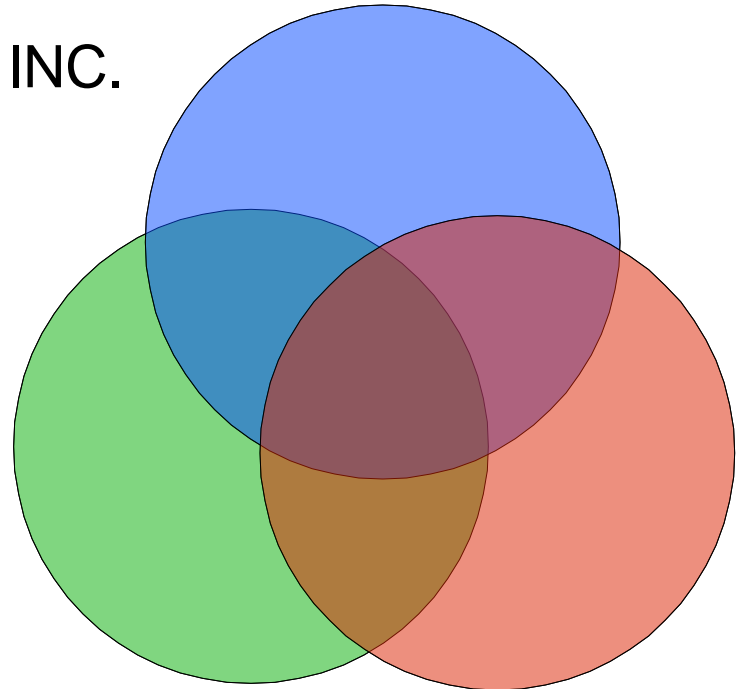


A Scalable Methodology for Analog & Mixed Signal Verification

Shyam Rapaka
Tapan Halder
SYNOPSYS INC.



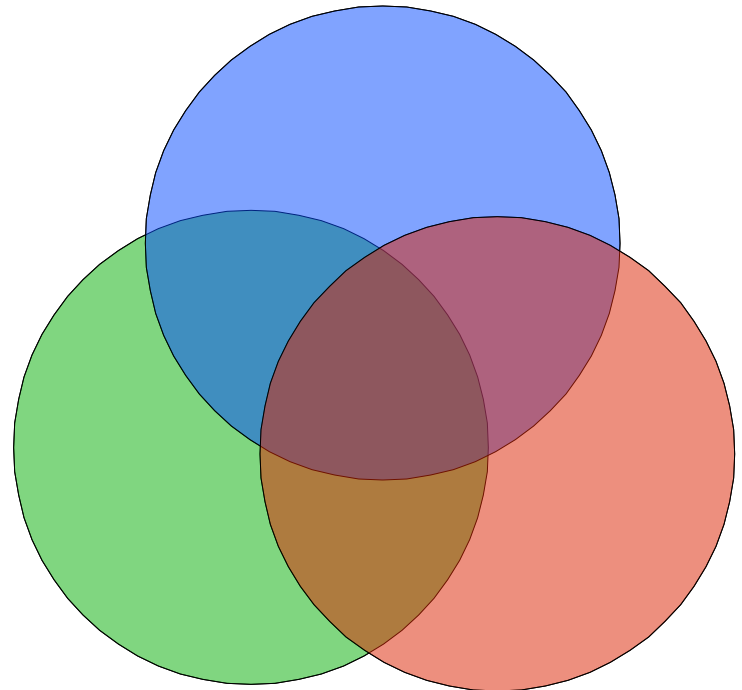
Outline

- ⦿ Segment 1
 - ⦿ Mixed Signal Verification
 - ⦿ Verification fundamentals
- ⦿ Segment 2
 - ⦿ Verilog recap
 - ⦿ Verilog-AMS basics
- ⦿ Segment 3
 - ⦿ SystemVerilog approach
 - ⦿ Advanced techniques

Segment 1

Mixed Signal Verification

Verification Fundamentals



Mixed Signal Verification

- Significance
 - ⊙ Mixed signal designs are becoming complex
 - ⊙ Time to market is shrinking
 - ⊙ Re-spins are expensive
- Issues
 - ⊙ Complex interaction between digital and analog blocks
 - ⊙ Simulation time can be in weeks
 - ⊙ Finding bugs late
- Solutions
 - ⊙ Unify the digital and verification flows
 - ⊙ Use high-level models for analog blocks
 - ⊙ A top down design and verification strategy

Goals

- Simulation
 - ⊙ Run full design simulations
 - ⊙ Test all configurations
 - ⊙ Blocks can be in Verilog-AMS or SPICE
- Find bugs upfront
 - ⊙ Connectivity errors
 - ⊙ Bus order errors
 - ⊙ Logic errors
- Unified methodology
 - ⊙ All the teams use the same flows
 - ⊙ Increase inter-operability
 - ⊙ Reuse of design and verification environment

Verification Challenges

- Problems facing design teams today
 - ⊙ Unpredictable design and verification schedules
 - ⊙ Concurrent execution bugs introduced by multi-core designs
 - ⊙ Functional bug escapes into production designs
 - ⊙ Inefficient application of design and verification resources
- Solutions
 - ⊙ Good design specifications
 - ⊙ Rigorous verification planning
 - ⊙ Automated verification management
 - ⊙ Live, executable verification plan

What is Functional Verification?

