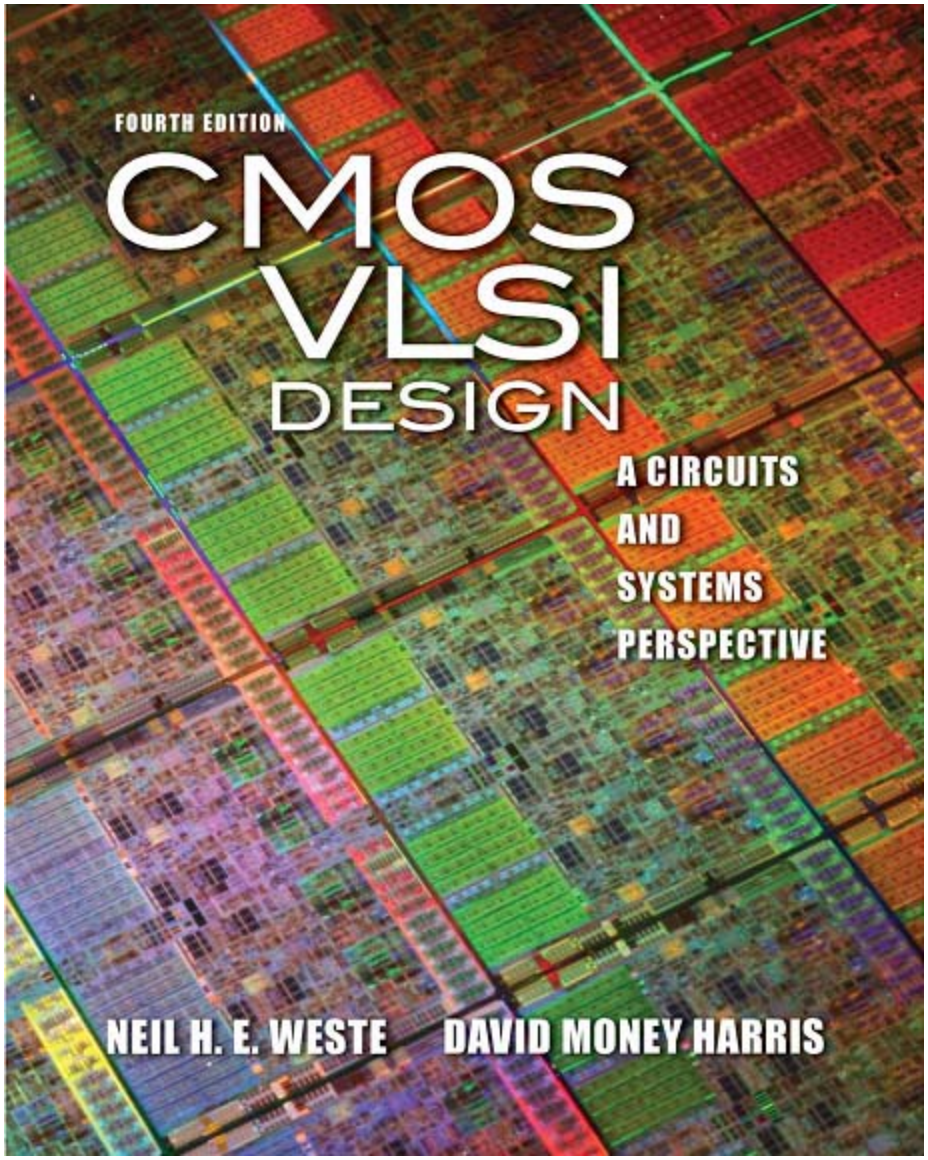


Hardware Description Languages

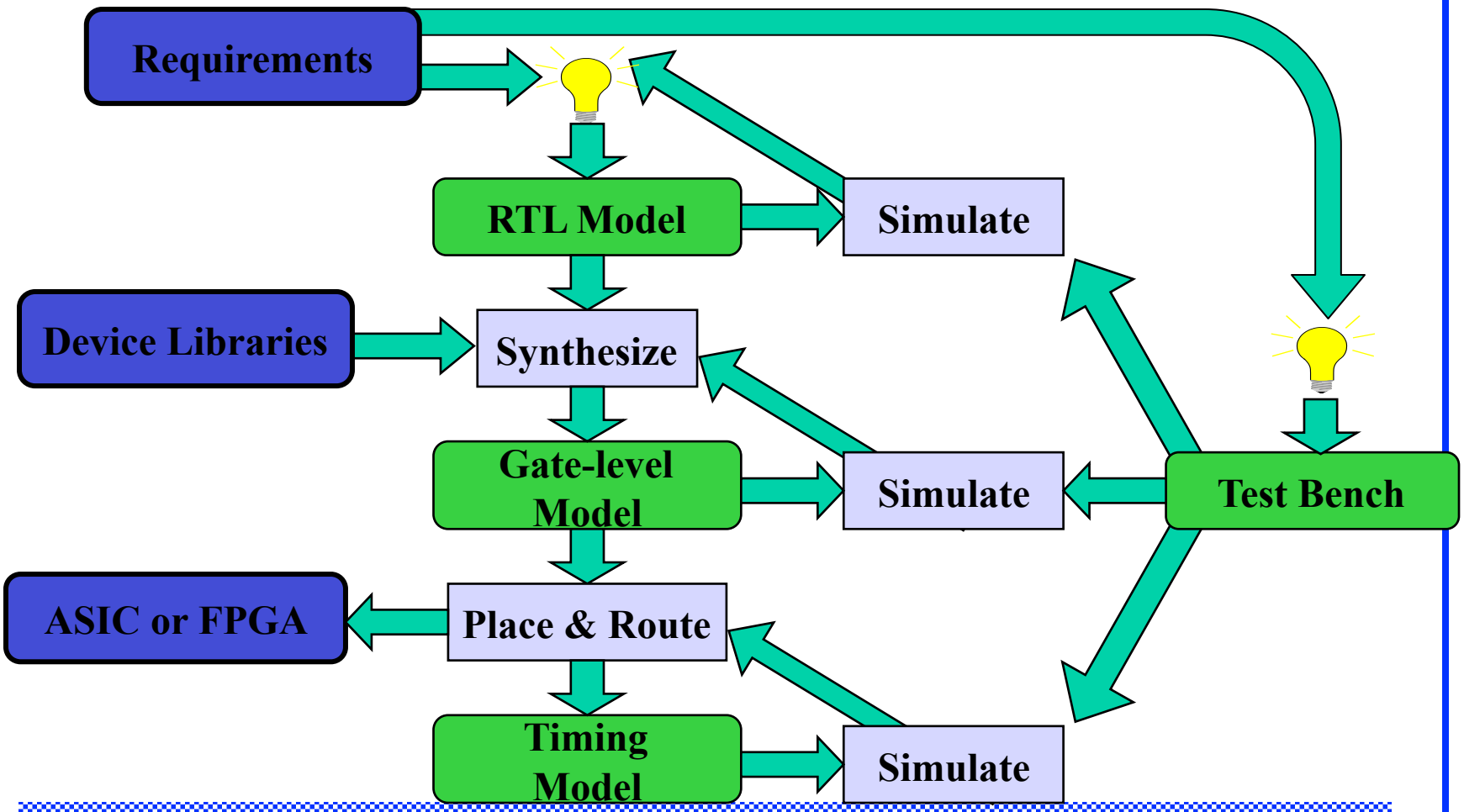
Outline

- HDL Overview
- Why not use “C”?
  - Concurrency
  - Hardware datatypes / Signal resolution
  - Connectivity / Hierarchy
  - Hardware simulation
- Basic VHDL Concepts
- Basic VerilogHDL Concepts
- SystemC Introduction
- SystemVerilog Introduction

