SYNOPSYS®

Digital Design With Verilog

Introduction

December 2023

CONFIDENTIAL INFORMATION

The information contained in this presentation is the confidential and proprietary information of Synopsys. You are not permitted to disseminate or use any of the information provided to you in this presentation outside of Synopsys without prior written authorization.

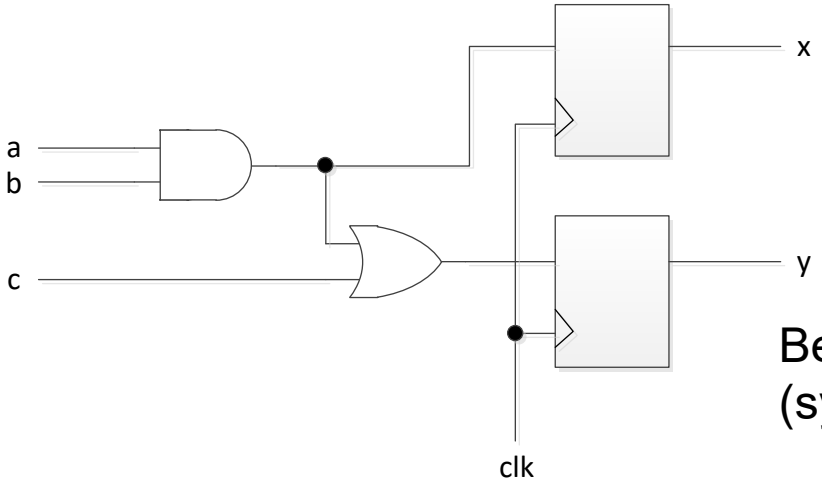
IMPORTANT NOTICE

In the event information in this presentation reflects Synopsys' future plans, such plans are as of the date of this presentation and are subject to change. Synopsys is not obligated to update this presentation or develop the products with the features and functionality discussed in this presentation. Additionally, Synopsys' services and products may only be offered and purchased pursuant to an authorized quote and purchase order or a mutually agreed upon written contract with Synopsys.

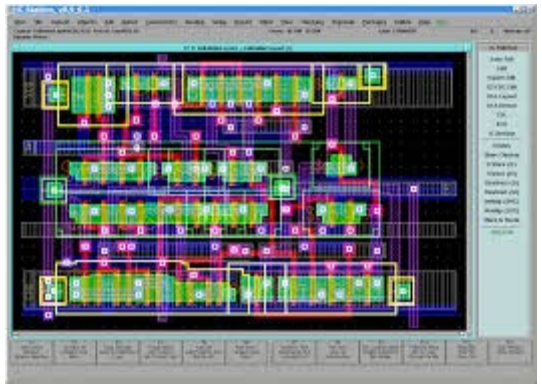
Agenda

- Overview
- Designing with Verilog
- Writing Testbenches
- VCS Setup and Use Model Information

Overview



Before 1990
(synthesis)



```
always @(posedge clk)
   begin
      x = b & c;
      y = a | x;
   end
```

HDL description

So we are going to talk about design

And how to use an HDL language to design?

What is HDL?

- Hard & Difficult Language?
  - No, means **H**ardware **D**escription **L**anguage
- High Level Language
  - To describe the circuits by syntax and sentences
  - As oppose to circuit described by schematics
  - Allows designers to model the **concurrency** of process found in hardware
- Widely used HDLs
  - Verilog – Similar to C
  - SystemVerilog – Similar to C++
  - VHDL – Similar to PASCAL/ADA