- **ver-i-fi-ca-tion** *n.* 1. *The process of demonstrating the intent of a design is preserved in its implementation.*
- Demonstrate
 - ⊙ Illustrate equivalent behavior
 - ⊙ Prove property conformance
- Intent
 - ⊙ Various abstractions of purpose, aim or objective
- Implementation
 - ⊙ Design-under-verification
 - ⊙ Flawed until proven otherwise

Design and Verification Cycles

- Write functional specification
- Write verification plan
- Design and implement DUV and verification environment
- Verify DUV
- Analyze results and adapt environment

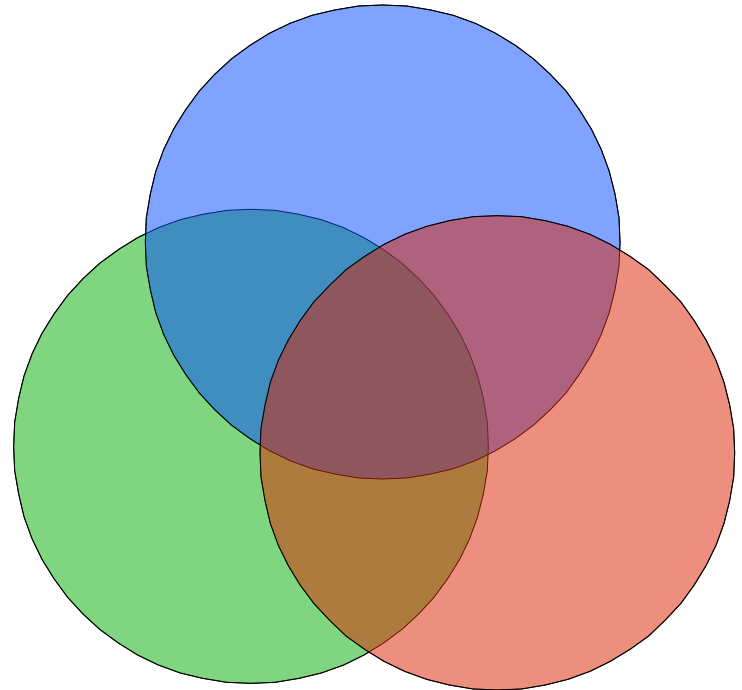
Dynamic Verification: Simulation

- Explore DUV behavioral space through execution
- Simulation, acceleration and emulation
- Stimulus generation
 - Legal, useful input data and sequential patterns
 - Constrained random generation, directed tests
- Response checking
 - Does DUV behavior fully conform to specification?
 - Reference model, assertions, behavioral rules, manual checking
- Coverage measurement and analysis
 - Has the DUV been fully exercised?
 - Limited execution samples
 - Quantify behavioral spaces of DUV features
 - Code and functional coverage

Segment 2

Verilog Recap

Verilog-AMS Basics



Verilog Basics

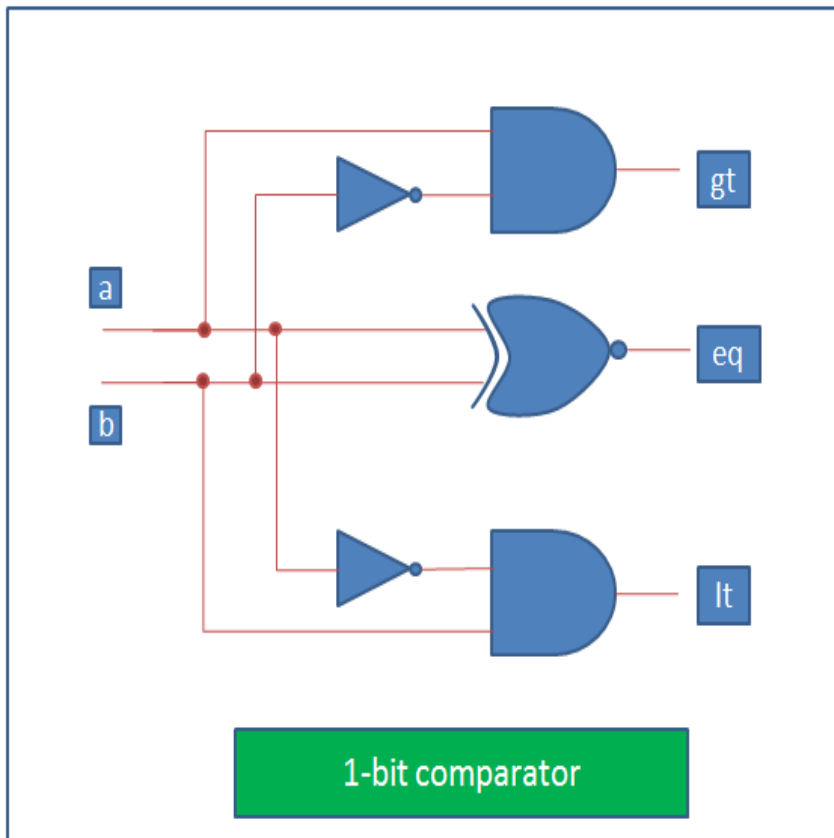
- Language features
 - ⊙ Data Types
 - ⊙ Behavioral modeling
 - ⊙ Hierarchy
- Design elements
 - ⊙ Combinational blocks
 - ⊙ Sequential blocks
 - ⊙ FSMs
- Simulation
 - ⊙ Stimulus
 - ⊙ Response checking
 - ⊙ Generating waveforms