HDL Overview

- Hardware Description Languages
  - Used to model digital systems
  - Can model anything from a simple gate to a complete system
  - Support design hierarchy
  - Support Hardware Design Methodology
- Can model “real” hardware (synthesizable)
- Can model behavior only (e.g. for test)
- Both are non-proprietary, IEEE standards
- Behavioral and structural coding styles

Basic Design Methodology



Why Not Use C or C++?

- HDLs need to support characteristics of “real” hardware
  - Concurrency
  - Hardware datatypes / Signal resolution
  - Connectivity / Hierarchy
  - Circuit timing
- HDLs must support hardware simulation
  - Time
  - Cycle-accurate or Event-driven (for simulation speed)
- Note: C++ **has** been extended for hardware
  - SystemC

Basic Comparison

Verilog

- Similar to C
- Popular in commercial, on coasts of US
- Designs contained in “module”s

VHDL

- Similar to Ada
- Popular in Military, midwest US
- Designs contained in “entity” “architecture” pairs

Concurrency

- HDLs **must** support concurrency
  - Real hardware has many circuits running at the same time!
- Two basics problems
  - Describing concurrent systems
  - Executing (simulating) concurrent systems

Describing Concurrency

- Many ways to create concurrent circuits
  - initial/always (Verilog) and process (VHDL) blocks
  - Continuous/concurrent assignment statements
  - Component instantiation of other modules or entity/architectures
- These blocks/statements execute in parallel in every VHDL/Verilog design

Executing Concurrency

- Simulations are done on a host computer executing instructions sequentially
- Solution is to use time-sharing
  - Each process or always or initial block gets the simulation engine, in turn, one at a time
- Similar to time-sharing on a multi-tasking OS, with one major difference
  - There is no limit on the amount of time a given process gets the simulation engine
  - Runs until process requests to give it up (e.g. “wait”)

Process Rules

- If the process has a sensitivity list, the process is assumed to have an implicit “wait” statement as the last statement
  - Execution will continue (later) at the first statement
- A process with a sensitivity list **must not** contain an explicit wait statement

Sensitivity List

With Explicit List

```
XYZ_Lbl: process (S1, S2)
begin
  S1 <= '1';
  S2 <= '0' after 10 ns;
end process XYZ_Lbl;
```

Without Explicit List

```
XYZ_Lbl: process
begin
  S1 <= '1';
  S2 <= '0' after 10 ns;
  wait on S1, S2;
end process XYZ_Lbl;
```

Incomplete Sensitivity Lists

- Logic simulators use sensitivity lists to know when to execute a process
  - Perfectly happy not to execute proc2 when “c” changes
  - Not simulating a 3-input AND gate though!
- What does the synthesizer create?

```
-- complete
proc1: process (a, b, c)
begin
  x <= a and b and c;
end process;
```

```
-- incomplete
proc2: process (a, b)
begin
  x <= a and b and c;
end process;
```

Datatypes

- Verilog has two groups of data types
  - Net Type – physical connection between structural elements
    - Value is determined from the value of its drivers, such as a continuous assignment or a gate output
    - wire/tri, wor/trior, wand/triand, trireg/tri1/tri0, supply0, supply1
  - Variable (Register) Type – represents an abstract data storage element
    - Assigned a value in an always or initial statement, value is saved from one assignment to the next
    - reg, integer, time, real, realtime

Datatypes

- VHDL categorizes objects in to four classes
  - Constant – an object whose value cannot be changed
  - Signal – an object with a past history
  - Variable – an object with a single current value
  - File – an object used to represent a file in the host environment
- Each object belongs to a type
  - Scalar (discrete and real)
  - Composite (arrays and records)
  - Access
  - File

Hierarchy

- Non-trivial designs are developed in a hierarchical form
  - Complex blocks are composed of simpler blocks

VHDL	Verilog
Entity and architecture	Module
Function	Function
Procedure	Task
Package and package body	Module

Hardware Simulation

- A concurrent language allows for:
  - Multiple concurrent “elements”
  - An event in one element to cause activity in another
    - An event is an output or state change at a given time
    - Based on interconnection of the element’s ports
  - Logical concurrency — software
  - True physical concurrency — e.g., “<=” in Verilog

Discrete Time Simulation

- Models evaluated and state updated only at time intervals — $n\tau$
  - Even if there is no change on an input
  - Even if there is no state to be changed
  - Need to execute at finest time granularity
  - Might think of this as cycle accurate — things only happen

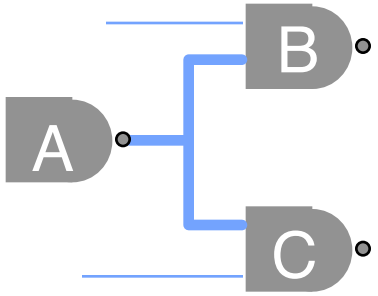
    @(posedge clock)
- You could do logic circuits this way, but either:
  - Lots of gate detail lost — as with cycle accurate above (no gates!)
  - Lots of simulation where nothing happens — every gate is executed whether an input changes or not.