Purpose of HDLs

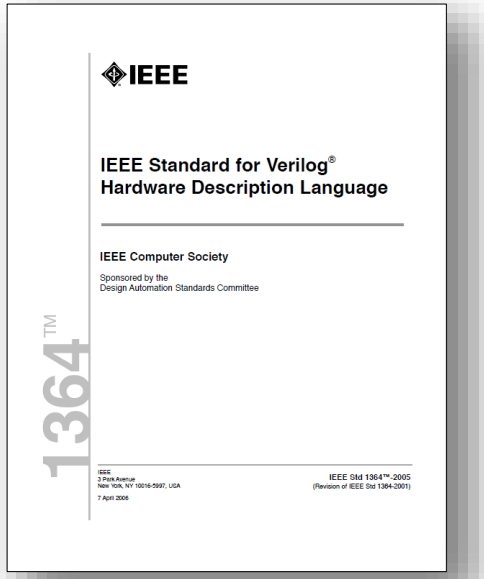
- Purpose of Hardware Description Languages
  - Capture **design** in RTL form
    - i.e. all registers specified
  - Use to simulate designs to verify correctness
  - Pass through synthesis tools to obtain reasonable optimal gate-level design that meets timing
  - Design productivity
    - Automatic synthesis
    - Capture design as RTL instead of schematic
    - Reduces time to create gate level design by an order of magnitude
- Synthesis
  - Basically, a Boolean combinational logic optimizer that is timing aware

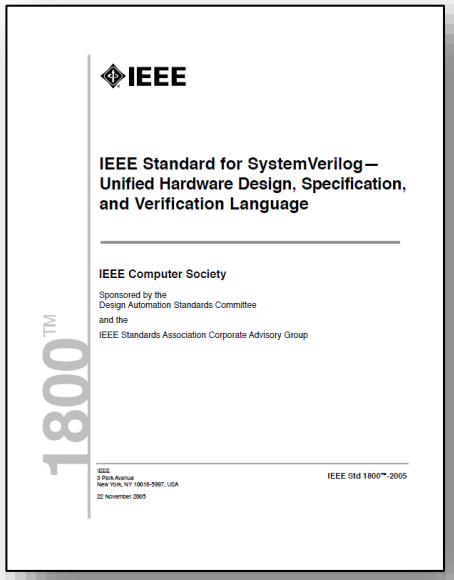
Verilog development history

1970:	• SILOS developed as an experimental simulation language for logic. Influenced a large amount of later solutions
1984:	• Verilog launched as a commercial product by Gateway Design Automation as a proprietary language for logic simulation
1989:	• Gateway was acquired by Cadence
1990:	• Verilog was made an open standard under the control of OpenVerilog International
1995:	• The language became an IEEE standard (IEEE 1364) and was updated in 2001 and 2005
2002:	• SystemVerilog released by Accelera, based on Superlog extension and inspired by the OpenVera language capabilities
2005:	• SystemVerilog adopted as an IEEE standard (IEEE 1800)
2012:	• SystemVerilog (IEEE 1800-2012) Released. Updated Verification Capabilities

IEEE-1364 / IEEE-1800

Verilog 2005 (IEEE Standard 1364-2005) consists of minor corrections, spec clarifications, and a few new language features





SystemVerilog is a superset of Verilog-2005, with many new features and capabilities to aid design-verification and design-modeling

