

TheSyDeKick

System development and verification framework in Python

Department of Electronics and Nanoengineering
Aalto University, School of Electrical Engineering

November 3, 2021

Outline

TheSyDeKick - What it is

What is it?

Simulator interfaces

Analog simulator interface

Digital simulator interface

Design Examples

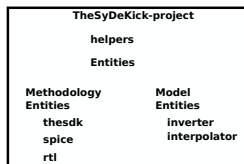
Example 2: Inverter Chain

Example 3: ADC Model

Conclusion

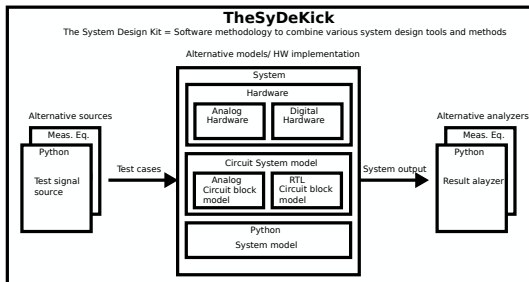
TheSyDeKick - What it is

What is it?



- ▶ TheSyDeKick is a well structured container for:
 - ▶ Python packages (Entities), that provide *formalism and methods to run simulations in python, and with external simulators*
 - ▶ Python packages (Entities), that provide formalism to create and use *IO compatible models*
- ▶ TheSyDeKick is a cumulative collection of Python classes and methods that help the designer to carry out the most common tasks encountered in microelectronics design. Most likely the *simulation task you have in mind has some supporting methodology available in TheSyDeKick*
- ▶ TheSyDeKick is *nothing more*

What is it?



- ▶ TheSyDeKick fromalims helps the designed to:
 - ▶ Write and simulate modular, hierarchical, IO compatible hardware models wit various levels of abstraction (python, spice/rtl)
 - ▶ Control the abstraction level with one parameter on chosen level of hierarchy.

What is it?

- ▶ Minimum set of structural constraints,
 - ▶ IO definitions
 - ▶ Blocks described as connected *Entities*.
- ▶ Aims to automation of *repetitive, well structured tasks*
 - ▶ Model and modeling environment structure with init scripts
 - ▶ Running simulations
 - ▶ Defining simulator calls
 - ▶ Documentation of the design with Docstrings
- ▶ Currently supports Python, Verilog, VHDL and three spice variants.
- ▶ Under construction: Measurement equipment.
- ▶ <https://github.com/TheSystemDevelopmentKit>

TheSyDeKick project file structure

TheSyDeKick project structure

```
thesydekick_project/  
├── Entities/  
│   ├── thesdk/  
│   ├── rtl/  
│   ├── spice/  
│   └── amplifier/  
│       ├── amplifier/  
│       │   └── __init__.py  
│       ├── spice/  
│       ├── Simulations/  
│       └── doc/
```

Simulator interfaces

Analog simulator interface

- ▶ Calling analog simulators is handled through a common interface, the **spice** module
- ▶ Ultimate goal of **spice**: support for most industry standard simulators
 - ▶ General (w.r.t to simulator), reusable simulation testbenches
 - ▶ Centralized post processing based on open source tools and libraries.
- ▶ Save time in re-writing testbenches for different simulations and centralize effort to all-in-one testbench
 - ▶ Currently support for DC, AC, and Transient analysis. Easy to add other analysis types as IO formats are the same.
- ▶ <https://github.com/TheSystemDevelopmentKit/spice>

Digital simulator interface

- ▶ Calling digital simulators is handled the **rtl** module
 - ▶ Most commonly used simulation testbenches automatically generated for both Verilog and VHDL design under tests.
 - ▶ Automated file IO generation for strings (bit vectors), integers, and complex numbers.
 - ▶ Support for any signal type through customizable format parameter.
- ▶ <https://github.com/TheSystemDevelopmentKit/rtl>

TheSyDeKick simulation procedure

1 Write the testbench

- 1 Specify analysis type, inputs, outputs, supplies etc. based on TB properties
- 2 Configure simulator options, corners, etc. based on testbench properties
- 3 Specify desired outputs, e.g. FFT of waveform, transient waveform
- 4 Specify run cases, e.g. transient analysis for 2 different netlists, etc.

