ESD Electronic Systems Design

Synthesis Verification Testing Power Communication

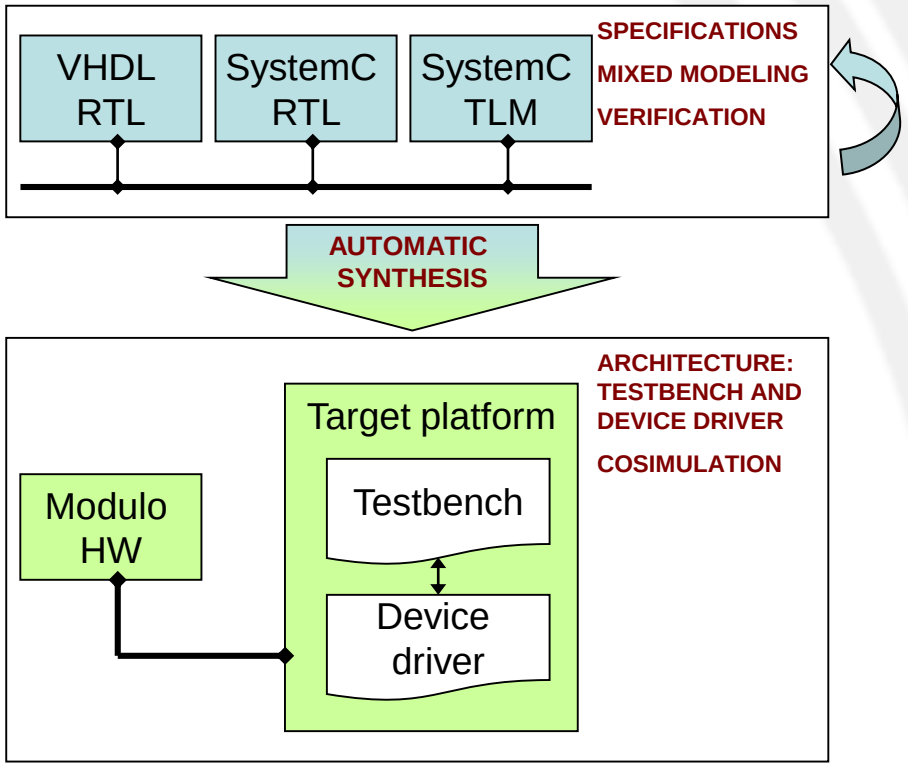
Introduction to Verification: Assertions

Stefano Centomo

Course introduction: Lab topics

- **Emulate the design flow of a real embedded/cyber-physical system**

- **Specification**
  - Compilation/execution/debugging of SystemC code
  - Modeling of SystemC at RTL level
  - Modeling of SystemC at TLM level
  - Time evolution in SystemC
  - Modeling analog and continuous behaviors: SystemC-AMS
  - Components integration
  - Mixing SystemC RTL, TLM and AMS
  - Model Verification



- **SW Synthesis**
  - Platforms, testbenches and drivers
  - Model Based Design

- **HW Synthesis**
  - VHDL modeling at RTL
  - Automatic Synthesis from TLM descriptions
  - Automatic Synthesis of RTL VHDL code

Verification

- Needed to prove that the initial specifications and expectations of the designers are implemented in the final design

- Formal verification is complex and very expensive in terms of time
  - Exhaustive
  - Model Checking
    - Based on state reachability: State Explosion Problem
  - Satisfability Modulo Theory
    - Automated Reasoning: based on SAT (NP-Complete)

- Dynamic verification based on simulation has been introduced in design flows
  - Simulation-based: not exhaustive
  - Good performance simulation implies good verification performace

Assertions

- Used to describe expected or unexpected conditions in the device under test
  - If an assertion fails, then an improper operation or condition has been detected
  - Assertions describe what should happen with no detail about the effective implementation
- Capture wrong or unexpected behaviours really closed to the source of the problem
  - Allow to see the effects of inputs and errors in the design (osservability)
- Do not increase controllability
  - Ability to stimulate the design functionalities by controlling its inputs
  - Assertions themselves must be stimulated by the verification environment

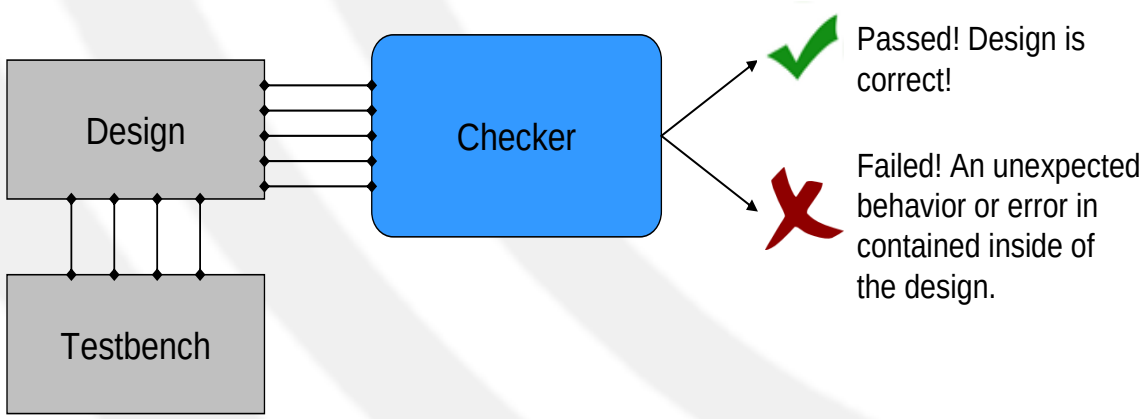
Assertions (Cont.d)