Discrete Event Simulation

- Discrete Event Simulation…also known as Event-driven Simulation
  - Only execute models when inputs change
  - Picks up simulation efficiency due to its selective evaluation
- Discrete Event Simulation
  - Events — changes in state at discrete times. These cause other events to occur
  - Only execute something when an event has occurred at its input
  - Events are maintained in time order
  - Time advances in discrete steps when all events for a given time have been processed

Discrete Event Simulation

- Quick example
  - Gate A changes its output.
  - Only then will B and C execute



- Observations
  - The elements in the diagram don't need to be logic gates
  - DE simulation works because there is a sparseness to gate execution — maybe only 12% of gates change at any one time.
    - The overhead of the event list then pays off

Synthesis

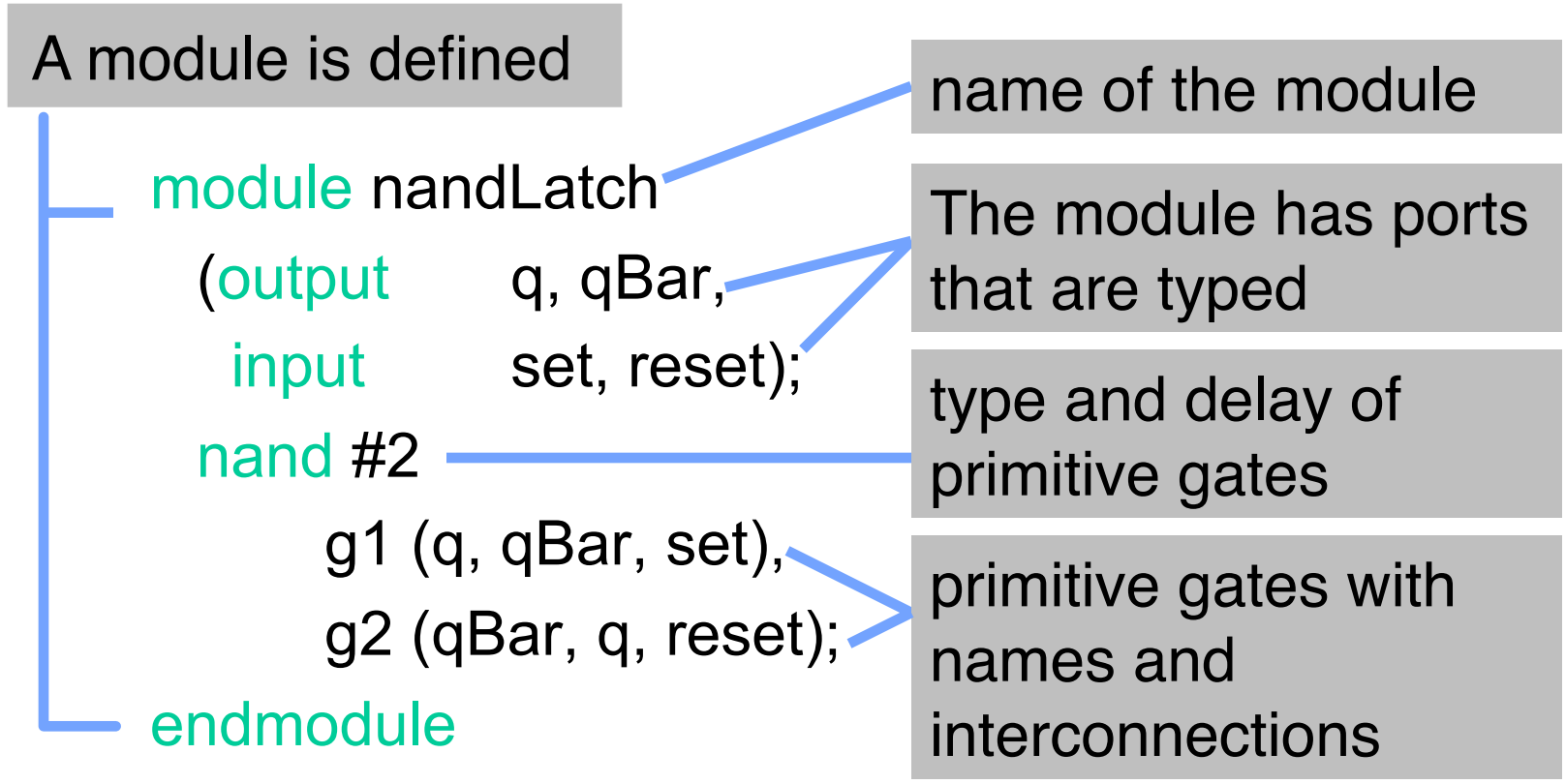
- Translates register-transfer-level (RTL) design into gate-level netlist
- Restrictions on coding style for RTL model
- Tool dependent

