- **You can debate about exact effort but**
  - It is the most time consuming part of the design process
  - For the past 50 years, no known guaranteed method to solve it
  - Any optimization of verification effort has huge returns

This is not religion, choose the method that works best

- **Ensuring there are no bugs is the most important thing…**
- **.. but efficiency is a close second in verification**
- **Relying on too many specialized tools/languages restricts the portability of your verification environment.**
  - All HDL verification **NEEDS** to support HDL. So testbenches that only rely on HDL are more portable
- **Familiarity with an environment should not be deciding factor**
  - Do not just say: *"I think ABC is difficult but I know XYZ pretty well"*
  - You will be using/working with this for a long time, try to learn how to do it more efficiently

Some last words on verification

- **Verification is a big part of the design process**
  - There is still no solution to address this problem directly, we need a combination of methods
- **Separate design and verification teams**
  - Having a bit of distance from the design helps finding issues better
- **Collect test cases from multiple sources**
  - You can not test all possible inputs, we need to find good enough vectors, you need every bit of help you can get. Make use of diverse set of tests