2 Run the simulation

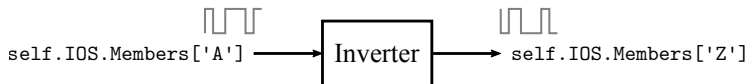
3 Analyze results

4 (Optional) Repeat simulation for different sets of parameters

- ▶ Once you have one testbench, reuse for similar applications makes things faster
- ▶ Results saved (if user chooses so), no need to repeat lengthy simulations to fix errors in post processing

Design Examples

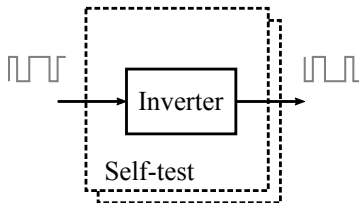
Example 1: Inverter



- ▶ Functional model: Inverts input signal
- ▶ IOS is a Bundle-type container for storing IO data
- ▶ Can have as many members as needed

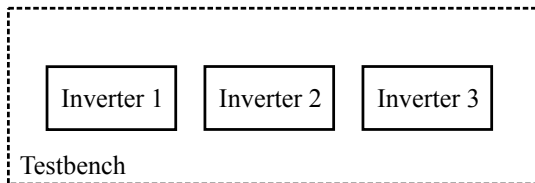
Example 1: Inverter

Python / VHDL / SystemVerilog / Eldo



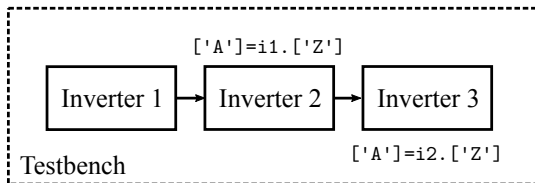
- ▶ Multiple representations of the same entity
- ▶ Self-test is a way to verify the functionality of entity “in vacuum”
- ▶ Self-test generates and feeds the identical input data to all simulators

Example 2: Inverter Chain




- ▶ A separate **Testbench** entity is instantiated
- ▶ Chain of inverter instance is created with a for loop

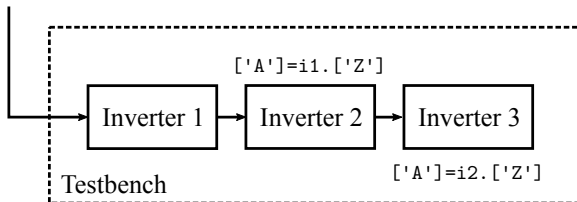
Example 2: Inverter Chain



- ▶ A separate **Testbench** entity is instantiated
- ▶ Chain of inverter instance is created with a for loop
- ▶ Data flow defined by IOS connections

Example 2: Inverter Chain

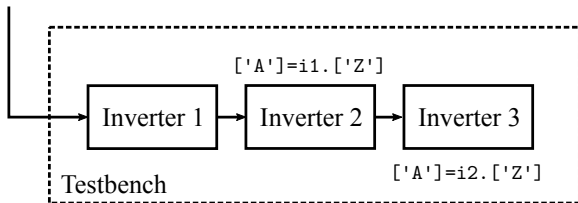
i1.['A'].Data= 



- ▶ A separate **Testbench** entity is instantiated
- ▶ Chain of inverter instance is created with a for loop
- ▶ Data flow defined by IOS connections
- ▶ Testbench input is assigned as input of 1st element

Example 2: Inverter Chain


```
i1.['A'].Data= [1,0,1,0]
```

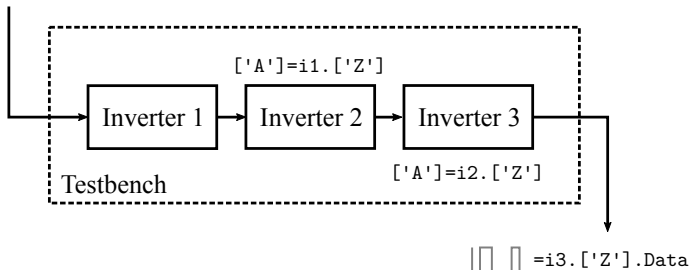


```
i1.Run() -> i2.Run() -> i3.Run()
```