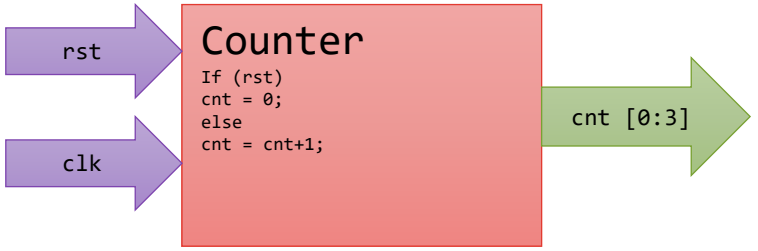
Simulation and Synthesis

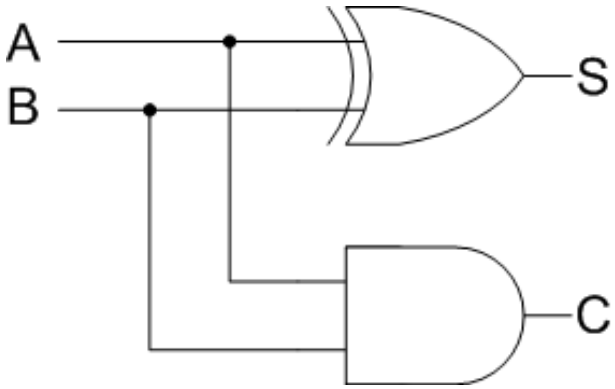
- The two major purposes of HDLs are logic simulation and synthesis
  - During simulation, inputs are applied to a module, and the outputs are checked to verify that the module operates correctly
  - During synthesis, the textual description of a module is transformed into logic gates
- Circuit descriptions in HDL resemble code in a programming language. But the code is intended to represent hardware
- Not all of the Verilog commands can be synthesized into hardware

- Our primary interest is to build hardware, we will emphasize a synthesizable subset of the language
- Will divide HDL code into synthesizable modules and test bench (simulation)
  - The synthesizable modules describe the hardware.
  - The test bench checks whether the output results are correct (only for simulation and cannot be synthesized)

Types of modeling

- Behavioral
  - Models describe what a module does.
  - Use of assignment statements, loops, if, else kind of statements
  - Related to the algorithm or behavior of the circuit (algorithmic modeling)
  - Highest level of abstraction
  - Module implemented in terms of the desired algorithm without concern of the hardware
  - Specifies the circuit in terms of expected behavior
  - Closest to natural language description, difficult to synthesize
- Structural (gate level)
  - Describes the structure of the hardware components
  - Interconnections of primitive gates (AND, OR, NAND, NOR, etc.) and other modules





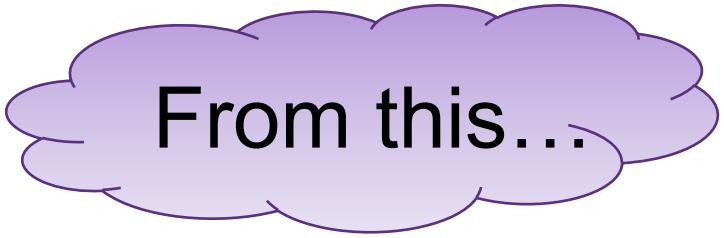
Behavioral - Structural

Behavioral

```
module cter (
   input  rst, clock,
   output reg  [1:0] count);

   always@(posedge clock)
      begin
         if (rst) count = 0;
         else     count = count+1;
      end
endmodule
```

Structural

```
module cter ( rst, clock, count );
   output [1:0] count;
   input rst, clock;
   wire   N5, n1, n4, n5, n6;
   FD1  U0 (.D(N5), .CP(clock),
            .Q(count[0]), .QN(n6));
   FD1  U1 (.D(n1), .CP(clock),
           .Q(count[1]), .QN(n5));
  MUX21 U2 (.A(N5), .B(n4),
            .S(n5), .Z(n1) );
  NR2   U3  (.A(n6), .B(rst),
             .Z(n4));
  NR2   U4  (.A(count[0]), .B(rst),
             .Z(N5));
endmodule
```

