Test Benches and Verification

Introduction

- Purpose of Verification:
  - Discover as many potential bugs in the design as reasonable before sending chip out for fabrication
  - Do this by simulating chip (and chip components) in Verilog
- Why is verification important?
  - Chip fab might cost $4M and take 8 weeks
  - Very expensive and time consuming to iterate chip fab!
  - Want to get prototype correct in one to two fab cycles
  - FPGAs can rely more on using the prototype for debug
    - But, note, it is more difficult to debug hardware than a simulation

… Introduction

- Verification consumes more than 60% of design resources
  - People, compute cycles
- Verification mainly done with pre-synthesis code
  - Though some simulation, and other checks, are done to make sure the netlist is correct
- With increased reuse of existing Intellectual Property (“IP”), verification has become very challenging
  - IP = Predesigned blocks, internally developed, purchased or obtained from open source
  - Debugging is often harder than design!
- Focus of these Notes
  - Primarily on verification tasks likely to be performed by module level designer, and code constructs commonly used

Verification and the Team

- Designer's Responsibilities:
  - Conduct reasonable levels of ad-hoc verification of design through simulation
  - Follow good coding practices to ease primary verification task
  - Include assertions in code as appropriate
  - Design in features to aid verification
    - E.g. Allow long FSM to be started in a specific "deep" state
- System level verification usually primarily the role of a separate verification team
  - Why?
    - Whole system, not individual design verification
    - When verifying his/her own design, designer often makes same (dumb) assumptions in the test fixture as in the design
      - i.e. Misses many of the bugs, especially mis-interpretations of specification
      - A separate team with an independently derived verification plan is less likely to do this
    - Becoming more of a specialty with own tools, methodologies, etc.

Verification Tools and Methods

- It is impossible to know that you have eliminated all the bugs in a design
  - Thus it is important to use a variety of tools, techniques and methods that give you a high probability of discovering bugs
    - And to have a plan to apply them!
    - Get as many “avenues of attack” as possible
- Available tools and methods include:
  - Simulation through test fixtures
    - Including mixed level simulation
  - Inserting and tracking assertions
  - Formal verification
  - Emulation

Simulations Through Test Fixtures

- Basic concept:
  - Apply vectors to design as stimulus
  - Observe outputs, and internal nodes, for correct functionality
- Key Questions:
  - Where to you get the vectors?
  - How do you observe the outputs?
  - What are the available coding styles?

Sources of Verification Vectors

1. From expected functionality
   - Vectors designed to exercise expected functions of chip or block
     - From specification or understanding of function of chip/block
     - Prioritized from “must work” to “would like to work”
2. From Higher Level Model
   - Obtain vectors for individual blocks from a higher level behavioral model
     - E.g. C model developed for project
   - Example: Run video stream through C model of MPEG encoder
     - Extract examples from this to run through Motion Estimator
     - C model here is an example of a “reference behavioral model”

… Sources of Simulation Vectors

3. Vectors added specifically as a result of production of verification plan
   - E.g. Vectors specifically designed to test “difficult” aspects of design
     - Features that were hard to design
     - Modules are more likely to be buggy
     - E.g. Bus arbiters
   - E.g. vectors designed to increase the “coverage” of the design
     - Increase code and functional coverage
4. Random vectors
   - Run random vectors
   - Compare results with same vectors run in a higher level model

… Sources of Simulation Vectors

5. System level vectors – simulating the chip in its entirety
   - Important to do a LOT of this
   - Very slow and time consuming
   - While design is incomplete, can be a mixed behavioral (e.g. C) and RTL simulation
     - Using Verilog Programming Language Interface (PLI)
   - Requires good behavioral models for interface chips – Memories, etc.

Observing Correctness

1. Observe in Waveform Viewer
2. Observing results of assertions
3. Try to write 'self-checking' test fixtures, that analyze the results and inform you of correctness.
   - Useful as it means you can automatically check other parts of a design when you redesign some portion.

```
#10 dec = 1;
#28 if (zero == 1'b1) $display ("Check 1 passed")
    else $display ("Error: Check 1 FAILED");
```

   - Try to take to a higher level. i.e. Incorporate `understanding' of function into self-checking feature

```
integer testData;  // test data being used
integer ExpectedDelay; // expected delay for test data
initial
  begin
     testData = 4;
     in = testData;
     ....
     ExpectedDelay = testData * 10;
     #ExpectedDelay if (zero == 1'b1) $display ("Check 1 passed")
                   else $display ("Error: Check 1 FAILED");
```

Verilog Code for Test Fixtures…Approaches

- Can use any syntactically correct code
- Choose test vector generation approach:
  - On-the-fly generation:
    - Use continuous loops for repetitive signals
    - Use simple assignments for signals with few transitions (e.g. reset)
    - Use tasks to generate specific waveform sets
  - Read vectors stored as constants in an array
  - Read vectors from a file
- Choose timing approach:
  - Relative Timing, or
  - Absolute Timing
- Generate clock separately from vectors
- Whenever possible check simulation results within test fixture
  - Against a stored set of 'expected' results, or
  - Against an internal model of expected behavior

Examples…On the fly generation

- Use a task to generate an often repeated vector set

```
task refresh;
// generate a RAS before CAS refresh cycle
output RAS, CAS;
begin
  // assume RAS and CAS high on entry
  #5 RAS = 0;
  #15 RAS = 1;
  #10 CAS = 0;
  #15 CAS = 1;
  #45;  // allow refresh to complete
end

initial
  begin
        …
            refresh (RAS, CAS);
```

Test Fixture Reading Vectors from an Array

- Example below also shows use of a for loop:

```
module test_fixture;
parameter TestCycles = 20;
parameter ClockPeriod = 10;
integer       I;
reg [15:0] SourceVectors [TestCycles-1 : 0];
reg [7:0] ResultVectors [TestCycles-1 : 0];
reg [15:0] InA;  // input port of module being tested
wire [7:0] OutB; // output port of module being tested
```

…Verilog in Test Fixtures

```
initial
  begin
    SourceVector [0] = 16'h735f;   // etc.
    ResultVector [0] = 8'h5f;  // etc….not all entries here
  end
initial
  begin
    SimResults = $fopen("errdet.txt"); // open error file
    clock = 1;
    #11 for (I=0; I<=TestCycles; I = I+1); // start 1 ns into first clock
   period
      begin
          InA = SourceVector[I];
          #ClockPeriod if (OutB != ResultVector[I])
            $fdisplay(SimResults, "ERROR in loop %d \n", I);
     end
```