Basic Verilog Concepts

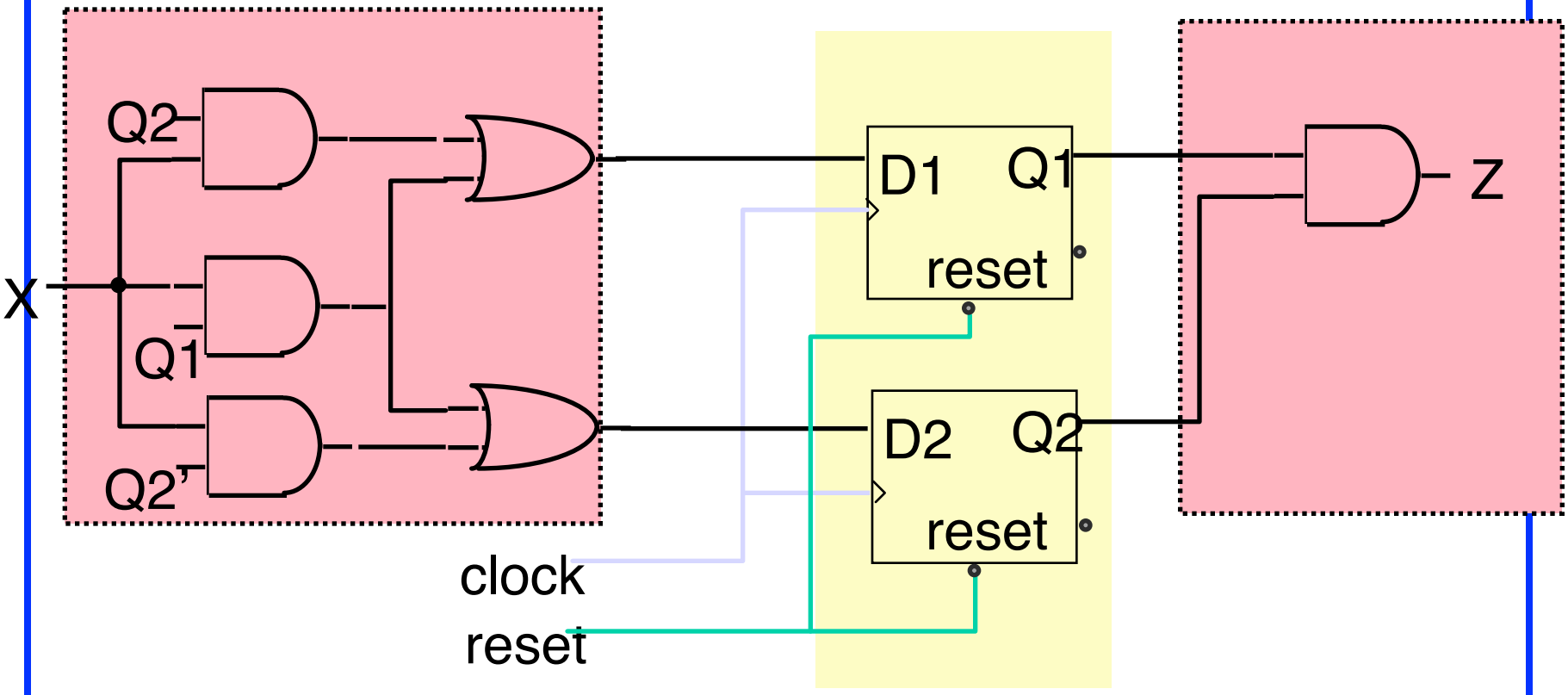
- Interfaces
- Behavior
- Structure

A Gate Level Model

- A Verilog description of an SR latch



A Behavioral Model - FSM



Organization for FSM

- Two always blocks
  - One for the combinational logic — next state and output logic
  - One for the state register

Behavioral Specification

```
module FSM (x, z, clk, reset);
        input          clk, reset, x;
        outputz;
        reg     [1:2]  q, d;
        reg            z;

        always
          @(posedge clk or negedge reset)
            if (~reset)
              q <= 0;
            else  q <= d;

        always @(x or q)
        begin
          d[1] = q[1] & x  |    q[2] & x;
          d[2] = q[1] & x  |  ~q[2] & x;
          z = q[1] & q[2];
        end

endmodule
```

The sequential part (the D flip flop)

The combinational logic part

next state

output

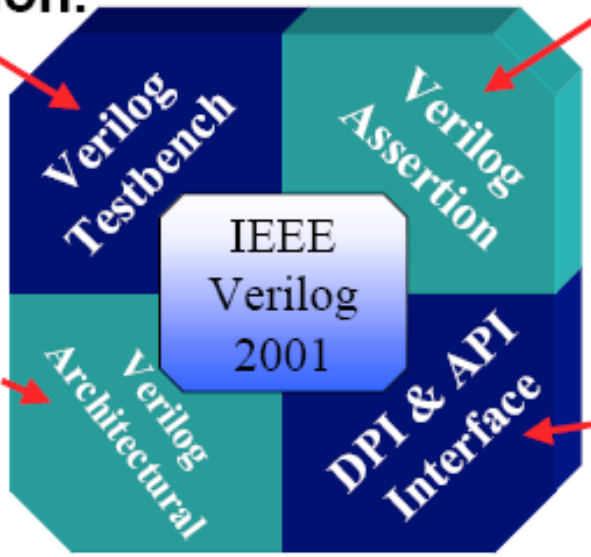
SystemC

- C++ class library developed to support system level design (Electronic System Level, ESL)
- Intended to cope with both hardware and software
- IEEE 1666 Standard
- Supports concurrency, hierarchy, signals, time
- Supports transaction level modeling
- Supported natively by Modelsim

SystemVerilog

- Charter: Extend Verilog IEEE 2001 to higher abstraction levels for Architectural and Algorithmic Design , and Advanced Verification.

Transaction-Level Full Testbench Language with Coverage

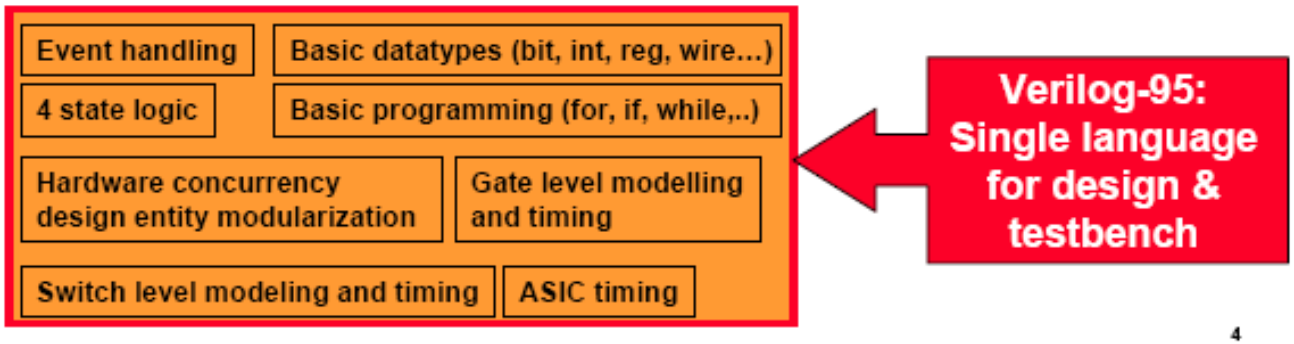


Advanced verification capability for semiformal and formal methods. The Assertion Language Standard For Verilog

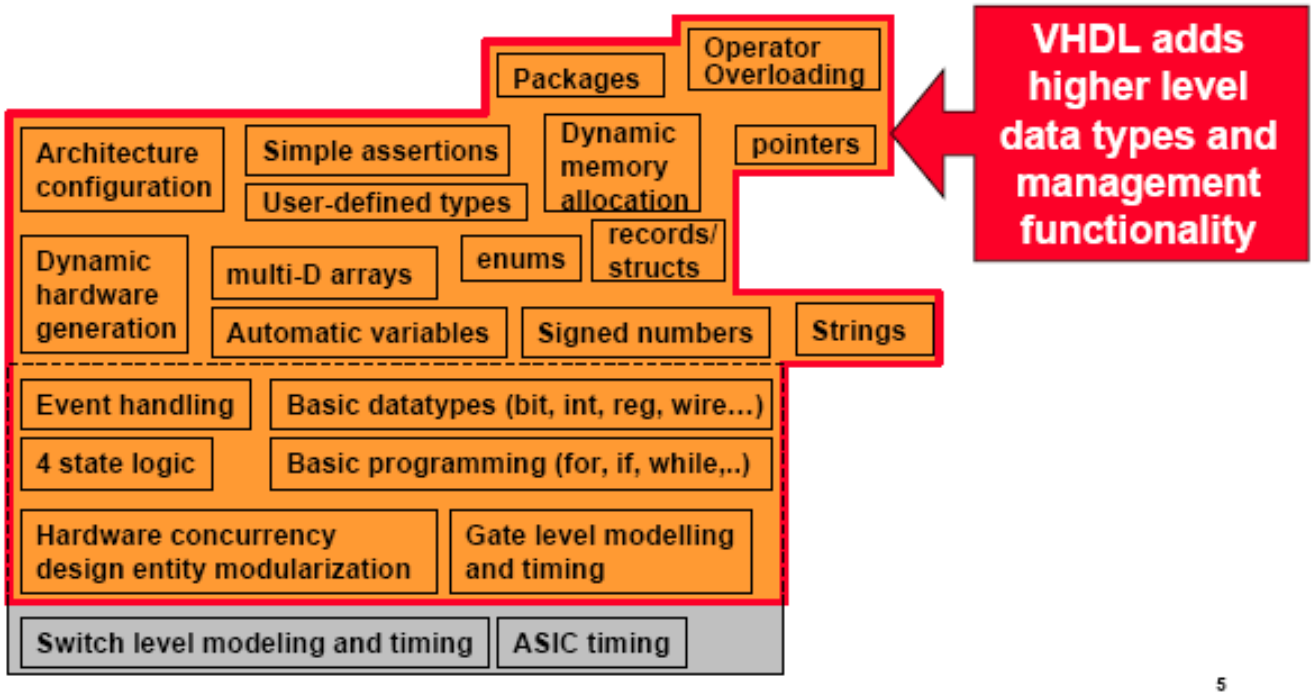
Design Abstraction: Interface semantics, abstract data types, abstract operators and expressions

