Simulation & Testbenching

98-154/18-224/18-624: Intro to Open-Source Chip Design

Verilog vs SystemVerilog

Verilog, SystemVerilog, and VHDL

- Verilog: Originally developed in 1984 for simulation and verification. Eventually co-opted for design
- SystemVerilog: An extension of Verilog, from 2009, with nicer features added for both design and verification
- VHDL: Similar to Verilog but focused on being more "strongly-typed" (fills a weird niche)
- Choice between (System)Verilog vs VHDL often depends on location, industry, and historical choices rather than language itself

Evolution of Verilog

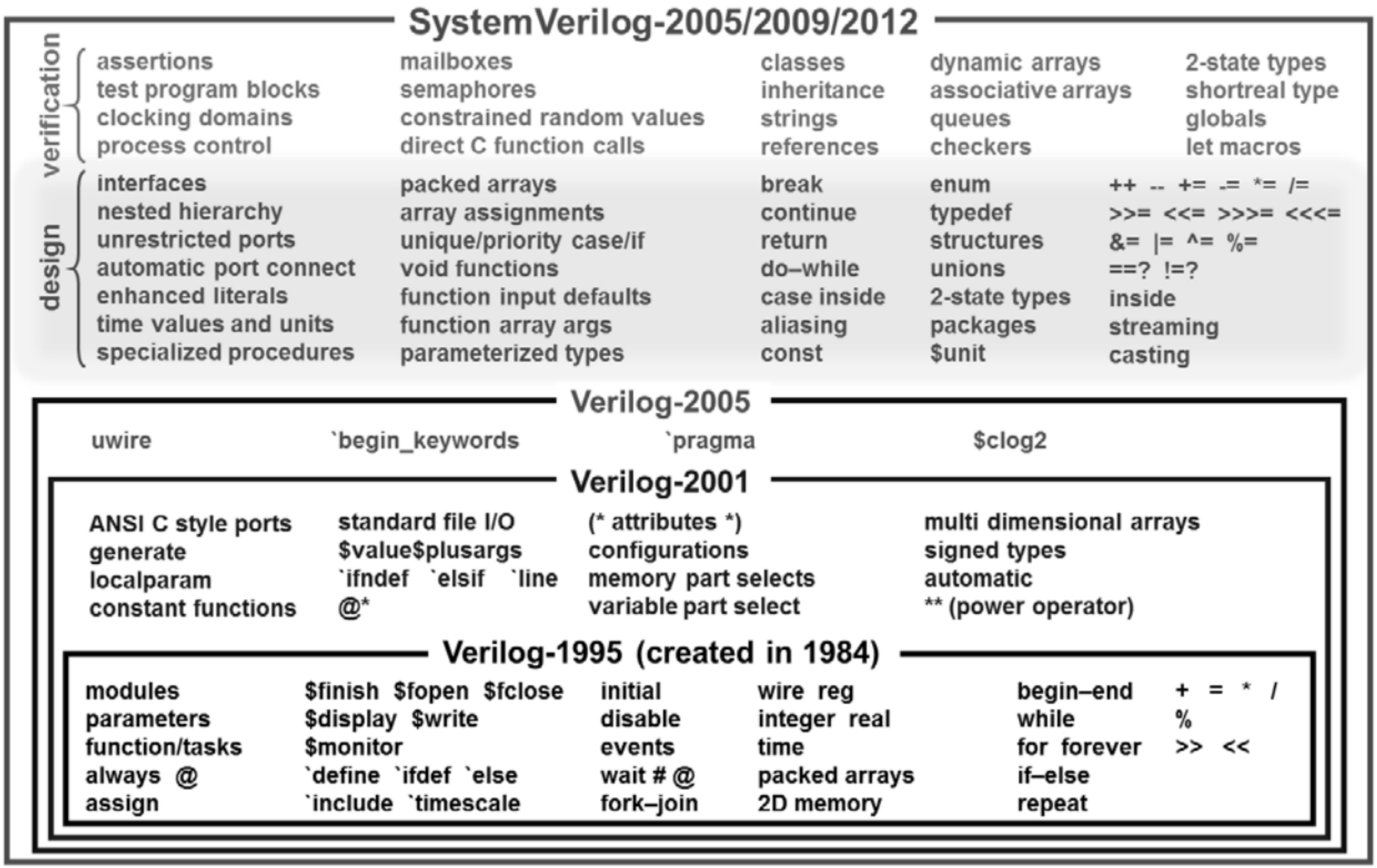


Image credit: Sutherland et al.