Language Features

- Module
- Ports
- Expressions
 - Logical operators
 - Delays
- Assignments
 - Continuous assignments
 - Procedural assignments
- Behavioral modeling
 - Conditional statements
 - Looping statements
 - Block statements

```
module nand2(y, a, b);  
    output y;  
    input a, b;  
  
    parameter dly = 2;  
  
    assign #dly y = ~(a&b);  
  
endmodule
```

Combinational Blocks



```
module comp(gt, eq, lt, a, b);  
    output gt, eq, lt;  
    input a, b;  
    wire abar, bbar;  
  
    not I1(abar, a);  
    not I2(bbar, b);  
    and A1(gt, a, bbar);  
    and A2(lt, abar, b);  
    xnor X1(eq, a, b);  
endmodule
```

Simulation Basics

- Stimulus
 - ⊙ Instantiate DUT
 - ⊙ Generate test input
- Response checking
 - ⊙ Use \$display for output
 - ⊙ Avoid race conditions
- Generating waveforms
 - ⊙ Generate VPD output
 - ⊙ Use \$vcdpluson

```
module top;
  reg a, b;
  wire y1, y2;

  initial begin
    $vcdpluson();
    a = 0;
    b = 0;
    #10 a = 1;
    #10 b = 1;
    #10 a = 0;
    #10 b = 0;
    #10 $finish();
  end

  nand2 #(.dly(3)) I1(y1, a, b);
  nand2 #(.dly(4)) I2(.a(a), .b(b), .y(y2));

  always @(a or b or y1 or y2) begin
    $display($time,,
      "a = %b, b = %b, y1 = %b, y2 = %b",
      a, b, y1, y2);
  end

endmodule
```

Analog Designs

- Cannot synthesize analog designs
- Custom designs
- High performance – less margin for error
- Bottom-up vs. Top-down
- Standalone verification
- System verification

Verilog-AMS

- Nature
 - ⊙ Potential or Flow
- Disciplines
 - ⊙ Associated with a net
- Analog Operators
 - ⊙ Accessing voltages and currents
- Analog Block
 - ⊙ Analog behavior modeling

Nature

- Attributes
 - ⊙ abstol
 - ⊙ access
 - ⊙ idt_nature
 - ⊙ ddt_nature
 - ⊙ units
 - ⊙ user defined attributes

Example

```
nature current  
units = "A";  
access = I;  
idt_nature = charge;  
abstol = 1u;  
endnature
```

Disciplines

- Property of a net
- Domains
 - ⊙ Continuous (default)
 - ⊙ Discrete
- Natures
 - ⊙ Potential
 - ⊙ Flow
- Empty discipline
 - ⊙ No nature bindings

Examples

```
discipline electrical  
domain continuous;  
potential Voltage;  
flow Current;  
enddiscipline
```

```
discipline ddiscrete  
domain discrete;  
enddiscipline
```

Disciplines Contd.

- Net disciplines
 - ⊙ Disciplines for nets
 - ⊙ Scalar and vector nets
 - ⊙ Different systems
 - Conservative
 - Signal flow
 - Discrete
- Default discipline
 - ⊙ `default_discipline

Examples

```
electrical e1;  
electrical [MSB:LSB] n1;  
voltage [5:0] n2, n3;  
magnetic inductor;  
ddiscrete [10:1] connector1;  
wire w1;
```

Wreal net

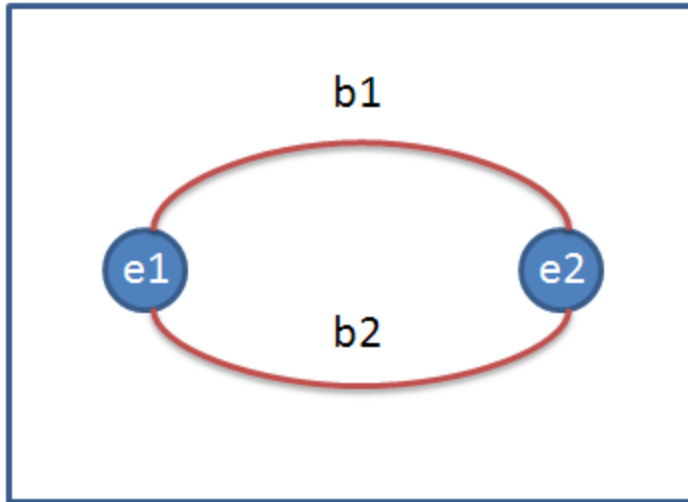
- **Net** of type real
- Either voltage **or** current
- Can be a module port
- Continuous assignment
- Single driver allowed
- Initialized to 0.0
- Always in digital domain

Example

```
module top();  
  real stim;  
  electrical load;  
  wreal wrstim;  
  assign wrstim = stim;  
  foo f1(wrstim, load);  
  always begin  
    #1 stim = stim + 0.1;  
  end  
endmodule
```

Branches

- Branch
 - ⊙ Path between two nets
 - ⊙ Conservative vs. Signal-flow
 - ⊙ Branch terminals
 - ⊙ Vector branch



Example

electrical e1;

electrical e2;

branch (e1, e2) b1, b2;

Analog Block

- Encapsulates analog behavior
 - ⊙ Continuous-time behavioral description
 - ⊙ All statements executed sequentially
 - ⊙ One per module
- Initialization
 - ⊙ @initial_step
 - ⊙ During simulation initialization
 - ⊙ Initialize analog owned variables
- Block statements
 - ⊙ Sequential blocks
 - ⊙ Block names