Direct C interface, Assertion API and Coverage API

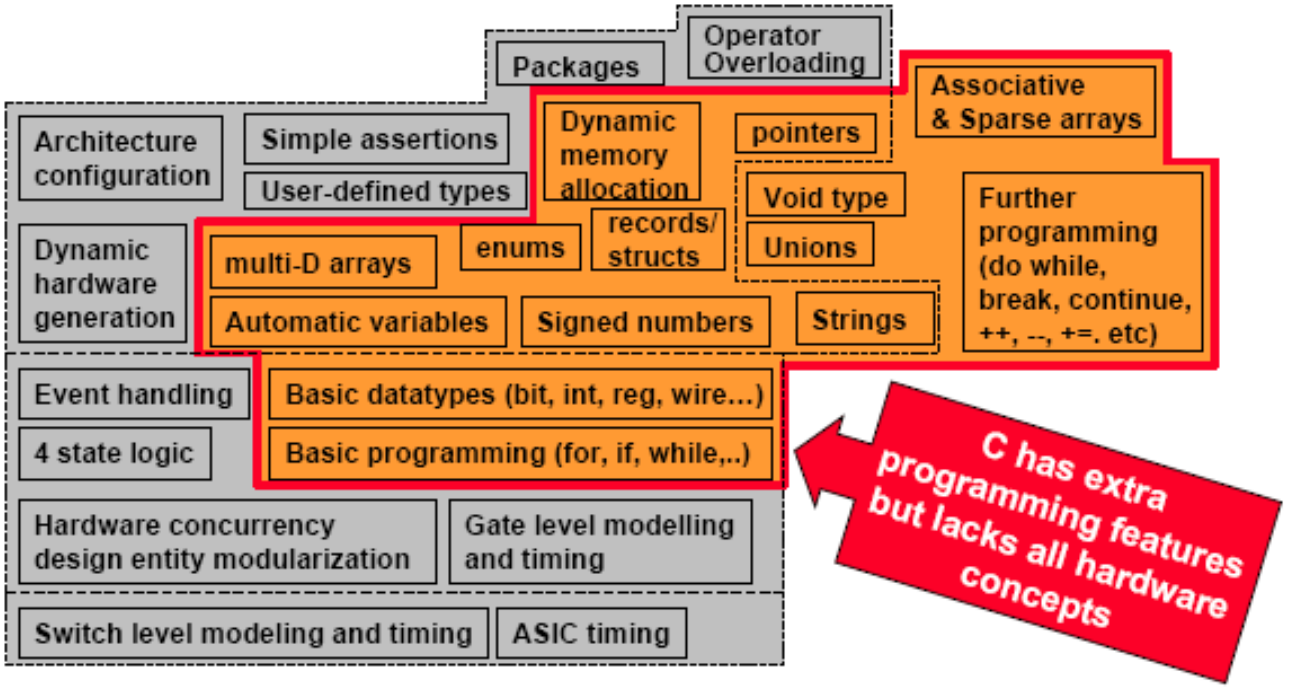
Verilog-95


Event handling
Basic datatypes (bit, int, reg, wire…)
4 state logic
Basic programming (for, if, while,..)
Hardware concurrency design entity modularization
Gate level modelling and timing
Switch level modeling and timing
ASIC timing
Verilog-95: Single language for design & testbench