- Assertions express properties of two main types:
  - Liveness: "Sooner or later, something good will happen"
    - Termination, service or starvation guarantees.
      - E.g. if you execute P, sooner or later it will produce an output
  - Safety: "Nothing bad will ever happen"
    - No bad condition must ever occur
    - Deadlocks, mutual exclusions, scheduler characteristics, etc.
      - E.g. no output must be returned until an input is given to the device.

- Different specification methods
  - Informal
    - Based on natural languages
    - Early stages of design
  - Formal
    - Based on formal frameworks
      - Temporal Logics (LTL, CTL, STL etc…)
    - Design Automation for Embedded Systems Course (SSE) next year!

Checkers

- Modules attached to the design under verification
  - Formal assertion specification allows for automatic generation
- Allowed to access both its internal and external variables, signals and ports
- Follow the module execution and that check whether a certain property is satisfied or not
  - If a property is not satisfied, an error message is shown to the user and simulation might be stopped.



SystemC into action

- Uncompress the archive
  - $ tar xzvf 06_transactors_and_assertions.tar.gz
  - $ cd 06_transactors_and_assertions/
  - $ ls
  - root_assertions root_transactors AMS_transactors
  - $ cd root_assertions
  - $ ls
    - bin inc Makefile obj src
- Compile and execute
  - $ cd root_assertions
  - $ make
  - $ ./bin/root_RTL.x

SystemC into action (cont.d)

- If number_isready is 1, then in less than 100 clock cycles result_isready goes to 1
  - Formally (Linear Temporal Logic)

$$(number\_isready = 1) \rightarrow \bigvee_{i=1}^{100} X^i(result\_isready = 1)$$

If number_isready == 1 then result_isready == 1 in less then 100 clock cycles

```
while(true){
  if (number_isready.read() == 1) {
    while((count < 100)&&(!true_property)) {
      wait();
      if (result_isready.read() == 1)
          true_property = true;
    }
    if (true_property) cout<<"true"<<endl;
    else cout<<"false"<<endl;
  }
  wait();
}
```

SystemC into action (cont.d)

- If number_isready is 0, then result_port is 0 and result_isready is 0
  - Formally
    $(number\_isready = 0) \rightarrow (result\_port = 0 \ \wedge resulto\_isready = 0)$

If number_isready == 0
then result_isready == 0
and result_port ==0

```
while(true){
  if (number_isready.read() == 0) {
    if ((result_isready.read() == 0)&&
        (result_port == 0))
          true_property = true;
    }
    if (true_property) cout<<"true"<<endl;
    else cout<<"false"<<endl;
  }
  wait();
}
```

SystemC into action (cont.d)

- If counter is less than 16, then the STATUS is ST4
  - Formally:

$$counter < 16 \rightarrow STATUS = ST4$$

  - This property is false!

If Counter < 16 then
STATUS = ST_4

```
if (Counter.read() < 16)
  if (STATUS.read() == ST_4)
    cout<<"true"<<endl;
  else cout<<"false"<<endl;
```

SystemC into action (cont.d)

- Check results and which properties are satisfied by the system

	Satisfied?
Property 1	
Property 2	
Property 4	

- Then, comment lines 38 to 63 in testbench.cpp file and uncomment lines 65 to 88

	Satisfied?
Property 1	
Property 2	
Property 4	

SystemC into action (cont.d)

- Implement property number 3:
  - If the number is not zero, the square root is not zero).
  - Formally:

$$(dout > 0) \rightarrow X^{latency}(dout\_rdy = 1 \wedge dout\ != 0)$$

    - The latency is the number of ticks (clock cycles in our case) elapsing from the input to the corresponding output

Final Report (Bonus)

- To prove your knowledge of properties, define three properties of your Floating point Multiplier design and write the checkers that implements them
  - The first two properties must be always true
    - Explain why these properties should be satisfied and show that the execution of your system reports no error
  - The third property must not be satisfied by the design
    - Show that the checker detects this condition and that an error message is written on standard output for the user during simulation
- No properties for the AMS part is required
  - Assertion-based verification of continuous and mixed systems is a current research field, not the state of the practice!
- You can test them plugged to the platform rather than to the single device
  - Why is this useful?