Example

```
module example;
  parameter integer p1 = 1;
  real r1, r2;
  analog begin
    @(initial_step) begin
      r1 = 2.5;
    end
  begin: myscope
    parameter real p2 = p1;
    real localVar = 1.5 * p2;
    r2 = localVar;
  end
end
endmodule
```

Contribution Statements

- Branch contribution operator <+
 - ⊙ Mathematical relationship
 - ⊙ Between one or more nets
 - ⊙ Left: signal access function
 - ⊙ Right: analog expression
- Relations
 - ⊙ Always on a branch
 - ⊙ Default second net: ground
 - ⊙ Each net has two quantities
 - Potential
 - Flow

Example

```
module resistor(p, n);  
  inout p, n;  
  electrical p, n;  
  branch (p,n) path;  
  parameter real r = 100.0;  
  analog  
    // V(p) <+ r*I(p);  
    // V(p,n) <+ r*I(p,n);  
    V(path) <+ r*I(path);  
endmodule
```

RLC Circuits

$$v(t) = Ri(t) + L\frac{d}{dt}i(t) + \frac{1}{C}\int_{-\infty}^t i(\tau)d\tau$$

$$V(p, n) \leftarrow R \cdot I(p, n) + L \cdot \text{ddt}(I(p, n)) + \text{idt}(I(p, n)) / C;$$

$$i(t) = \frac{v(t)}{R} + C\frac{d}{dt}v(t) + \frac{1}{L}\int_{-\infty}^t v(\tau)d\tau$$

$$I(p, n) \leftarrow V(p, n) / R + C \cdot \text{ddt}(V(p, n)) + \text{idt}(V(p, n)) / L;$$

Cross Statement

```
cross ( expr [ , dir [ , time_tol [ , expr_tol [ , enable ] ] ] ] )
```

```
analog begin
```

```
  @(cross(V(smpl) - thresh, dir))
```

```
    state = V(in);
```

```
  V(out) <+ transition(state, 0, 10n);
```

```
end
```

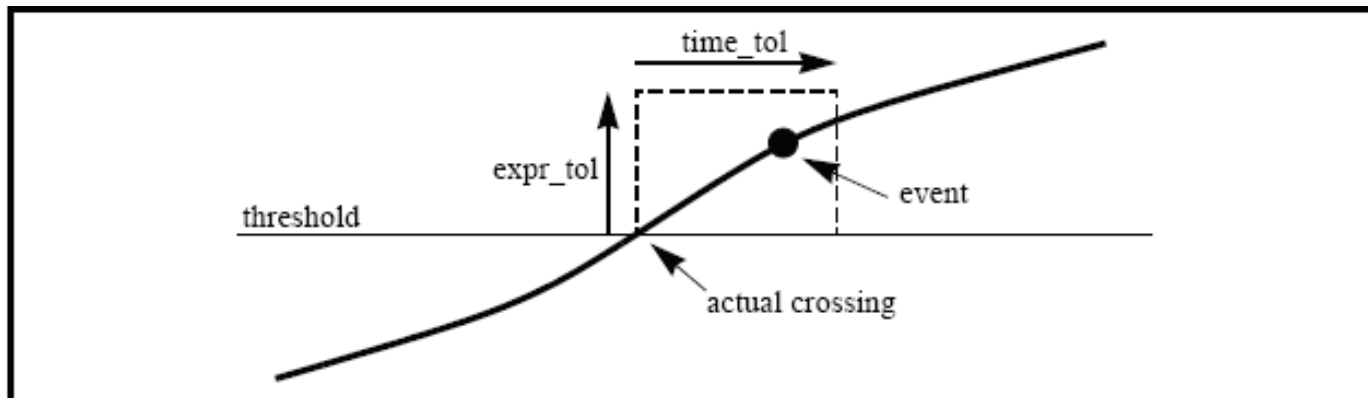


Figure 5-6: Timing of event relative to threshold crossing.

Source: Verilog-AMS LRM

Above Statement

```
above ( expr [ , time_tol [ , expr_tol [ , enable ] ] ] )
```

```
// Same as cross, except triggers at time 0
```

```
module sh (in, out, smpl);  
output out;  
input in, smpl;  
electrical in, out, smpl;  
real state;  
  
analog begin  
    @(above(V(smpl) - 2.5))  
        state = V(in);  
    V(out) <+ transition(state, 0, 10n);  
end  
  
endmodule
```

Accessing Events

Discrete events in continuous context

```
analog begin
  @(posedge clk1 or cross(V(clk2), 1))
  vout = V(in);
  V(out) <+ vout;
end
```

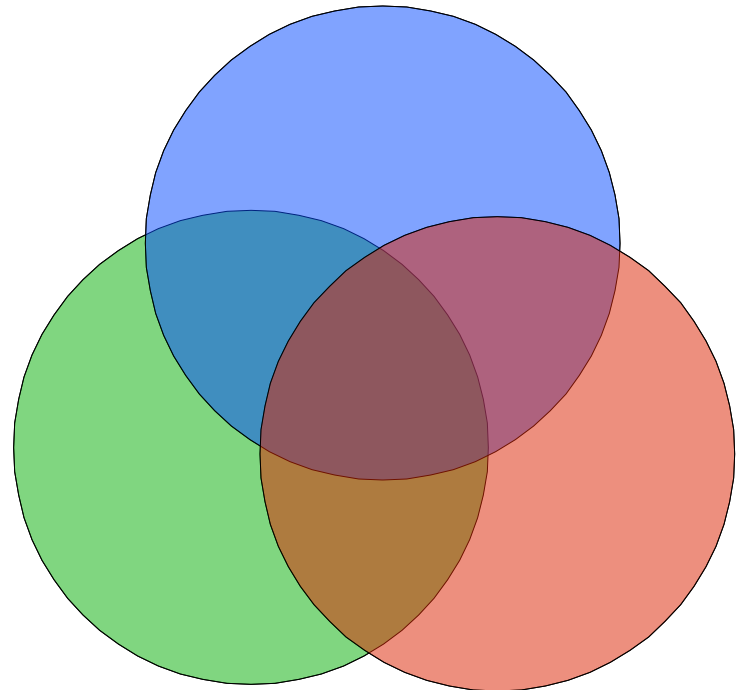
Continuous events in discrete context

```
always @(cross(V(clk) - 2.5, 1))
  out = in;
```