…Verilog in Test Fixtures…reading vectors from file

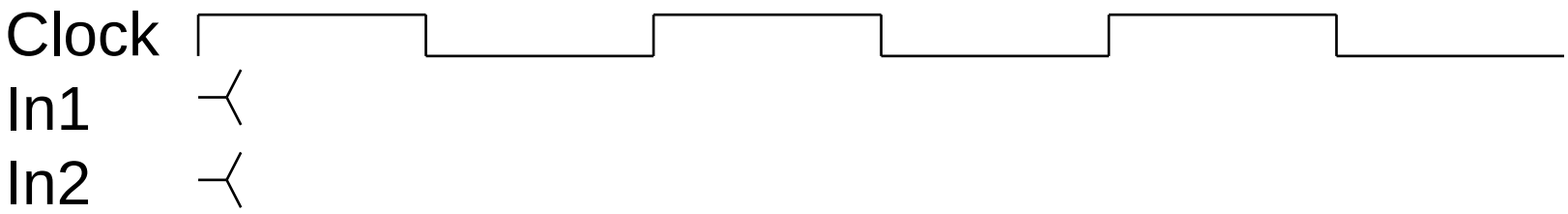
- Can also store the verification vectors in a file.
- For example, you could generate the file during the behavioral ‘C’ simulation and use during RTL verification

```
module test_fixture;
reg [15:0] SourceVectors [TestCycles-1 : 0];
initial
  begin
    $readmemh("source_vec.txt", Source_Vectors);
    …
-----------------------
source_vec.txt:
// Source Vectors for SourceVectors array for design
73hf   // first vector
beef   // second vector
```

Absolute vs. Relative timing

- Relative Timing Example:

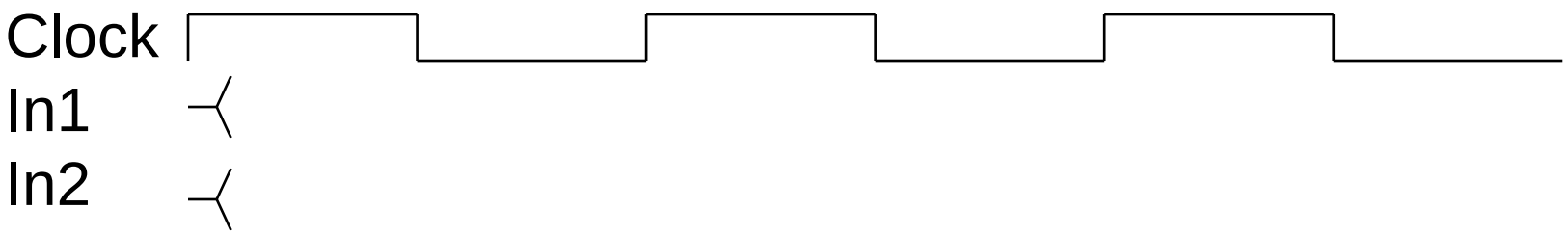
```
  module test_fixture;
parameter ClockPeriod=10;
initial
  begin
    #1 In1 = 2'b00;
       In2 = 2'b01;
    #ClockPeriod In1 = 2'b01;
                 In2 = 2'b00;
    #ClockPeriod In1 = 2'b11;
                 In2 = 2'b10;
  end
```



…Absolute vs. Relative Timing

- Absolute Timing Example:

```
module test_fixture;
parameter ClockPeriod=10;
initial
  fork
    #1 In1 = 2'b00;
    #1 In2 = 2'b01;
    #(ClockPeriod+1) In1 = 2'b01;
    #(ClockPeriod+1) In2 = 2'b00;
    #(ClockPeriod*2+1) In1 = 2'b11;
    #(ClockPeriod*2+1) In2 = 2'b10;
  join
```



Putting it All Together

- What a test fixture might look like:

```
module test_fixture;
  \\ declare variables assigned within test fixture as type reg
  reg clock;
  \\ declare variables that come from module output ports as type wire
  wire [7:0] data_out;
  initial \\ test fixture contents
    begin
    …
    end
  \\ declare non-synthesisezed parts, e.g. memories
  SRAM1  m1 (clock, …);
  \\ declare module to be tested
  top  u1 (clock, … , data_out);
endmodule
```

Behavioral Models for Non-Synthesized Designs

- Often need to model the following:
- Parts provided by other vendors (ask Vendor first)
- modules in your chip that are not synthesized, such as memories, some arithmetic units, analog portions.
- Cells in cell library
- Approaches to modeling these modules:
- Can use any correct Syntax verilog for model
- User Defined Primitives (UDP) are useful for combinational logic and designs containing a single register
  - Examples: NOR2 gate and DFF from CMOSX library
- Use a spec param block to capture timing requirements
  - Example: Embedded memory array
- Verilog-A used to model analog portions
- Must verify these models carefully too

User Defined Primitives

- primitive prim_dff(q,cp,d);
- output q;

```
        reg q;
        input cp,d;
        table
//   cp     d     :     q     :     q+
        r     1     :     ?     :     1;
        r     0     :     ?     :     0;
        n     ?     :     ?     :     -;
        *     0     :     0     :     0;
        *     1     :     1     :     1;
        ?     *     :     ?     :     -;
        endtable
        endprimitive
```

State transition table

Inputs : Current State : Next State

r = rise

n = fall

* = any possible transition (edge)

? = don't care (0,1,x) (level)

- = no change

User Defined Primitives

- primitive prim_dff(q,cp,d);
- output q;

```
        reg q;
        input cp,d;
        table

//      cp      d       :       q       :       q+
        r       1       :       ?       :       1;
        r       0       :       ?       :       0;
        n       ?       :       ?       :       -;
        *       0       :       0       :       0;
        *       1       :       1       :       1;
        ?       *       :       ?       :       -;
        endtable
        endprimitive
```

Primitive Declaration

Rising edge on cp ➔ next q = d

falling edge on clock➔ q stays same

other clock transitions (to/from x) ➔ no change

Ignore edges on d

specparam blocks

- Used to specify timing for non-synthesized logic.
- Again, example from CMOSX cell library….