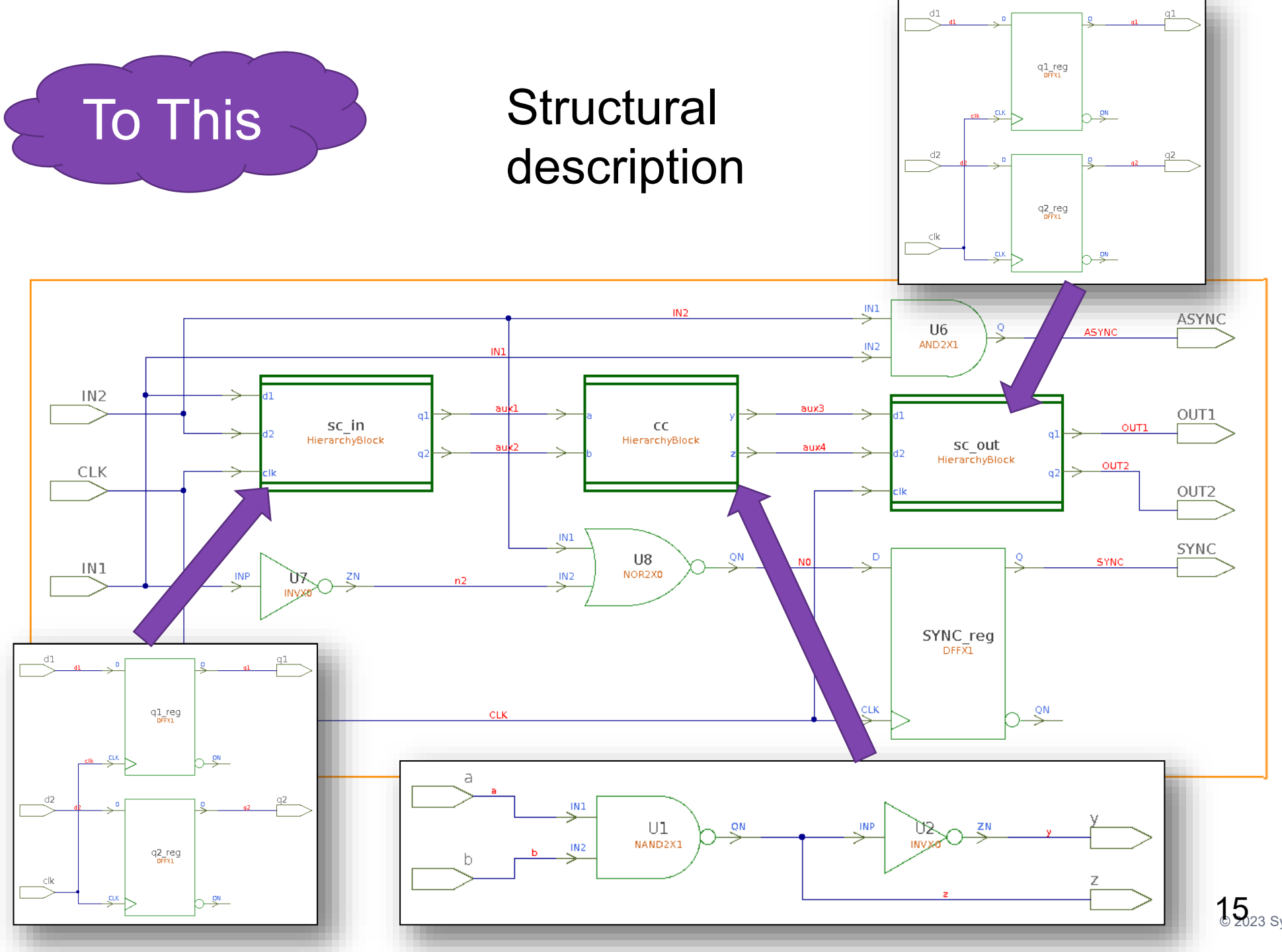



```
module comb_cell (input a,b, output y,z);
  assign y = a && b;
  assign z = ! y ;
endmodule

module seq_cell (input d1,d2, clk, output reg q1,q2);
  always @( posedge clk)
  begin
      q1 <= d1;
      q2 <= d2;
  end
endmodule

module top (input IN1,IN2,CLK, output OUT1, OUT2, ASYNC, output reg SYNC);
  wire aux1,aux2,aux3,aux4;

  // continous assignment
  assign ASYNC = IN1 && IN2 ;

  //procedural block
  always @ (posedge CLK)
  begin
    SYNC = IN1 && ( ! IN2 ) ;
  end

  //instantiation
  seq_cell sc_in (.clk(CLK), .d1(IN1), .d2(IN2), .q1(aux1), .q2(aux2) );
  comb_cell cc (.a(aux1), .b(aux2), .y(aux3), .z(aux4));
  seq_cell sc_out (.clk(CLK), .d1(aux3), .d2(aux4), .q1(OUT1), .q2(OUT2) );
endmodule
```