- ▶ A separate **Testbench** entity is instantiated
- ▶ Chain of inverter instance is created with a for loop
- ▶ Data flow defined by IOS connections
- ▶ Testbench input is assigned as input of 1st element
- ▶ Signal is propagated through the chain by running the entities in sequence

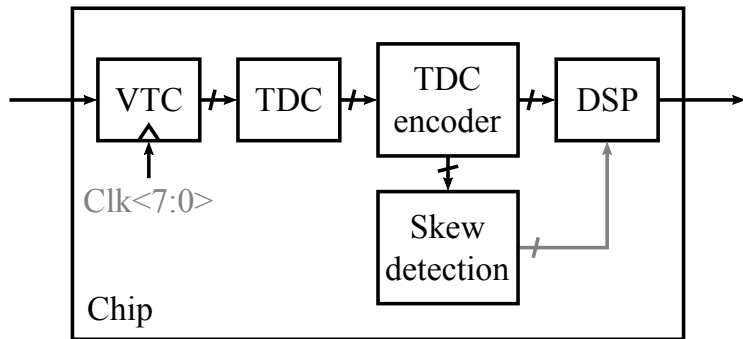
Example 2: Inverter Chain

i1.['A'].Data= 

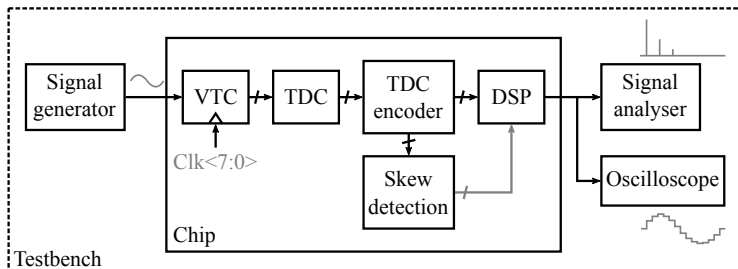


- In the end, the output of the last element contains processed data

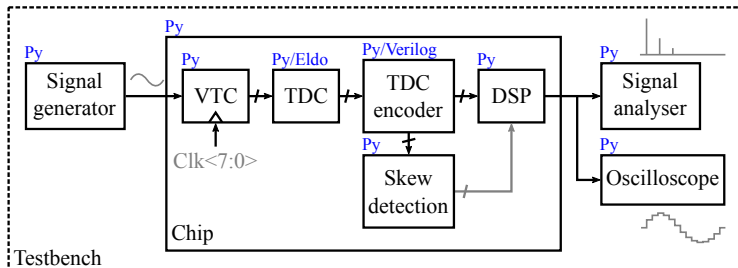
Example 3: ADC Model



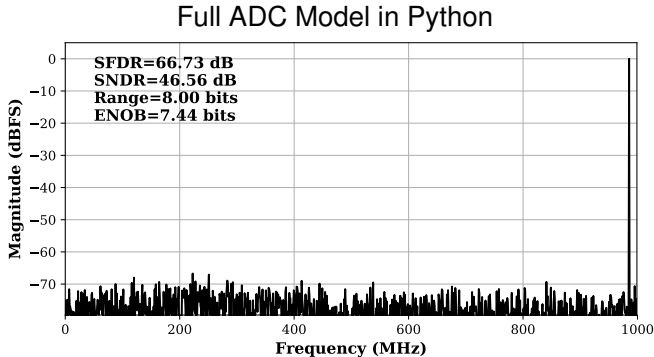
Example 3: ADC Model



Example 3: ADC Model



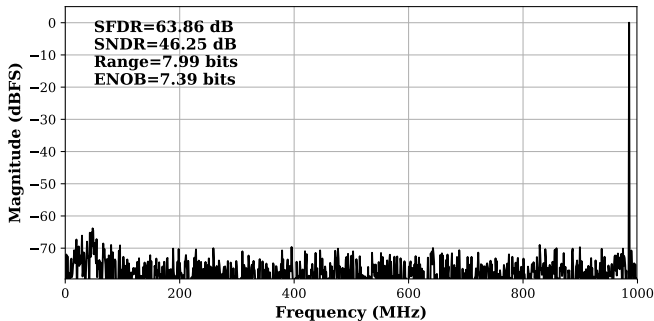
Example 3: ADC Model



- Spectrum plotted by `signal_analyser` -module, which also calculates SNDR etc.

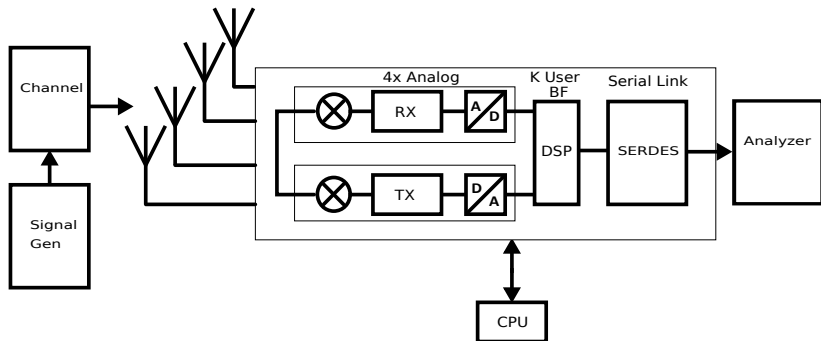