Min : typical : max

```
`celldefine
`timescale 1ns / 10ps
module DFF(Q, QBAR, CP, D);
output Q, QBAR;
input CP, D;
specify
        specparam CP_01_PD10_QBAR = 0.320:0.685:1.75;
        specparam CP_01_PD01_Q = 0.270:0.629:1.68;
        specparam CP_01_PD01_QBAR = 0.261:0.616:1.71;
        specparam CP_01_PD10_Q = 0.320:0.628:1.55;

        specparam SLOPE0$CP$QBAR = 0.308:0.478:0.831;
        specparam SLOPE1$CP$Q = 0.258:0.609:1.59;
        specparam SLOPE1$CP$QBAR = 0.169:0.403:1.03;
        specparam SLOPE0$CP$Q = 0.451:0.714:1.32;

        specparam STANDARDLOAD = 0.350:0.350:0.350;

        specparam tSU_D = 0.30:0.60:1.40;
        specparam tHOLD_D = 0.10:0.05:0.01;
        specparam MPWL_CP = 0.20:0.30:0.90;
        specparam MPWH_CP = 0.08:0.20:0.60;
        specparam MPER_CP = 0.40:0.80:2.20;
        specparam MFT_CP = 4.00:39.00:380.00;
```

… DFF module from CMOSX lib

```
specparam FanoutLoad$CP = 0.0147:0.0216:0.0309;
specparam FanoutLoad$D = 0.0104:0.0135:0.0184;
specparam FanoutLoad$Q = 0.00504:0.0106:0.0117;
specparam FanoutLoad$QBAR = 0.0114:0.0127:0.0223;

(CP=>QBAR)=(CP_01_PD01_QBAR, CP_01_PD10_QBAR);
(CP=>Q)=(CP_01_PD01_Q, CP_01_PD10_Q);

$setup(D, edge[01] CP, tSU_D);
$hold(edge[01] CP, D, tHOLD_D);
$width(negedge CP, MPWL_CP);
$width(posedge CP, MPWH_CP);
$period(posedge CP, MPER_CP);

endspecify

prim_dff U1(Q_int,CP,D);
not U2 (QBAR,Q_int);
buf U3 (Q,Q_int);

endmodule
`endcelldefine
```

Clock – Q / Qbar delays

(rising Q, falling Q)

Checks based on parameters

Assertions

- Code blocks that check for correct and incorrect behavior
  - Put inside RTL code (but do not synthesize)
    - Usually inserted by designer
  - System Verilog allows more concise assertions, but can also be written in normal Verilog
- Example (Verlog95):

```
// synopsys off
`ifdef Assertions_on
// check ONE bus request granted ONE clock cyle after any reqest
always@(posedge clock)
if ((|request) & (~|grant))  // request, no active grants
  begin
       @(posedge clock) // wait one cycle
       if (~|grant) $display("ERROR: bus access not granted");
       else if ((grant[0] + grant[1] + grant[2] + grant[3])>1)
         $display ("ERROR: multiple buses accesses granted");
  end
`endif
// synopsys on
```

Formal Verification

- Equivalency Checking
  - Determines that two designs are logically equivalent
  - Examples:
    - RTL and netlist
    - Different netlists after non-design coding changes
  - Often used to help verify output of synthesis
- Model Checking
  - Trying to prove or disprove that a circuit possesses a property that is part of a more abstract, higher-level specification
    - E.g. Correct design capture of a Finite State Automata
    - Requires good capture of specification in a suitable language

Simulation “Engines”

There are never enough simulation cycles to complete verification

1. Event based Verilog simulator
   - Most general but slowest
2. Cycle based Verilog simulator
   - Slightly less general but faster
3. Verilog simulator hardware accelerator
   - Use hardware as a co-processor to accelerate simulation of Verilog (that does not have a lot of I/O – i.e. not all signals captured)
4. Emulation
   - i.e. Build a multi-FPGA system that can emulate the standard cell ASIC, though at a slower clock rate
   - Allows very complete verification (except for timing critical issues) but takes a lot of engineering resource

Verification Metrics

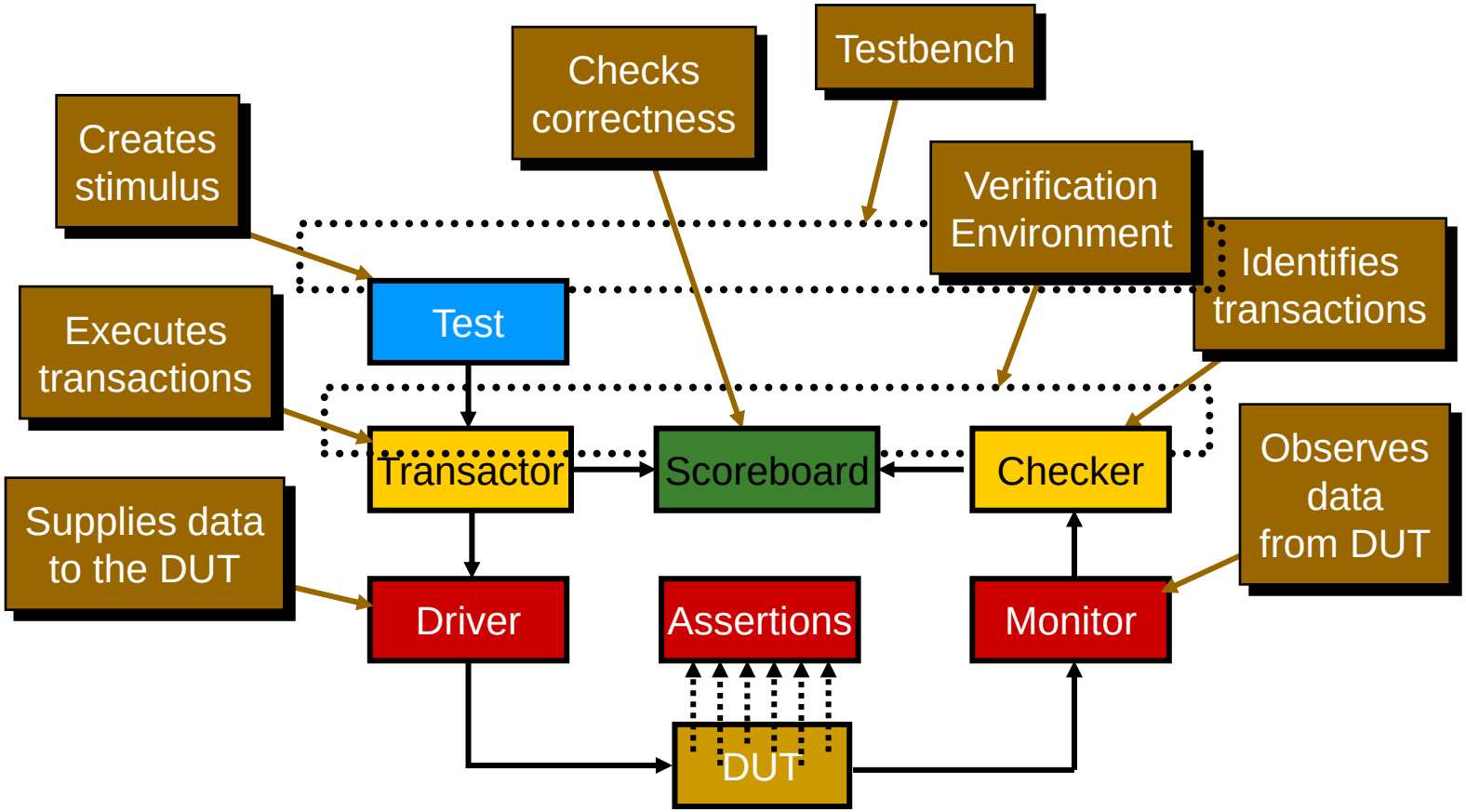
- How do you know your chip is ready for fabrication?
  - You can never know you are bug-free!
  - General solution: When cost (and opportunity cost) of more verification is higher than the cost of using the first silicon to complete the debug process
    - i.e. When it is quicker and cheaper to build the chip to find the remaining bugs
    - Note: Some bugs can be worked around with firmware
- Common Metrics:
  1. Bug discovery rate
  2. Code coverage
  3. Functional Coverage
  4. Assertion coverage