Behavioral description

To This

Structural description



15

Other Types of Modeling

- Data-Flow Modeling
  - Based on logic equations and usage of *assign*
  - Specification made by assigning expressions on input signals
- Switch Level Modeling
  - Consist of transistors, switches and buffers
  - Modules are described terms of transistors, switches, storage nodes, and interconnections
  - Geared for post implementation or interface simulation (digital to analog interfaces, chip pins and output drivers)

Designing with Verilog

Design Mantras

- One clock, one edge, Flip-flops only
- Design BEFORE coding
- Behavior implies function
- Clearly separate control and datapath

Basic Verilog Elements

- Modules: Any design is contained in a module
  - Module is seen from the outer world through its interface
  - Content is conditioned by parameters
  - Can contain module instances
  - Can be thought as a page in a multi page hierarchical schematic
  - A file can contain one or more modules

```
`timescale 10ns/1ps
module test_bench;
// Interface to communicate with the DUT
logic a, b, clk;
logic c;
// Device under test instantiation
DUT U1 (.in1(a), .in2(b), .clk(clk), .out1(c));
initial
begin  // Test program
  test1 ();
  $finish;
end
initial
begin
  clk = 0;
  forever #5 clk = ~clk;
end
initial
begin // Monitor the simulation
  $display ("clk | in1| in2 | out1 |");
  $monitor ("  %b| %b |  %b |   %b |",clk, a, b, c);
end
endmodule
```

```
module DUT (in1, in2, clk, out1);
input in1, in2;
input clk;
output logic out1;
always @(posedge clk)
 out1 = in1^in2;
endmodule
```

Basic Verilog Elements

- Variables
  - Reg: Stores values from procedural statements
  - Wire: Connects elements continuously
  - Logic: Any of the above (adapts to the context)
  - Integer: Signed 32-bit value (or a machine word) for simulator or conditioning purposes
  - Time: machine word to track current simulation time
  - Arrays:
    - logic [<word>] variable [<pack/index>]

```
`timescale 10ns/1ps
module test
logic a, b, clk;
logic c;
logic d;
logic [31:0] memo [255:0];
…
…
```

Basic Verilog Elements

- Description Statement Forms
  - Procedural blocks
    - The whole block executes upon events defined in the sensitivity list.
    - This execution is concurrent with any other block or form that responds to the same defined events.
    - When executes, statements between begin-end execute in sequence.
    - Values can be taken from wire variables and stored into reg variables.
    - Usual procedural forms available (if-else, for, while, case)
    - Blocking and non blocking assignment evaluation.
  - Continuous Assignment Statements
    - A single expression that is evaluated at all events (hence concurrent with everything active at any time).
    - Represents connections
    - Allows multiple forcing elements
    - Can evaluate simple logic expressions and conditionals
    - Values can be taken from wire and ref, and assigned to other wires

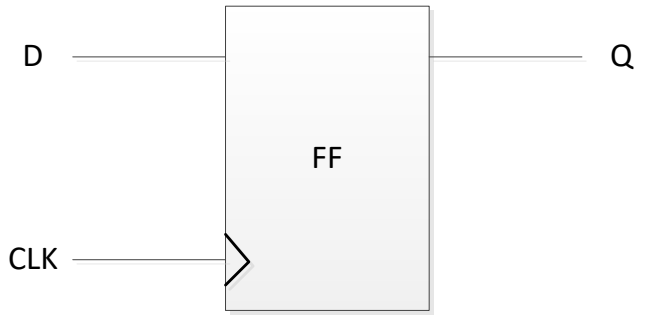
```
always @(a or b or c)
   if (a) out = b ^ c;
   else out = b | c;
```

```
assign out = a ?
b ^ c : b | c;
```