Segment 3

SystemVerilog Approach

Advanced Techniques

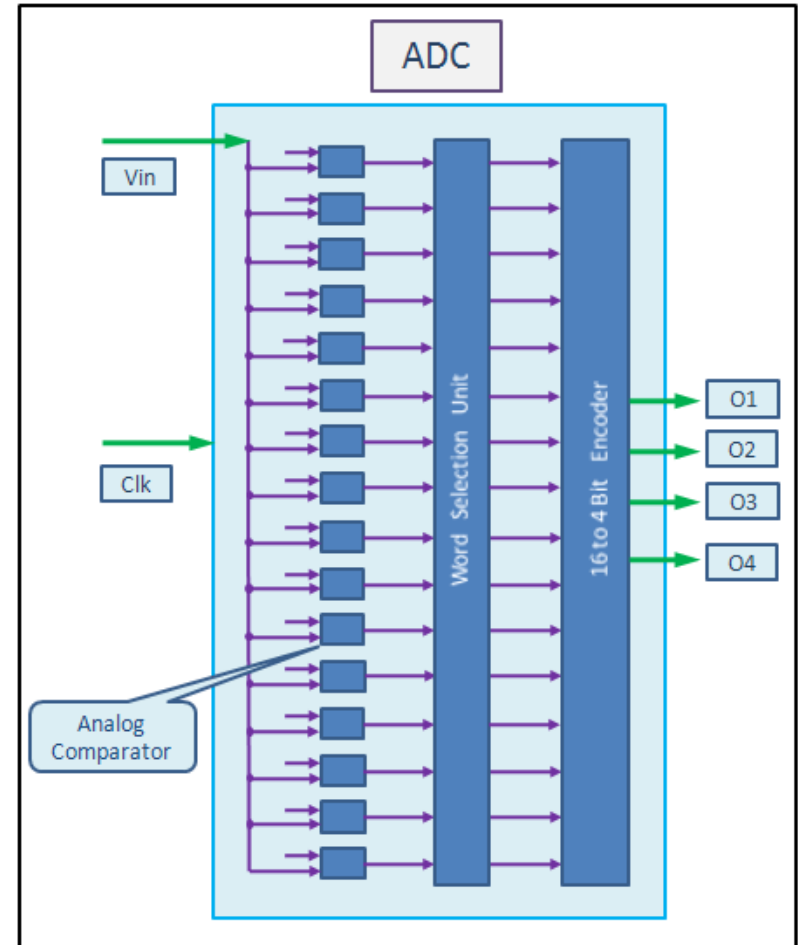


Case Study: ADC

- Design Details
- SystemVerilog Approach
- Checker Modules

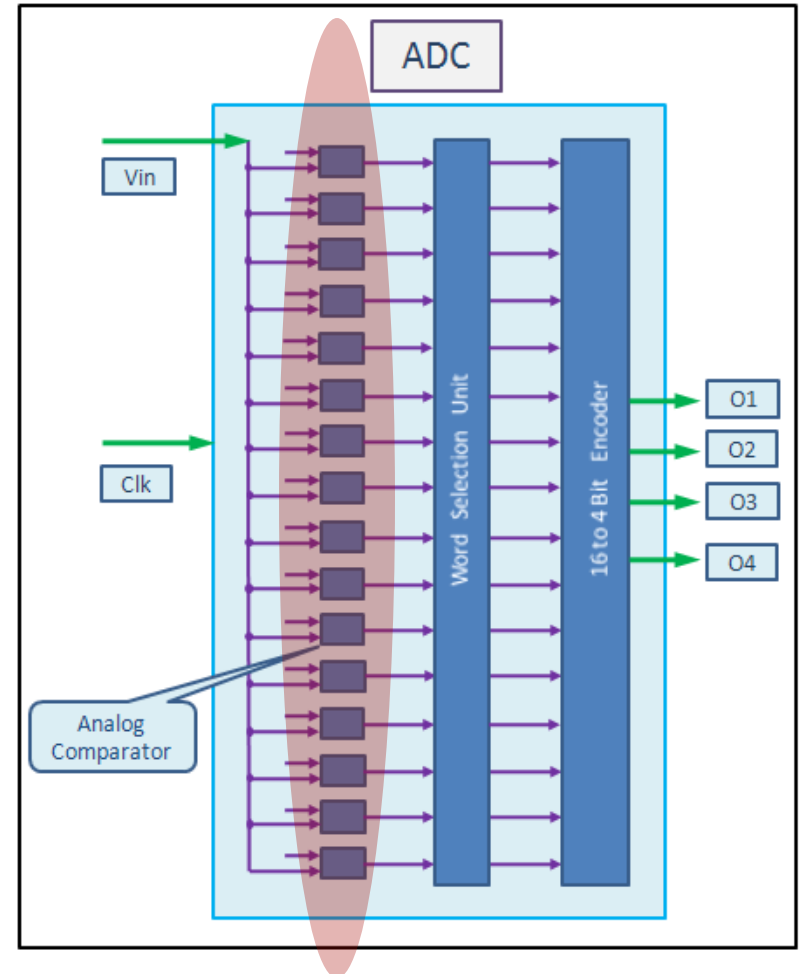
Design Under Verification

- Analog Input
 - V_{in}
- 4-Bit Digital Output
 - O4, O3, O2, O1
- Input Range
 - 0.0V to 1.8V