Example 3: ADC Model

Full ADC Model in Python with Transistor-level TDC Model



- ▶ Same exact simulation as before, except the TDC block is simulated as a spice-netlist in Eldo

Example 4: Multi-user MIMO receiver



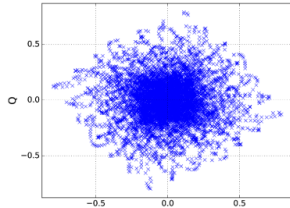
► Goals:

- First, model the receiver of a single chip with python
- Python models swappable to RTL and analog models
- Generated sub-blocks verified at the system level
- Control the HW generators of the block (emerging feature)

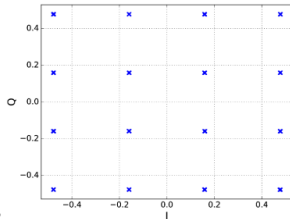
Channel 802.11n C



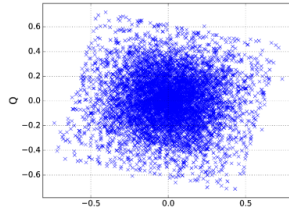
DSP, Ant=0, Usr=0, py, EVM=0.19 dB, BER= 0.364



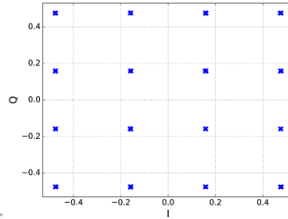
Postproc, Usr=0, py, EVM=-55.09 dB, BER= 0



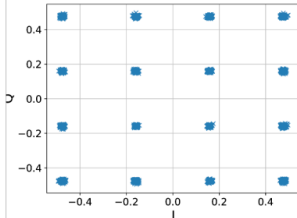
DSP, Ant=0, Usr=1, py, EVM=2.09 dB, BER= 0.453