Verification Metrics

- Code Coverage
  - Has every line of code been simulated?
  - What percentage of possible paths have been simulated?
    - E.g. All alternatives in an if-then sequence
  - What percentage of possible state sequences have been simulated?
  - Requires instrumentation of code and appropriate data collecting and reporting tools
- Functional Coverage
  - Have all the functions in the specification been simulated?
    - E.g. All interface modes in a USB interface
  - Requires writing of code (SystemVerilog or integrated via PLI) to monitor the hardware that implements these functions and data collecting within the test fixture

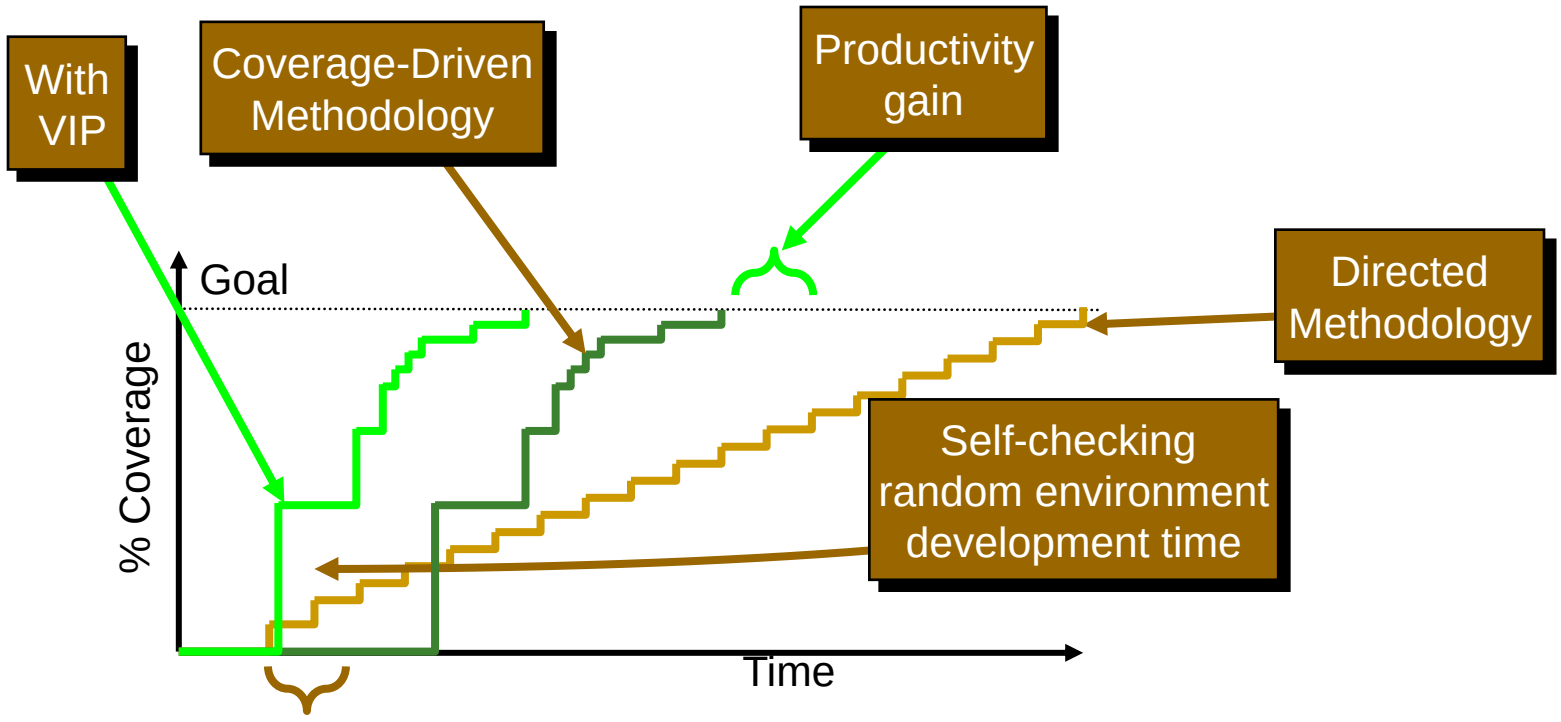
Verification Environment

Definitions