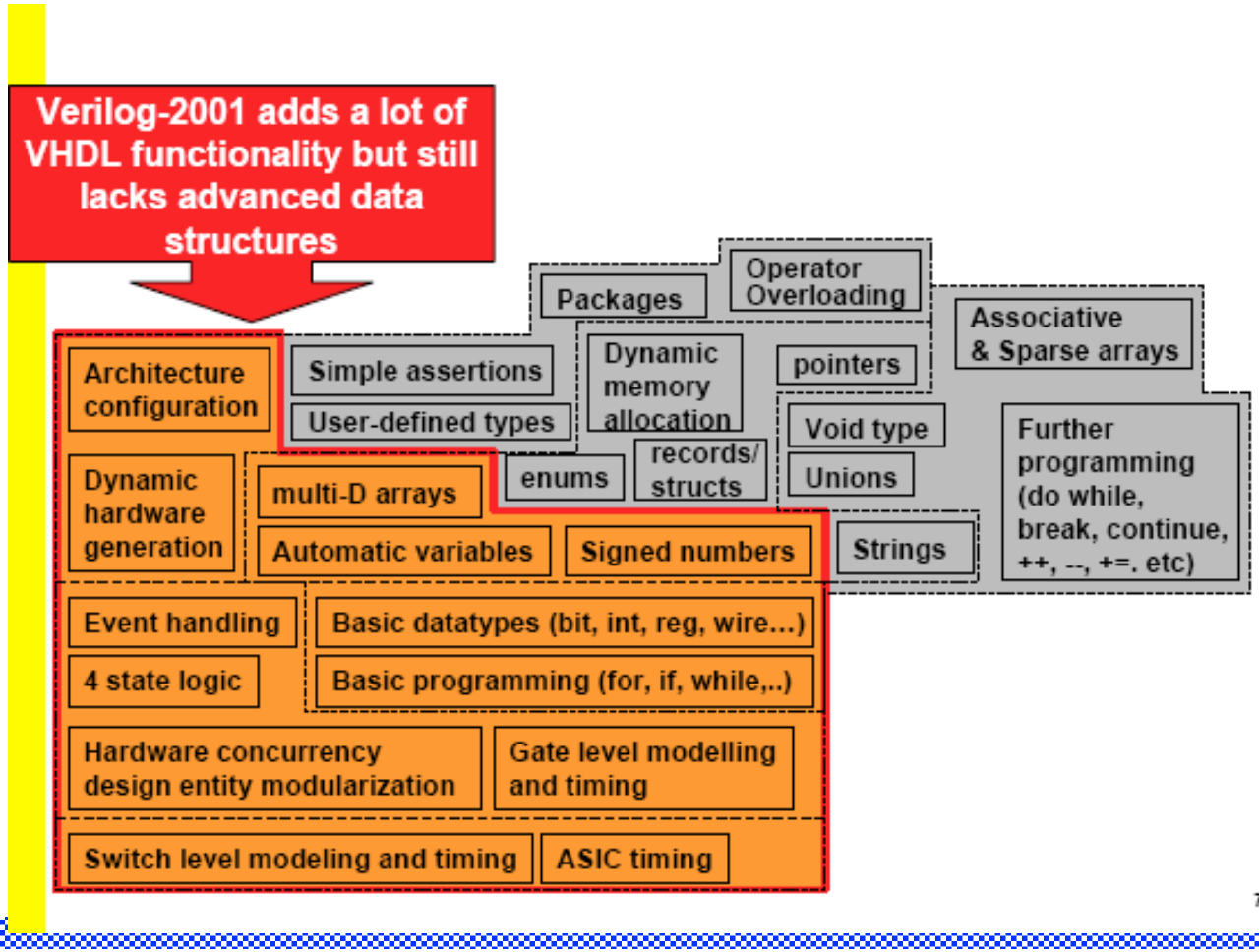
VHDL Richer Than Verilog



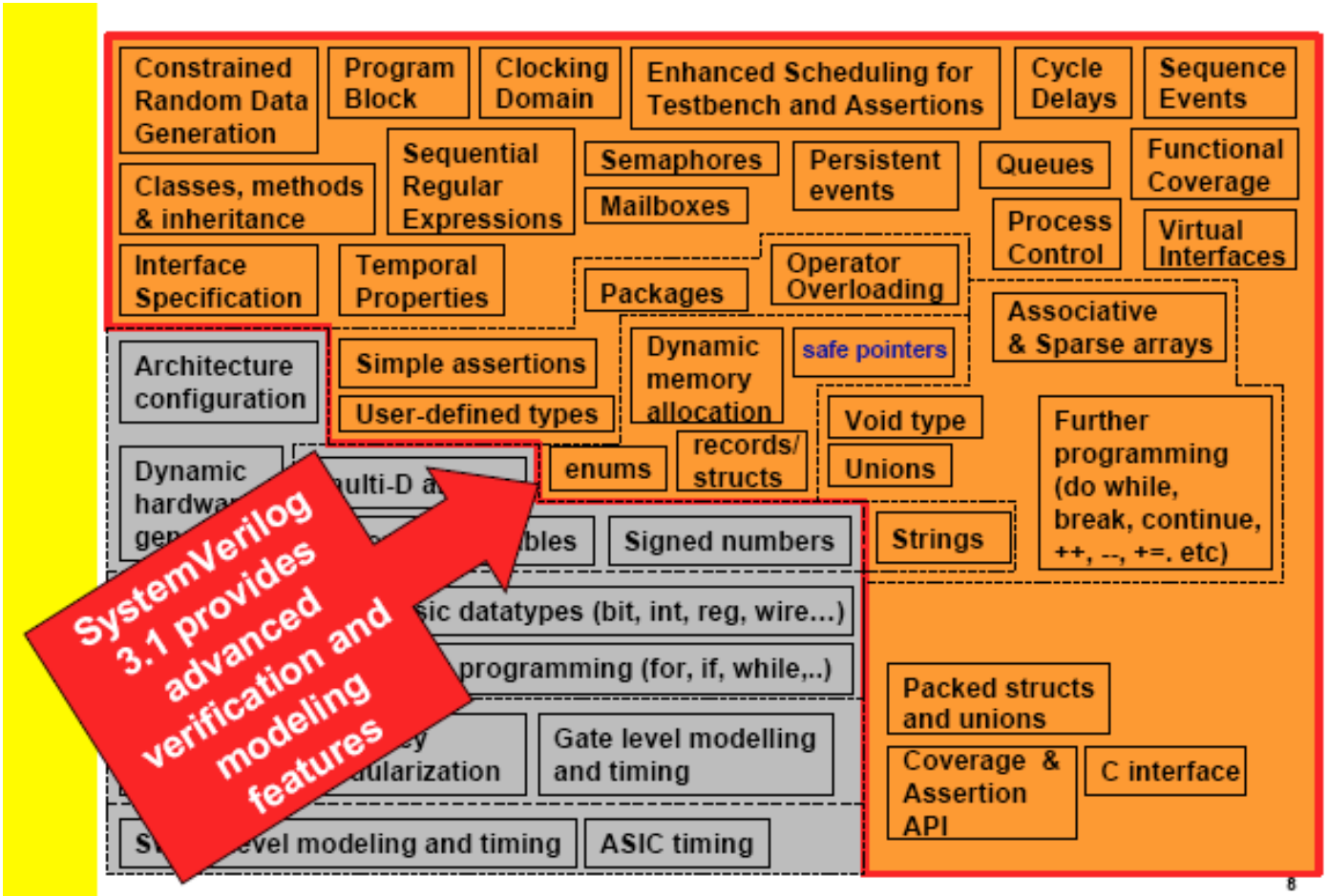
C Can't Do Hardware



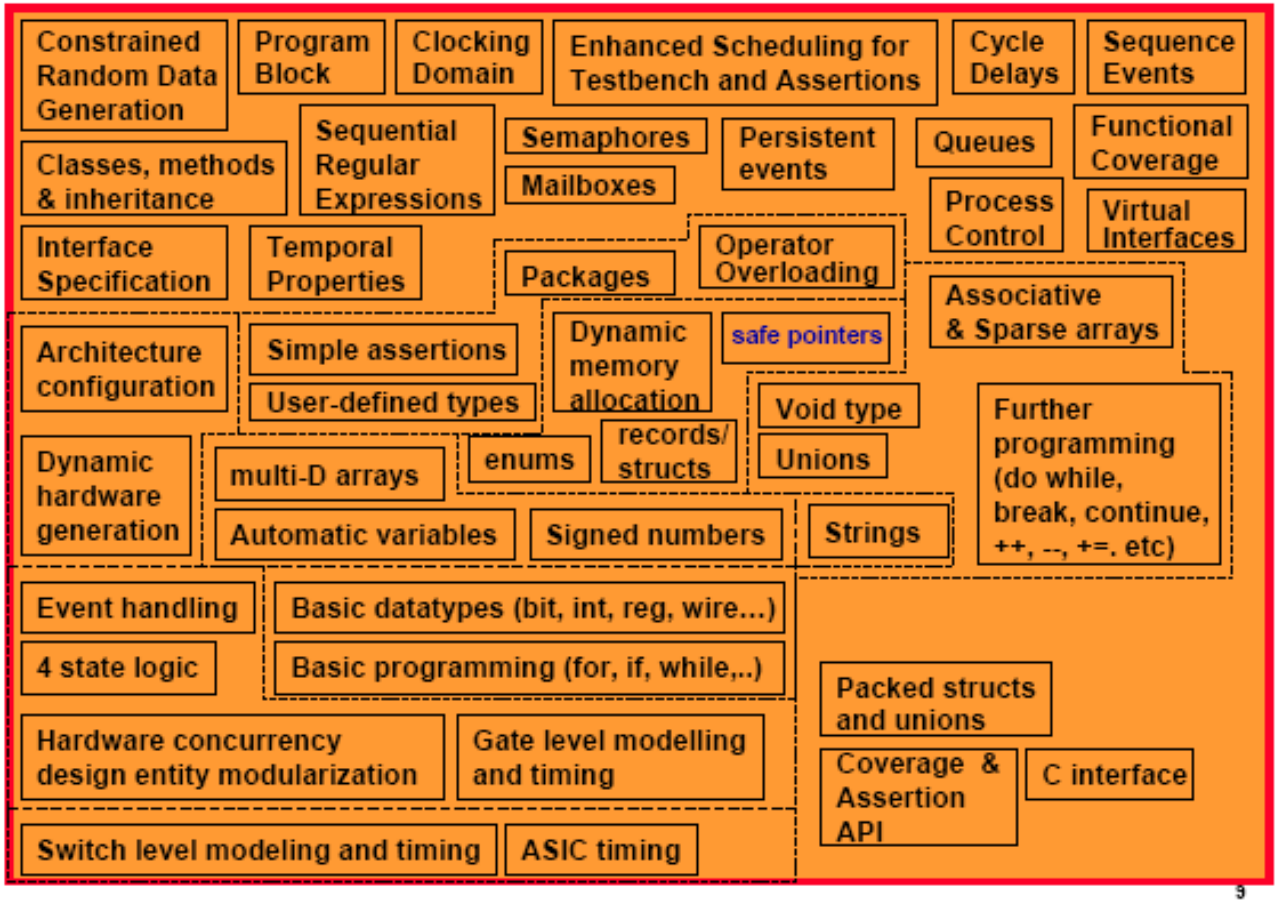
Verilog-2001



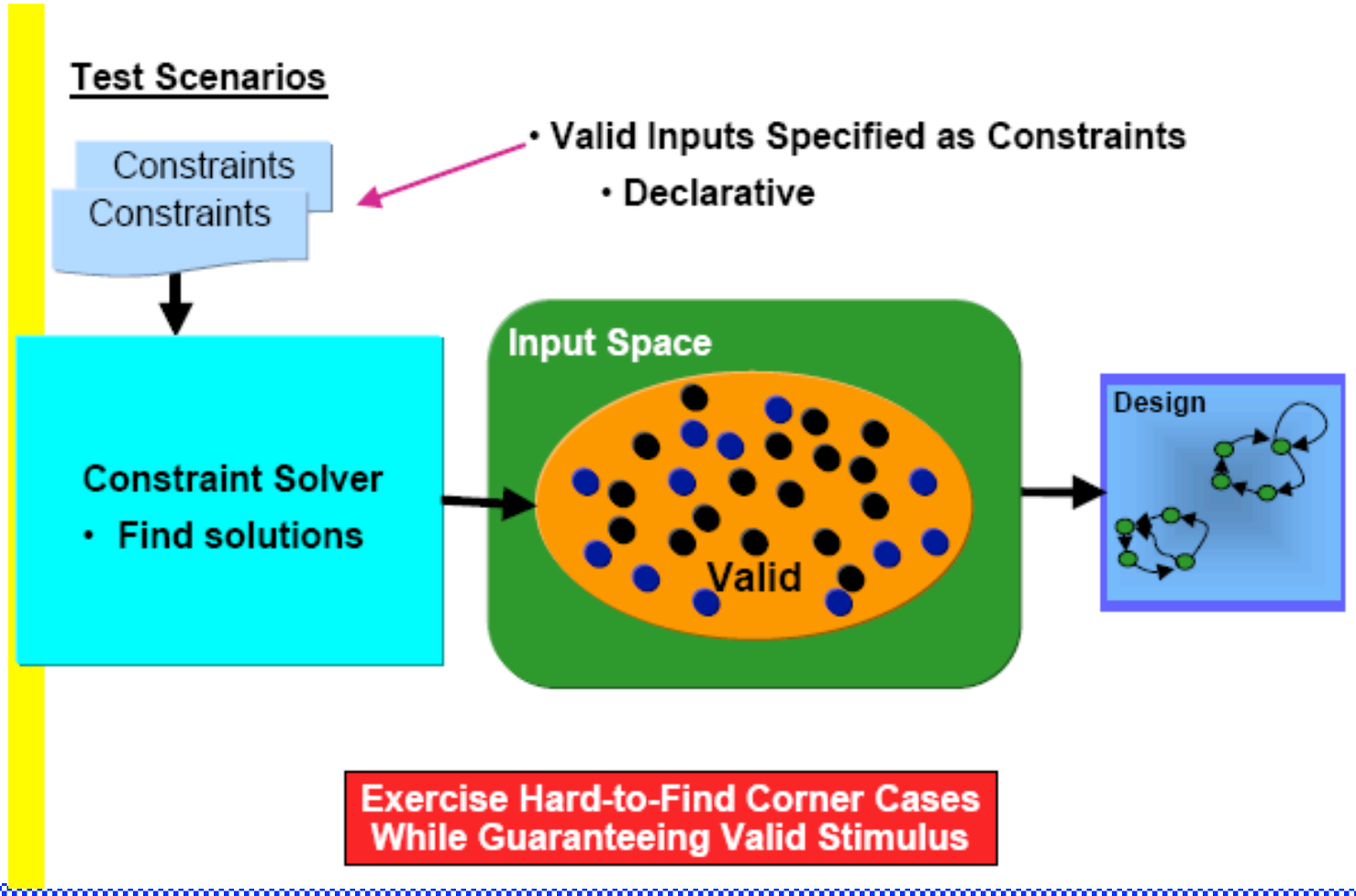
Verification and Modeling



SystemVerilog: Unified Language



Constrained Random



Basic Constraints

- Constraints are Declarative

```
class Bus;
rand bit[15:0] addr;
rand bit[31:0] data;
randc bit[3:0] mode;
constraint word_align {addr[1:0] == 2'b0;}
endclass
```