Verilog vs SystemVerilog: Synthesis

SystemVerilog features that are missing in Verilog-2005:

- `logic` datatype (instead of `reg` and `wire`)
- Specialized `always` blocks
- Array literals, arrays as module ports, C-style arrays, etc.
- Custom types: `typedef`, `enum`, `struct`, `union`
- Advanced case statements (`unique`, `priority`, etc.)
- Really good 2013 paper that goes into more depth

Verilog vs SystemVerilog: Testbenching

SystemVerilog features that are missing in Verilog-2005:

- Object-oriented (classes, strings, inheritance)
- Advanced assertions
- Dynamic & associative arrays
- Concurrency tools: mailboxes, semaphores, process control
- Queues, checkers, constrained random-values
- …and a lot more

A note on `wire`, `reg`, and `logic`

- `wire`: Is a “net” type, can be used for `assign` statements and module ports. In hardware, always generates a wire
- `reg`: Is a 4-state (0, 1, Z, X) “variable” type, can be used in `always` blocks. Can generate a wire, register, or latch depending on how it is used
- `logic`: Infers either `wire` or `reg` based on context. Recommended to use this exclusively

Other built-in types

- `bit`: Like `reg`, but only 2-state (0, 1). Sometimes handy for testbenches; not recommended to use in synthesis because it may cause synthesis-simulation mismatch
- `integer`: Alias for 32-bit `reg` type
- `byte`/`shortint`/`int`/`longint`: Aliases for 8/16/32/64-bit `bit` type

Sad State of Affairs

- Serious lack of open-source SystemVerilog-supported tools (partly because of complications in the standard)
  - Vendors will often do whatever they feel like, instead of following a legitimate standard
- Very limited support for testbenching-specific SystemVerilog features in open-source simulators
- All hope is not lost, however!

Our Savior `sv2v`

- Originated out of a CMU research project (Prof. Ken Mai and Prof. Dave Eckhardt) in 2019, still actively maintained
- Transpiler from SystemVerilog to Verilog-2005, was originally built as part of a “virtual FPGA” project
- Primarily designed for synthesis, although supports a surprisingly good set of simulation features as well
- The test suite shows most of the available features

Why do we need to simulate our hardware anyways?

Why Simulation

- Full introspection into logic (waveforms, etc.); can't probe every single wire in hardware
- Unit-testing (test components individually before doing full-system integration); can help find subtle bugs early
- Develop and test modules without access to FPGA
- Often faster iteration cycle (~minutes instead of ~hours, no need to wait for slow synthesis software)
- Start developing firmware/drivers using the simulation, even before hardware is fully built

Simulation Methodology

Example Design

Going to use an example design to demonstrate the upcoming verification techniques:

- Matrix-vector multiplication accelerator
- Based on a simple FSM and counter design
- Note the use of `$clog2`

Demo: Example Design & `sv2v`

Pure-Verilog Simulation

- Couple of options for pure-Verilog simulation
  - Icarus Verilog (mostly Verilog-2005), can use with `sv2v` but not ideal
  - Verilator (as of very recently) supports SystemVerilog testbenching as an experimental feature
- Alternatively, write design in SystemVerilog and testbench it in another language
  - Some advantages, some disadvantages

Event-Based Simulation in Verilator

- Verilator recently added capabilities for dynamic scheduling
  - Allows you to write testbenches in Verilog instead of C++
- We won’t go too deep into this (focusing on software-hardware co-simulation)
  - AntMicro blog post
  - AntMicro post on UVM
  - AntMicro examples

Proprietary Tools

- Proprietary (non-open-source) simulators include ModelSim, Questa, VCS, Xcelium, Riviera Pro
- Subtle differences in the SystemVerilog implementation between different tools
- You'll probably see them used in the industry
- EDA Playground gives you free access to various proprietary tools with a student email

Demo: Icarus Verilog

VCD Files & GTKWave

- VCD (Value Change Dump) files, can use to export waveforms generated by a simulation
  - Other formats like FSDB also exist, but VCD is simple and universal
- Pros: “Do one thing and do it well” (can have a separate tool for simulation vs viewing)
- Cons: Struggles with overly-complicated waveforms, less tightly-integrated with source-code and coverage

Demo: GTKWave

Demo: VGA Decoder

vcdvcd library for Python

Software-Based Simulation

- When writing a testbench, you're really just writing sequential code
  - "Give this input to the module, wait for time, read the output, check if the output is equal to the expected"
- Can write this in a real software language focused on productivity, like Python or C++?
- Allows you to use software libraries and models much more easily

Software-Based Simulation: Pros & Cons

- Pro: Often easiest to develop/model algorithms in software
- Pro: Hardware-software co-simulation
- Pro: Reusable and modular, can build simulation models for peripherals (uart_sim)
- Con: Not always intuitive; have to manually handle clocking, stepping, and propagation
- Con: Edge-cases can cause friction at the boundary between languages

Hardware-Software Co-Sim: Why?

- Very rarely do you just have a standalone piece of hardware; usually there is some software running (in an embedded CPU or a host machine) to handle control
- Build an abstraction around the low-level interface to your hardware (i.e. memory-mapped I/O) and then have separate wrappers for synthesis and simulation
- Start developing firmware & software before the hardware is finished

Hardware-Software Co-Sim: How?