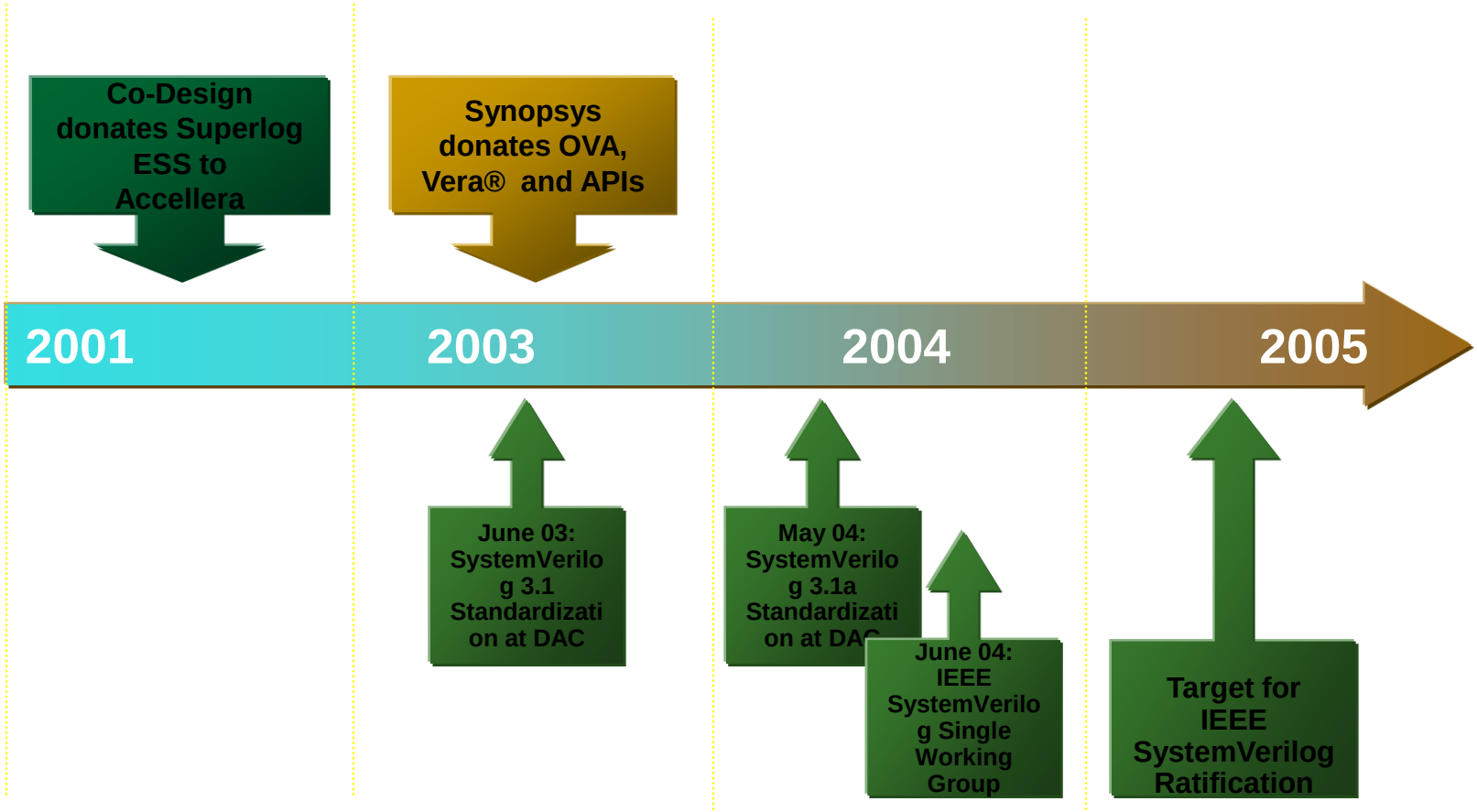


Coverage-Driven Verification

- Measure progress using functional coverage

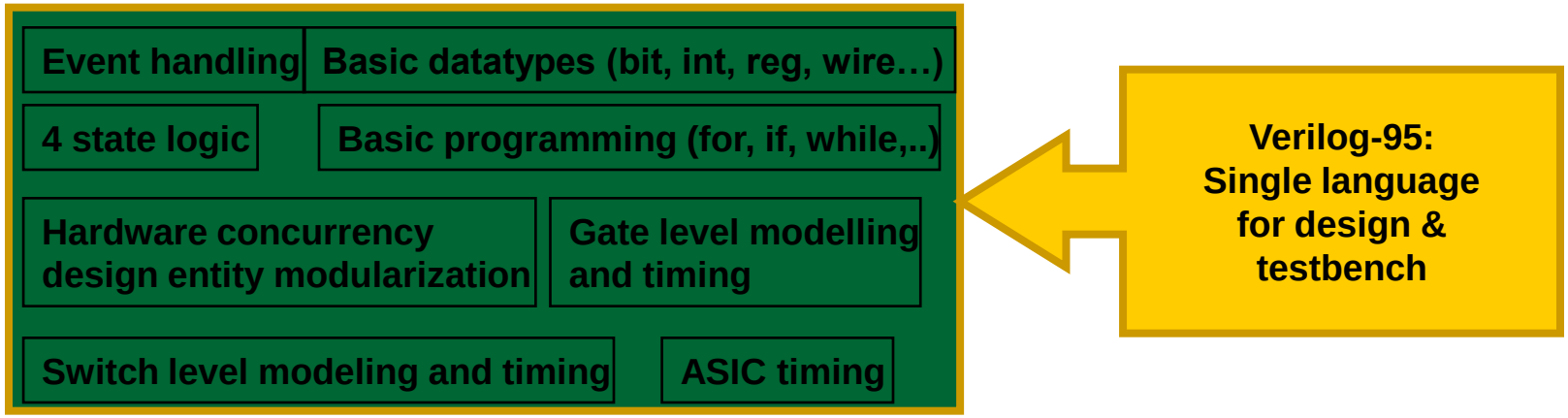


SystemVerilog Standardization Timeline

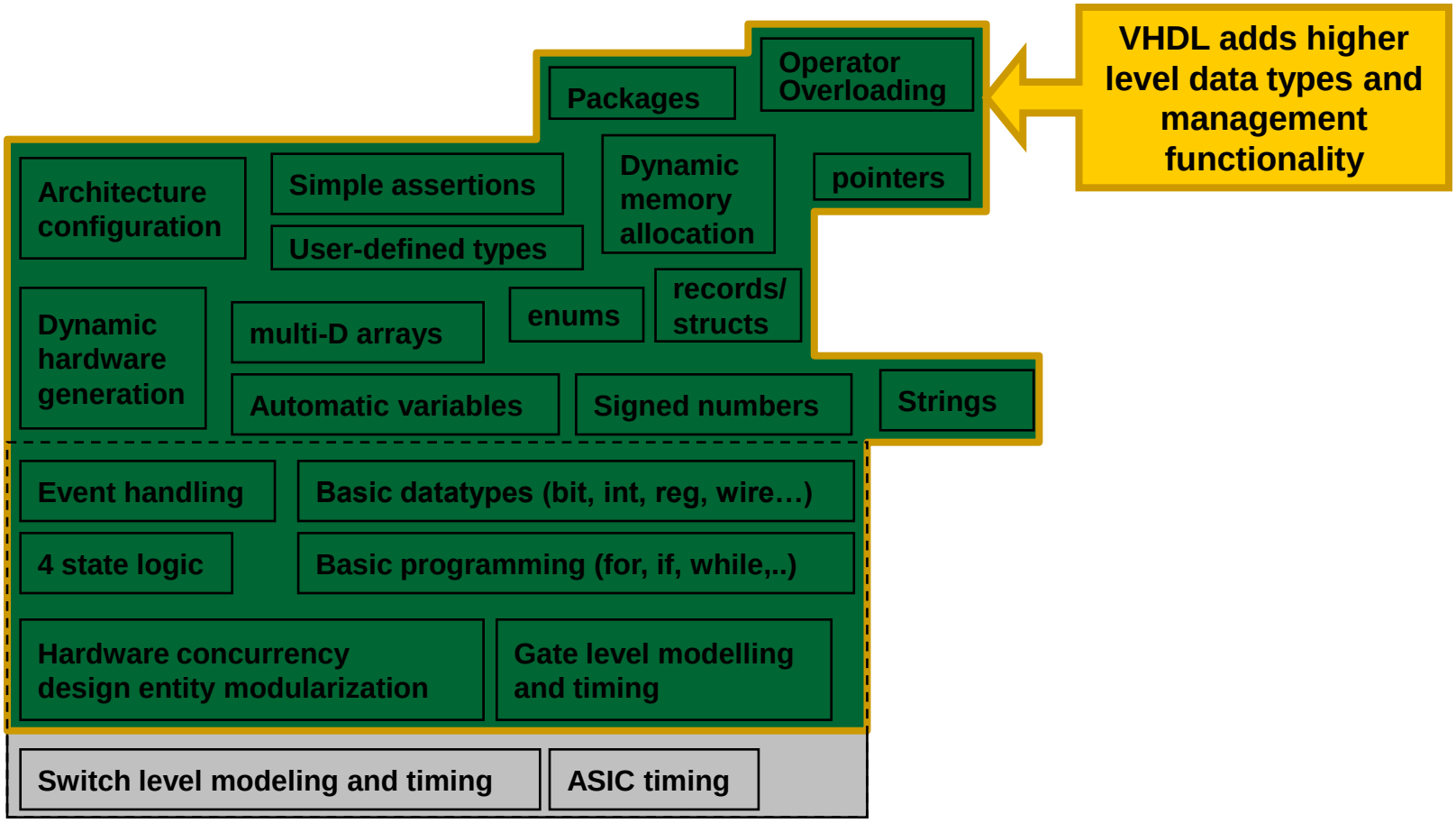


SystemVerilog: Verilog 1995

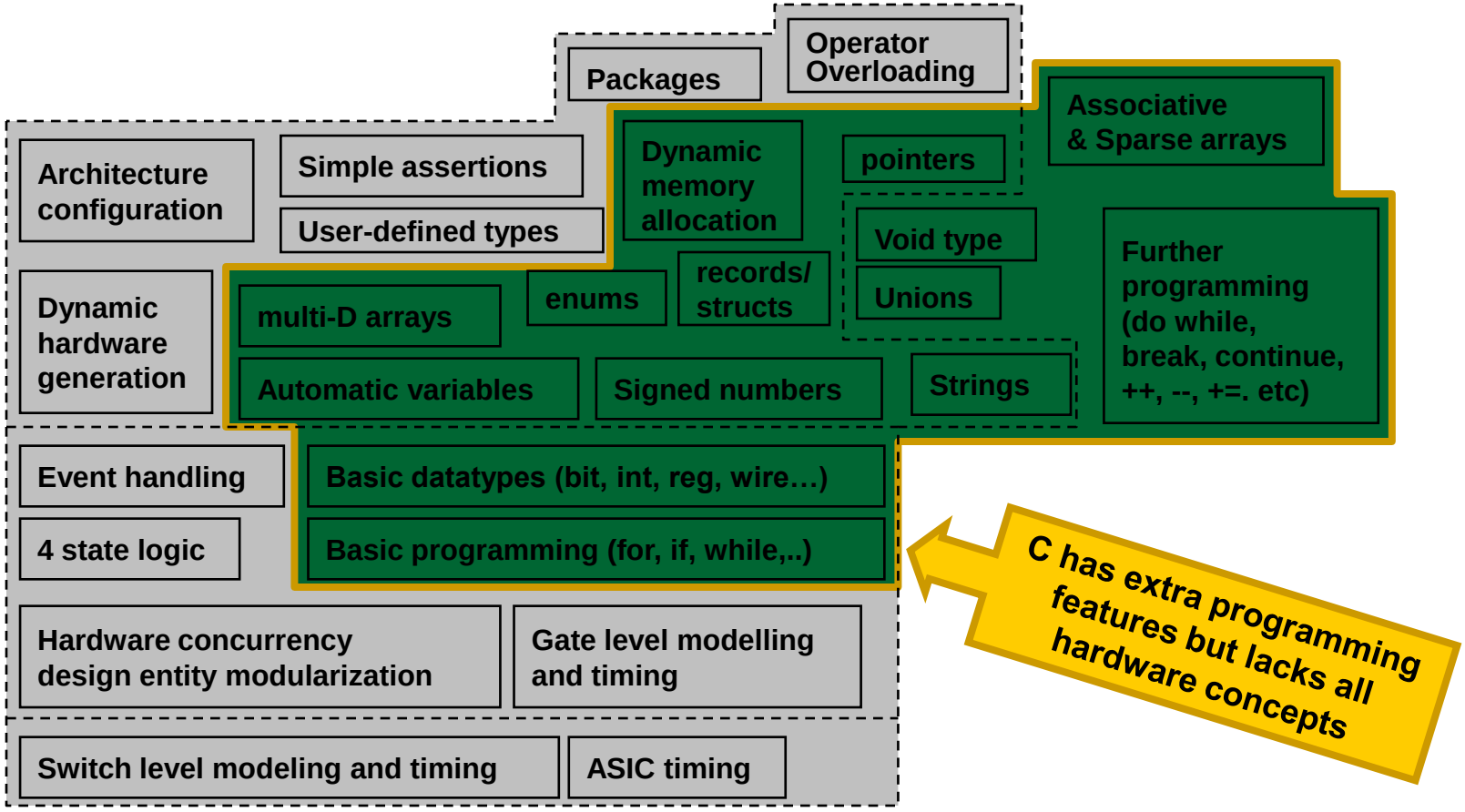
Slides provided by
David Oterra, Synopsys