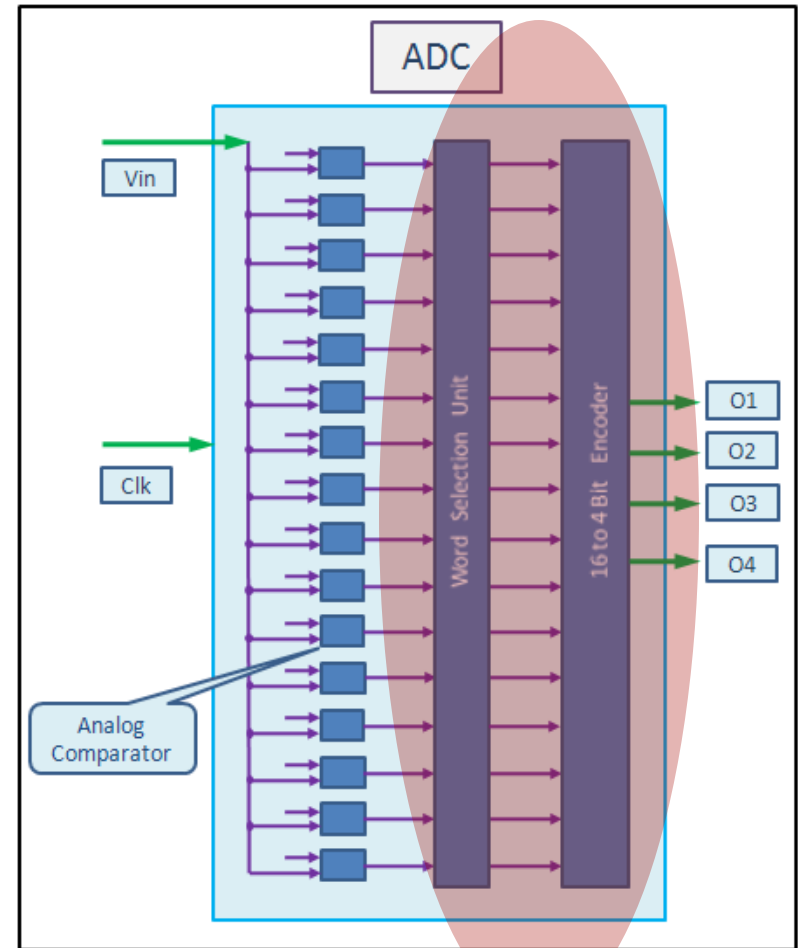
Analog Comparator

- Two Analog Inputs
 - V_{in} & V_{ref}
- Digital Output
 - cmpOut
 - Output '1' when $V_{in} > V_{ref}$
- Array of Comparators
- Comparator Error



Output Generation

- Word Selection Unit
 - 16-Bit Input
 - Comparator Outputs
 - 16-Bit Output
 - One Hot Encoding
- 16-Bit to 4-Bit Encoder
 - Generates final Output

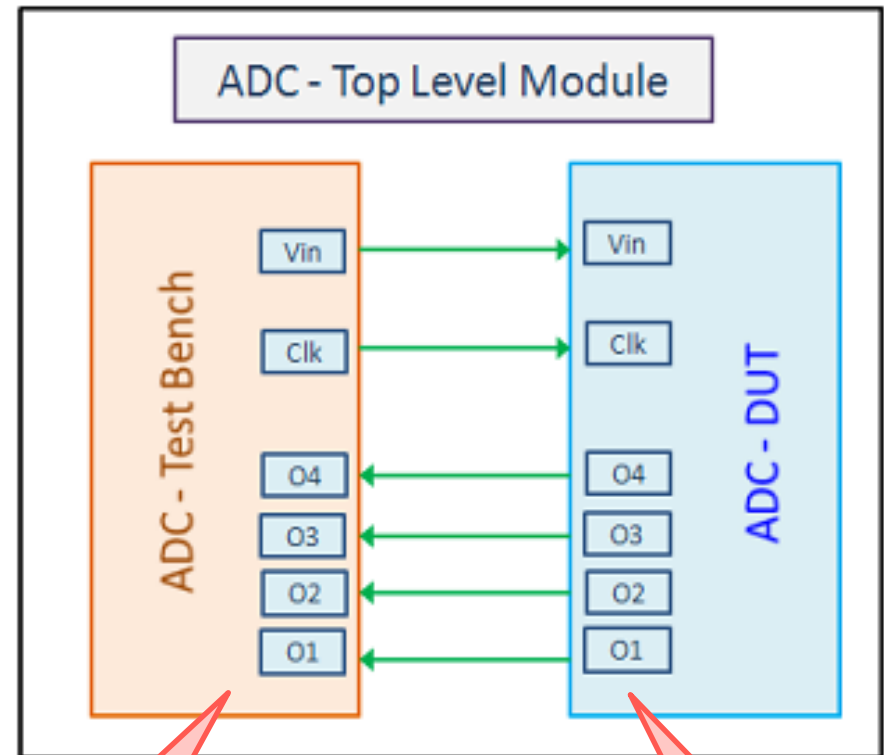


SystemVerilog Approach

- SystemVerilog–SPICE mixed signal verification
 - ⊙ Pure SPICE DUT
 - ⊙ SystemVerilog modules
- Analog/Digital Interface
 - ⊙ Wire Ports
 - ⊙ Real Ports
- Advanced Features
 - ⊙ Classes
 - ⊙ Assertions
 - ⊙ Packages