- Calling `randomize` selects values for all random variables in an object such that all constraints are satisfied
  - Generate 50 random data and word_aligned addr values

```
Bus bus = new;
repeat (50)
  if ( bus.randomize() == 1 ) // 1=success,0=failure
    $display ("addr = %16h data = %h\n", bus.addr,
    bus.data);
  else
    $display ("Randomization failed.\n");
```

Weighted Random Case

- Randomly select one statement
  - Label expressions specify distribution weight

```
randcase
    3 : x = 1;      // branch 1
    1 : x = 2;      // branch 2
    a : x = 3;      // branch 3
endcase
```

  - If a == 4:
    - branch 1 taken with 3/8 probability (37.5%)
    - branch 2 taken with 1/8 probability (12.5%)
    - branch 3 taken with 4/8 probability (50.0%)

Program Block

- Purpose: Identifies verification code
- A `program` differs from a `module`
  - Only initial blocks allowed
  - Special semantics
    - Executes in *Reactive* region

      design → clocking/assertions → program
  - Program block variables cannot be modified by the design

```
program name (<port_list>);
  <declarations>;// type, func, class, clocking…
  <continuous_assign>
  initial <statement_block>
endprogram
```

The Program block functions pretty much like a C program
Testbenches are more like software than hardware

Why Use Assertions?

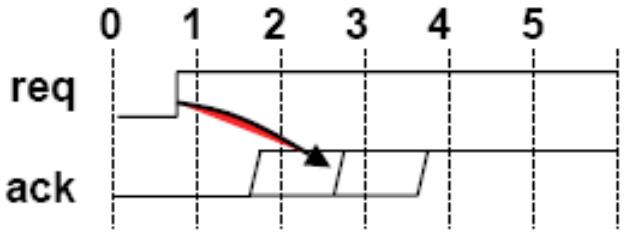
- Limitations of Directed testing
  - To be practical, testing has to be high level
  - Locating a logic error can take a lot of time
  - New tests may need to be written to close in on failure
  - Signal relationships are complex and lower level.
- Assertions target interesting signal relationships
  - Like handshake signals, bus protocols etc.
  - Execute in parallel with Verification tests
  - Efficiently capture Verification IP (bus protocols etc)
  - Often reusable across design and/or project

What Assertions Can Do

- Find logic errors earlier
  - Detect low-level errors that functional tests miss
  - Explicitly indicate time when a failure occurs
  - Explicit hierarchical locations and signal names
- Coverage of expected or unexpected events
  - Assertions coverage permits uses beyond checking
    - How many times an event occurs
    - Proof that a negative event did NOT happen

Assertions

A concise description of [un]desired behavior



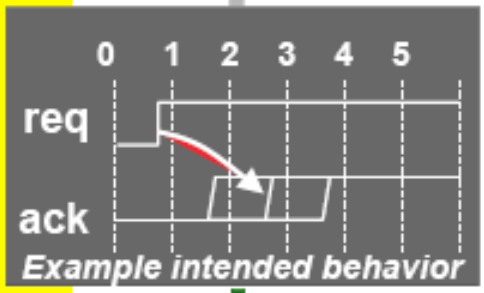
Example intended behavior

"After the request signal is asserted, the acknowledge signal must come 1 to 3 cycles later"

More Concise Than VHDL

SVA Assertion

```
property req_ack;
 @(posedge clk) req ##[1:3] $rose(ack);
endproperty
as_req_ack: assert property (req_ack);
```



HDL Assertion

VHDL

```
sample_inputs : process (clk)
begin
  if rising_edge(clk) then
    STROBE_REQ <= REQ;
    STROBE_ACK <= ACK;
  end if;
end process;
protocol: process
  variable CYCLE_CNT : Natural;
begin
  loop
    wait until rising_edge(CLK);
    exit when (STROBE_REQ = '0') and (REQ = '1');
  end loop;
  CYCLE_CNT := 0;
  loop
    wait until rising_edge(CLK);
    CYCLE_CNT := CYCLE_CNT + 1;
    exit when ((STROBE_ACK = '0') and (ACK = '1')) or (CYCLE_CNT = 3);
  end loop;
  if ((STROBE_ACK = '0') and (ACK = '1')) then
    report "Assertion success" severity Note;
  else
    report "Assertion failure" severity Error;
  end if;
end process protocol;
```

Assertion Based Verification

- Make Assertions part of Design and Verification flows
  - » Embed design assumptions into the design
  - » Place protocol & functional spec checks outside (bind)
  - » Make use of Assertion Coverage data
- Consider Assertions in Test Plan
  - » Identify key protocols and target-able blocks
    - » Input assumptions, Output expectations
  - » Leverage Assertion Libraries first
    - » OVL, QVL, Checkerware
  - » Write custom Assertions (e.g SVA)
    - » Expertise and training

Assertion Characteristics

- Automated checks on signal behavior & functionality
  - Boolean statement that specifies the logical relationship between a set of signals over a specified period of time
  - Checks performed at user-specified intervals (sample points)

```
property p_one_hot;
  @(posedge Clk) disable iff(Reset)
     $onehot( {var1, var2, var3} );
endproperty
rx_fsm_one_hot : assert property( p_one_hot );
```

Embedded Assertions

- These are assertions embedded in procedural code
  - Ideal for designers – almost like active comments
  - Must have write access to the source-code to add these
  - Likely to be of use ONLY to simulator tools

```
always @ (posedge clk)
  if (cond1_is_met)…
    if (cond2_is_met)begin
      access_grant = request1 || request2;
      assert(access_grant);
```

Concurrent Assertions

- These are assertions outside of procedural code
  - Ideal for Verification - No source-code access required.
  - Totally independent of design (black-box checking)
  - Used by simulation and other tools

```
property p_one_hot;
  @(posedge Clk) disable iff(Reset)
    $onehot( {var1, var2, var3} );
endproperty

rx_fsm_one_hot : assert property( p_one_hot );
```

Debugging Assertions

- Assertions are compact code structures
  - Challenging to write, even moreso to debug
  - Need good tools to help visualize the assertion
- Questa has powerful visualization and debug tools
  - Analysis pane
    - » Lists all assertions at current hierarchical level and their stats.
  - Waveform View of assertion and it's signals
    - » Clear indication of status: active/inactive/pass/fail
  - Thread View
    - » decomposes assertions clause by clause for easy debug

Assertion Summary

- Limited visibility to signals within SOC's
  - One contributor to the Verification gap
  - Assertion Based Verification is a solution
- Questa supports Assertion Based Verification
  - Industry leading implementation of SV
    - » OOP, Functional Coverage etc.
    - » Broadest Assertion Library support
    - » SV Assertions
  - Comprehensive debug toolchain
    - » Assertion Pane
    - » Waveform display of assertions
    - » Assertion Thread Viewer