- Write a software-level abstraction for the low-level interface
  - Abstracting away the interface (instead of simulating the interface) saves overhead and is much faster
- Use polymorphism or compiler tricks to test higher-level software code against simulation, then compile the same code to interface with actual hardware

Verilator

- Compile Verilog code to C++ for simulation and testing
- Very performant, but limited mostly to cycle-based simulation. Still have to manually handle stepping, propagation, and such
- Built specifically for testbenching, hence much better than something like CXXRTL
- Capable of multi-threading
- Verilator Docs

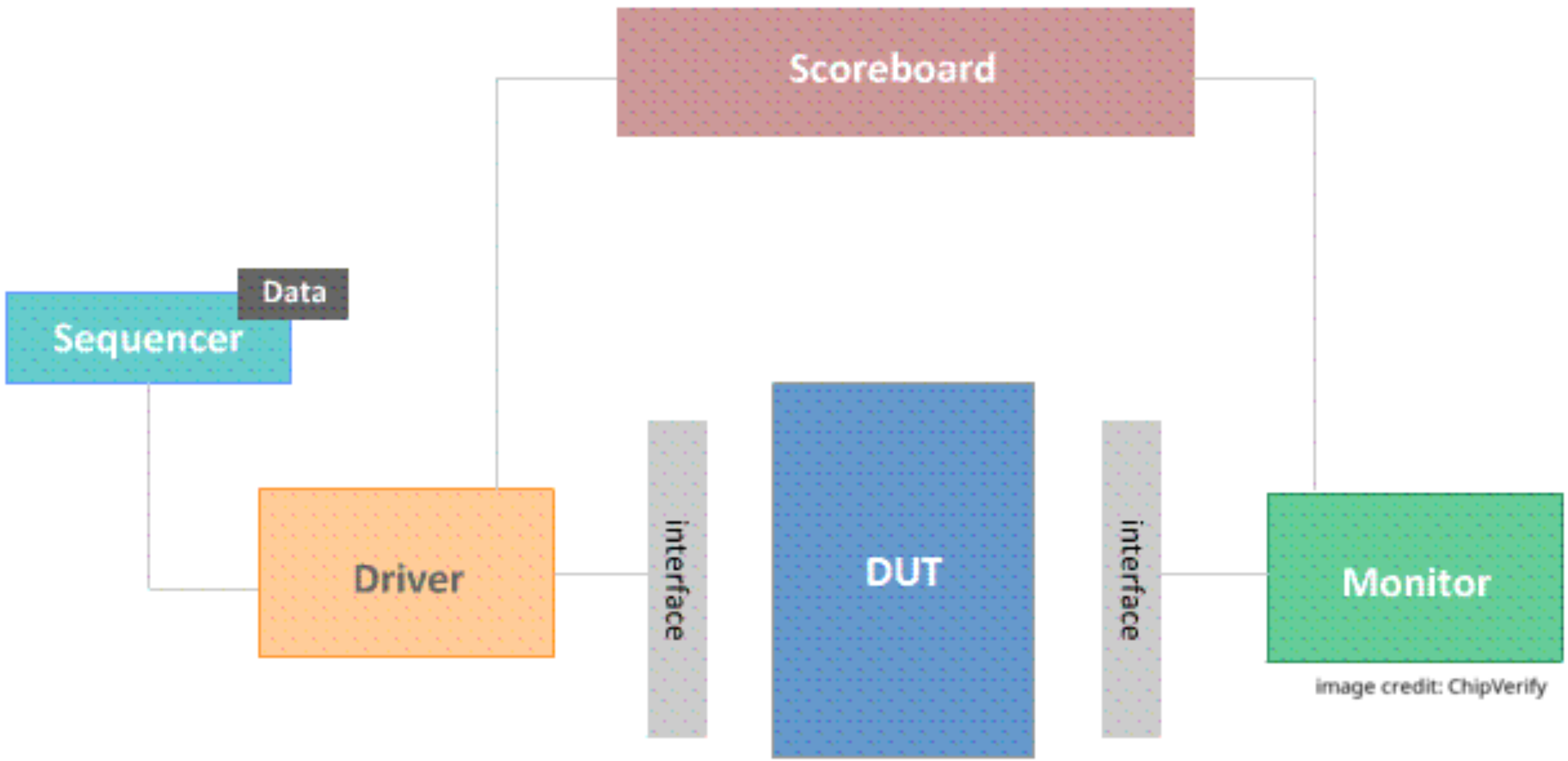
Demo: Verilator

PyVerilator

- Wrap of Verilator-generated C++ into Python using `ctypes`
- Just a cleaner / easier-to-use Verilator alternative
- PyVerilator example

Components of a Testbench

The “common” way to testbench complicated hardware:



CocoTB

- Testing is inherently parallel: run many drivers, scoreboards, monitors, etc.
- Synchronization between the parallel operations is important
- CocoTB is built around the use of Python coroutines, which means you can spawn unlimited things in parallel and sync them up; more powerful for complex designs than something like Verilator or PyVerilator
- Use of Python gives access to tons of libraries, useful for algorithm modeling

Demo: CocoTB Basics

Demo: CocoTB Scoreboarding

Demo: CocoTB Golden Models