Testbench

- Advantages
 - Decoupled from design
 - Encapsulate behavior
 - Complex test cases
 - Reuse packages
 - Easy to develop
 - Easy to analyze
 - Easy to maintain



System
Verilog

SPICE

Stimulus Generation

- Simplified transitions
- Easy to program
- Complex functions
- SystemVerilog Classes
 - ⊙ Data Hiding
 - ⊙ Methods
 - ⊙ Inheritance
- Packages
 - ⊙ Code reuse

```
•initial begin
•      Vin = 0.7;
•      #23 Vin = 0.4;
•      #20 Vin = 0.83;
•      #20 Vin = 1.1;
•      #20 Vin = 1.5;
•      #20 Vin = 1.7;
•      #20 Vin = 1.14;
•      #20 Vin = 0.1;
•      #20 Vin = 0.5;
•      #20 Vin = 0.85;
•end
```

SystemVerilog Package Example

```
package A2DPkg;

function logic real2logic(real r);
begin if(r > 0.9)
    real2logic = 1'b1;
else
    real2logic = 1'b0;
end
endfunction
```

```
class a2dMonitor;
    local logic [3:0] a2dVal;
    real vinRec;
```

```
function new();
    a2dVal = 0;
    vinRec = 0.0;
endfunction
```

```
task setVin(real v);
    vinRec = v;
endtask
```

```
function logic[3:0] getExpectedVal();
    if (vinRec < 0.1125)
        getExpectedVal = 4'b0000;
    .....
else
    getExpectedVal = 4'b1111;
endfunction
```

```
task verify( real r1, real r2,
            real r3, real r4);
```

```
    logic [3:0] expVal;
    a2dVal[0] = real2logic(r1);
    a2dVal[1] = real2logic(r2);
    a2dVal[2] = real2logic(r3);
    a2dVal[3] = real2logic(r4);
```

```
    expVal = getExpectedVal();
    if (a2dVal == expVal)
        $display($time,,, "PASSES");
    else
        $display($time,,, "FAILS");
```

```
endtask
```

```
endclass
```

```
endpackage : A2DPkg
```

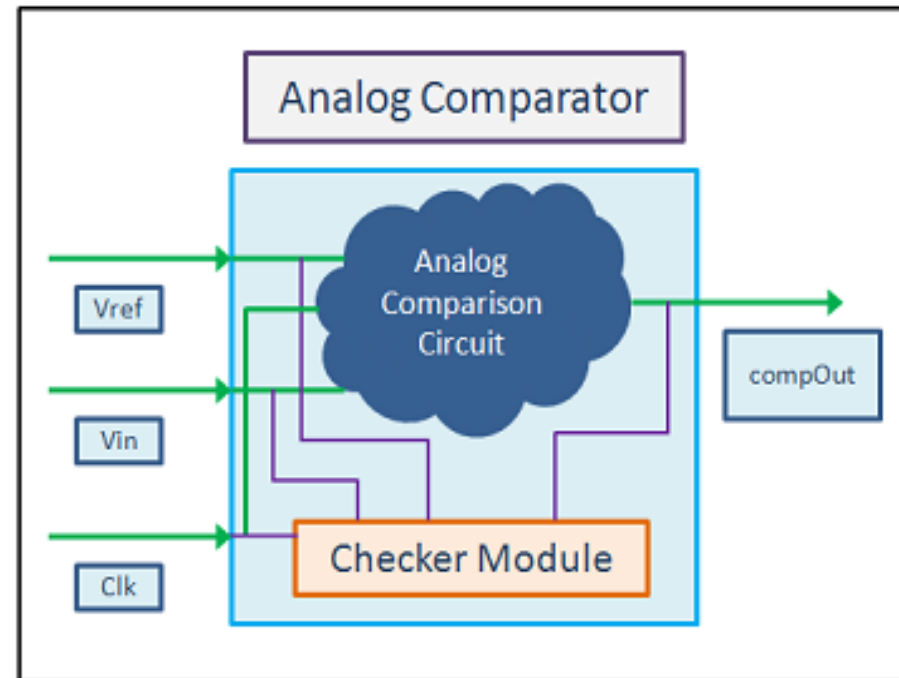
Output Verification

- Encapsulate behavior
- Complex I/O relationship
- Sampling Inputs
 - ⦿ System or internal clock
- Sampling Outputs
 - ⦿ Internal clock
- No waveform analysis
- Regression suites

```
A2DPkg::a2dMonitor am1 = new;  
  
always @(posedge intClk) begin  
    am1.setVin(Vin);  
end  
  
always @(posedge anaSampleClk)  
begin  
  
    if ($time > 30) begin  
        am1.verify(O1, O2, O3, O4);  
    end  
end
```

Checker Modules

- Ensure expected behavior
- Inserted in SPICE
- Using Assertions
- Debugging Circuits
- Sub-circuit development
- Complete design testing
- Identifying Bugs
- Terminating Simulation