Postproc, Usr=1, py, EVM=-52.50 dB, BER= 0

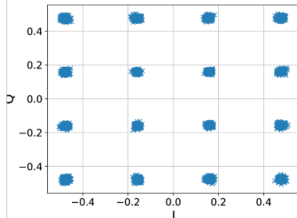


Postproc, U_{sr}=0, py, EVM=-40.96 dB, BER= 0



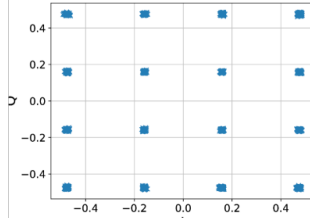
DSP Decimator in python

Postproc, U_{sr}=0, py, EVM=-34.47 dB, BER= 0



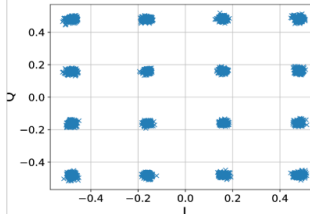
DSP Decimator in verilog

Postproc, U_{sr}=1, py, EVM=-41.39 dB, BER= 0



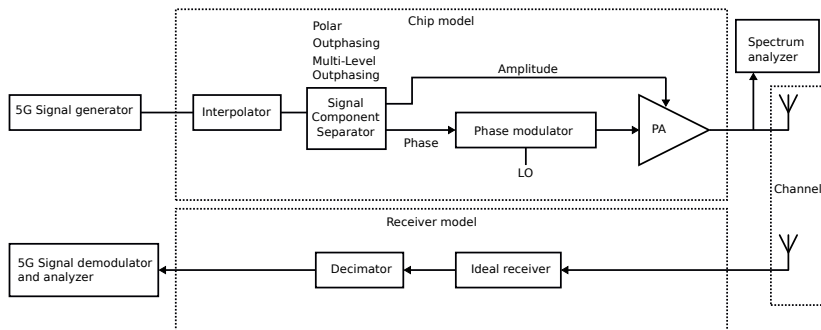
DSP Decimator in python

Postproc, U_{sr}=1, py, EVM=-33.41 dB, BER= 0



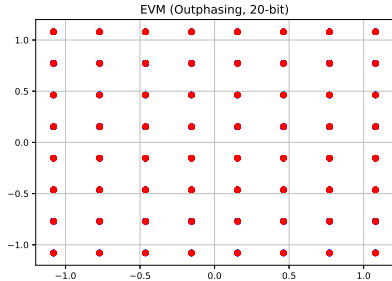
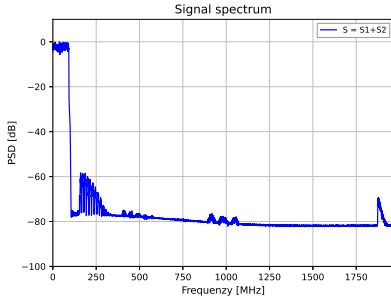
DSP Decimator in verilog

Example 5: Outphasing transmitter



- ▶ System development
 - ▶ Model the transmitter chip with python
 - ▶ Model the ideal receiver in python
 - ▶ Simulate the functionality

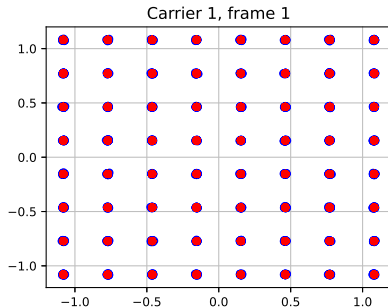
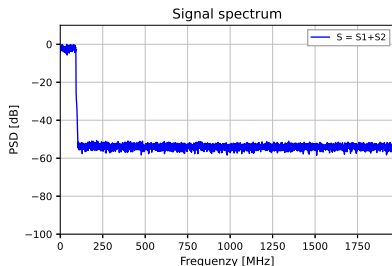
Example 5: Outphasing transmitter



► EVM 0.34

Hattrick: Outphasing transmitter

- ▶ System verification
 - ▶ Replace the transmitter DSP with RTL models
 - ▶ Verify functionality



- ▶ EVM=0.60

Conclusion

- ▶ Modular design environment through well defined IO boundaries and *Entity* definitions.
- ▶ Open Source
- ▶ Automates repetitive verification tasks and provides means for programmatic verification with various simulators
- ▶ Support for measurement equipment under development.
- ▶ *Enables* co-development by multiple designers using various verification tools.
- ▶ *Enabled by* utilization of programming methodology and version control in hardware design context.