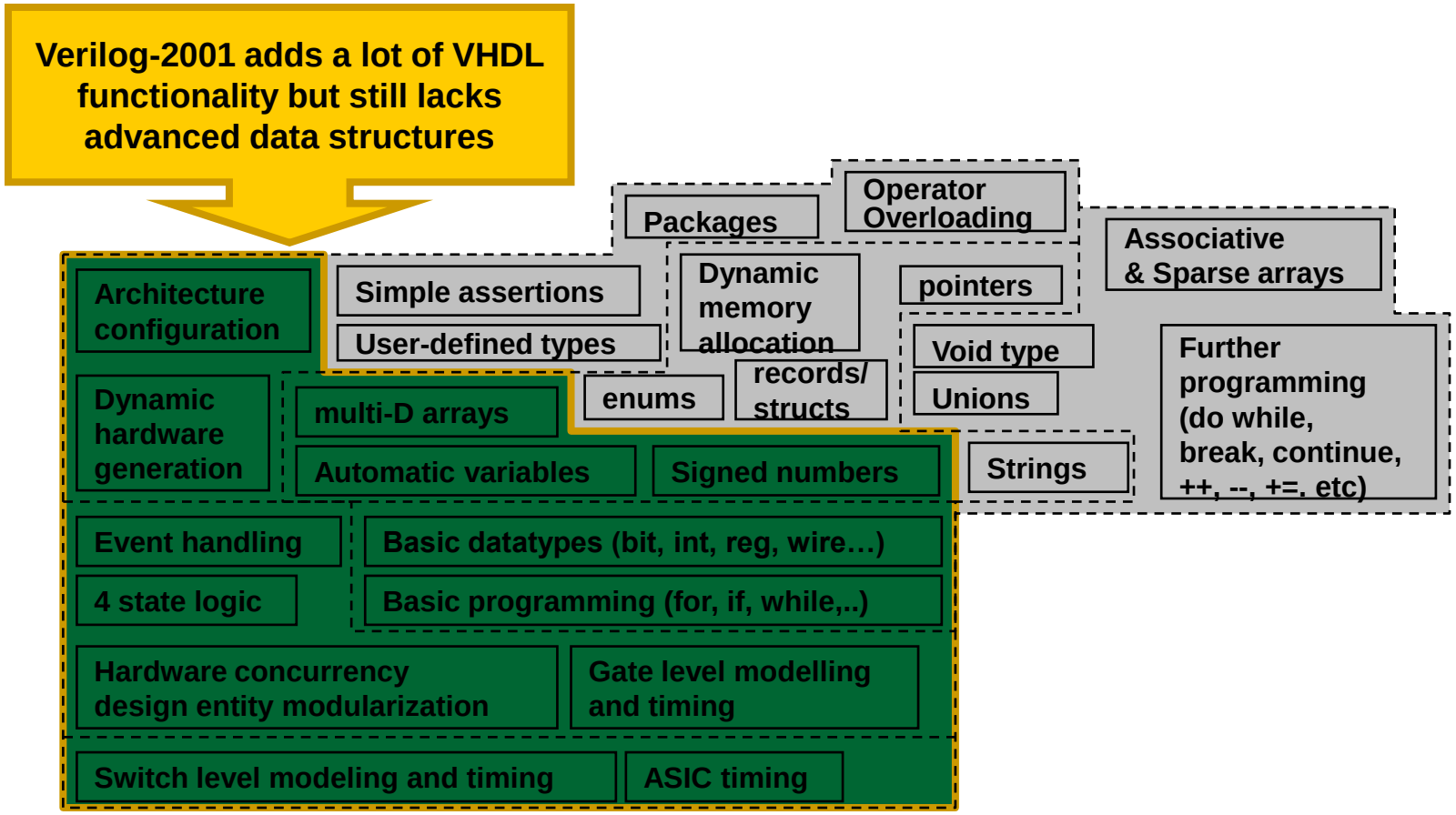
SystemVerilog: VHDL



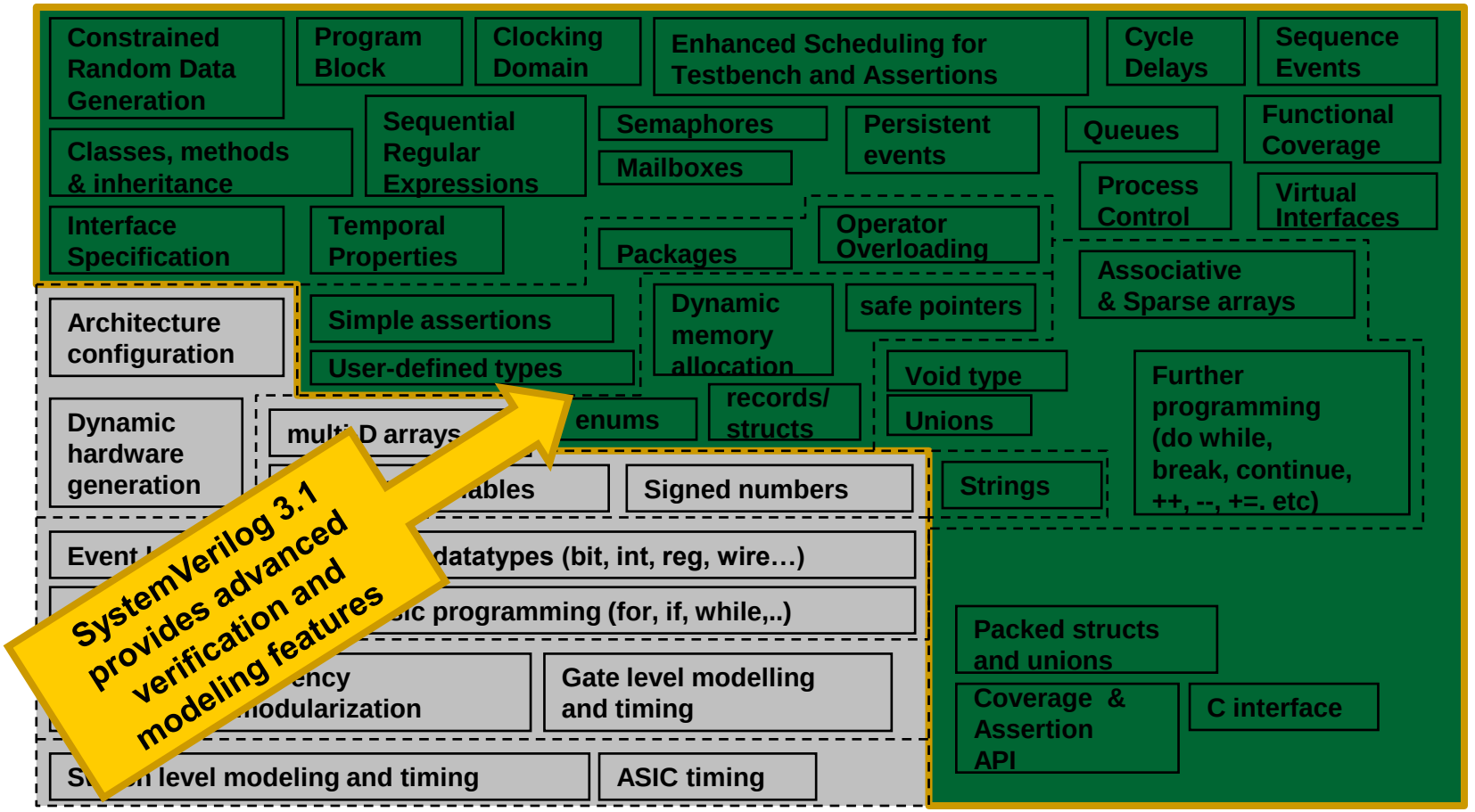
Semantic Concepts: C



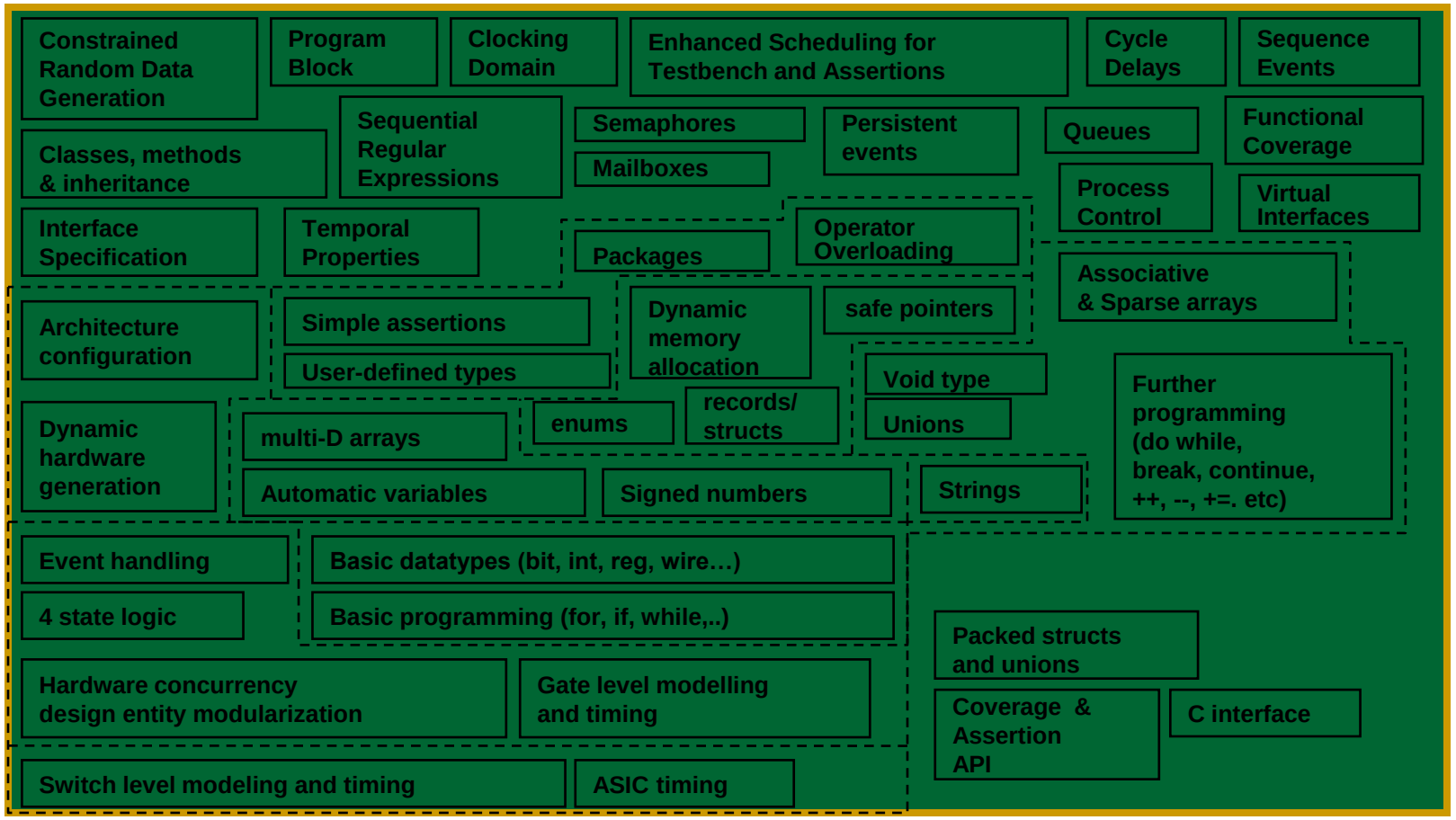
SystemVerilog: Verilog-2001



SystemVerilog: Enhancements



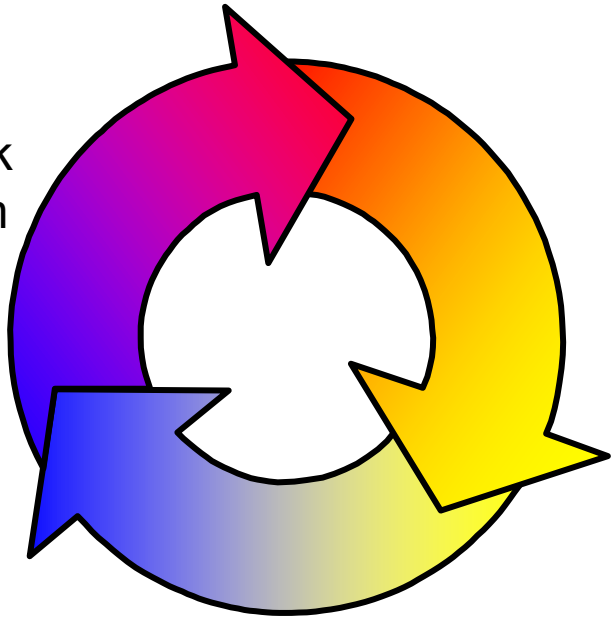
SystemVerilog: Unified Language



The Value of a Single Language

Unified Scheduling

- Basic Verilog won't work
- Ensures Pre/Post-Synth Consistency
- Enables Performance Optimizations



Knowledge of Other Language Features

- Testbench and Assertions
- Interfaces and Classes
- Sequences and Events

Reuse of Syntax/Concepts

- Sampling for assertions and clocking domains
- Method syntax
- Queues use common concat/array operations
- Constraints in classes and procedural code

Review Questions

- For verifying an individual module what is a good “reporting” strategy?
- What was the value of having a C-type model of the chip?
- Are functional vectors alone sufficient?
- What is essential for system-level verification?
- When is fork-join used?