Checker Module Example

```
module anaComparatorChecker(Vin, Vref,
    cmpOut, Clk);
    input Vin, Vref, cmpOut, Clk;
    wreal Vin, Vref, cmpOut, Clk;
    logic intClk; wire anaSampleClk;

    initial intClk = 0;
    always @(Clk) begin
        intClk = A2DPkg::real2logic(Clk);
    end

    assign #1 anaSampleClk = intClk;
```

```
always @(posedge anaSampleClk) begin
    if ($time > 30) begin
        assert (((cmpOut > 0.9) && (Vin >
            Vref)) || ((cmpOut < 0.9) && (Vref >
            Vin)))
            `ifdef DISPLAY_PASSES
                $display($time,,, "PASSES");
            `endif
        else
            begin
                $display($time,,, "Analog
                    Comparator (%m) Malfunction: Vin=%g,
                    Vref=%g, cmpOut=%g\n",
                    Vin, Vref, cmpOut);
            end
        end
    end
end

endmodule
```

Wreal Modeling

- High level abstraction
 - ⊙ Wreal nets are nets with real values
 - ⊙ Can be used to represent either current or voltage
 - ⊙ Very useful for system level verification
- Ease of modeling
 - ⊙ No contribution statements required
 - ⊙ Simple translation of functional specification
 - ⊙ Accuracy controlled by user through models
- Simulation speed
 - ⊙ Pure digital construct
 - ⊙ Analog simulator might not be invoked
 - ⊙ Many orders of speedup

Verilog-AMS vs. Wreal

Verilog-AMS

```
module top();  
  real stim;  
  electrical evin;  
  always begin  
    #1 stim = stim + 0.1;  
  end  
  analog begin  
    V(evin) <+ stim;  
  end  
endmodule
```

Wreal

```
module top();  
  real stim;  
  wreal wvin;  
  always begin  
    #1 stim = stim + 0.1;  
  end  
  assign wvin = stim;  
endmodule
```

Using SPICE

- SPICE models
 - ⊙ Described in different (.spi) files
 - ⊙ Require corresponding technology (.mod) files
- Hierarchy
 - ⊙ SPICE sub-circuits can be instantiated in Verilog-AMS files
 - ⊙ SPICE (sub-)circuits can instantiate Verilog-AMS modules
- Control file
 - ⊙ Specify the SPICE file to be used
 - ⊙ Specify the SPICE sub-circuits to be used
- Ports
 - ⊙ Default directions are INOUT
 - ⊙ Disciplines are always electrical

Use Model

SPICE sub-circuit

```
.include "cmos35t.mod"
```

```
.subckt inv_sp in out
```

```
mp1 out in vdd vdd p
```

```
w=20.00u l=0.35u
```

```
mn1 out in gnd gnd n
```

```
w=10.00u l=0.35u
```

```
.ends
```

```
.global vdd gnd
```

```
vsu vdd gnd 3.3
```

```
.end
```

Control file (.init)

```
use_spice -cell inv_sp;
```

```
choose nanosim -n inv.spi -c svams.cfg ;
```

Verilog-AMS

```
module mid(in, out);
```

```
    input in; output out;
```

```
    inv_sp I1(in, out);
```

```
endmodule;
```

Reference Modeling

- Reference models
 - For SPICE sub-circuits
 - Used as a specification for developing SPICE models
 - Used as a substitute for SPICE in system level verification
- Abstraction
 - Verilog – for digital sub-circuits
 - Verilog-AMS – for mixed-signal/analog sub-circuits
 - Wreal – for mixed-signal/analog sub-circuits
- When to develop
 - At the beginning of the design cycle for the top level blocks
 - Along with the SPICE sub-circuits for the components
 - After the SPICE sub-circuits to check for specific features
 - Start with a simple model
 - Enhance through the design cycle

Significance of Reference Models

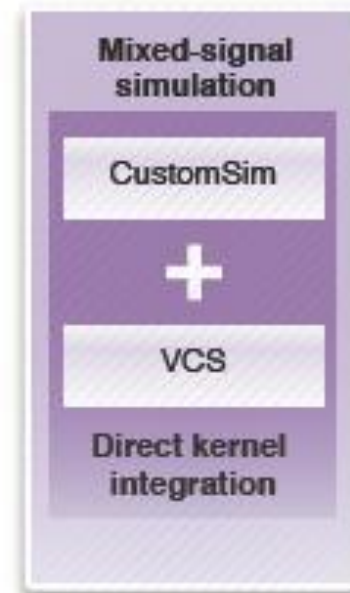
- Mixed Signal Design
 - ⊙ Make sure everyone follows the same specification
 - ⊙ Catch design errors early
 - ⊙ Designers know exactly how the adjacent blocks behave
- Mixed Signal Verification
 - ⊙ Run full-chip simulations
 - ⊙ Regressions for different configurations
 - ⊙ Control what is printed through different macros
- Debugging
 - ⊙ What went wrong
 - ⊙ Which signal and instance were involved
 - ⊙ At what time-points did the failure occur
 - ⊙ Easier than measuring waveforms
 - ⊙ Point out who has to fix the issue!

Summary

- Verilog/SystemVerilog
- Verilog-AMS
- Wreal Modeling
- Checker Modules
- Reference Models
- Reduction in Human Error
- Improved Productivity

Information

- Synopsys Discovery-AMS Platform
 - ⦿ VCS – Digital Simulator
 - ⦿ CustomSim – FastSPICE solutions
 - HSIM, Nanosim and XA
- More information
 - ⦿ www.synopsys.com



THANK YOU!