Basic Verilog Constructs

- Flip Flop
  - Behavior
    - Every time a non-sensible variable changes, and there is a sensible variable that later changes, if it is assigned to a non-sensible variable, a storage element is inferred in the assigned non-sensible variable.
  - Behavior implies function
    - Edge sensitivity will create a flip flop. Value sensitivity will create a latch.
  - Code behavior
    - Q re-evaluated every time there is a rising edge of the clock
    - Q remain unchanged between rising edges

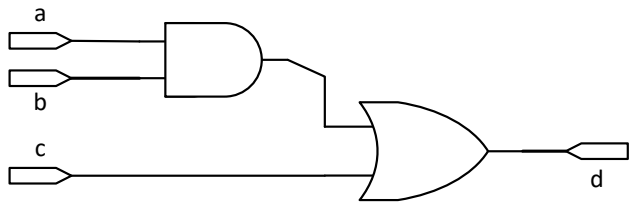
```
always_ff @(posedge clk)
   Q <= D;
```



Basic Verilog Constructs

- Logic Expressions
  - Behavior
    - Any activity on the input variables triggers evaluation
    - A logic expression is evaluated from those variables and assigned to a variable
  - Behavior implies function
    - An element sensible to any input can be considered “permanent” or continuous.
    - A combinational logic circuit that fits the expression will be generated
  - Code behavior
    - Any movement on a,b,c will trigger the always_comb block
    - d will be updated with the calculated value from the expression

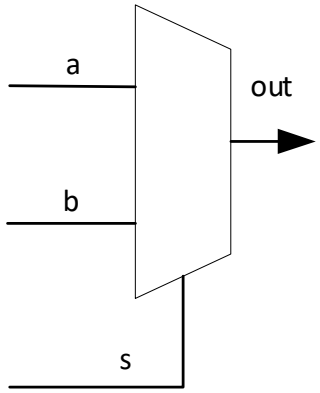
```
always_comb
   d = c ||(a && b);
```



Basic Verilog Constructs

- Selector
  - Behavior
    - Every time a sensible variable changes, evaluate this block.
    - If certain, say “a”, condition is met, evaluate corresponding branch “a”, if not, evaluate other
    - Every branch has its own logic expression, or another selection and must be ready to evaluate at any sensible event.
  - Behavior implies function
    - A multiplexer circuit is inferred: selects one or another circuit input and shows its value in the output.
    - To the inputs, a circuit that evaluate the branch expression is synthesized.
  - Code behavior

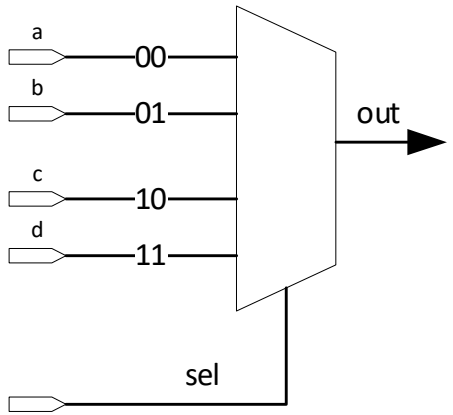
```
always_comb
begin
   if (s==1)
      out=b;
      else
      out=a;

endif
end
```



Basic Verilog Constructs

- Selector (multi-branch)
  - Behavior
    - Every time a sensible variable changes, evaluate this block.
    - There is large set of conditions. Only one of those is evaluated upon matching.
    - Every branch has its own logic expression, or another selection and must be ready to evaluate at any sensible event.
  - Behavior implies function
    - A multiplexer circuit is inferred: selects one circuit input and shows its value in the output.
    - To the inputs, a circuit that evaluate the branch expression is synthesized.
    - Match is done upon the selecting variable, n-patterns will imply n-inputs
  - Code behavior

```
always_comb
begin
   case (sel)
    00:
        begin
            out=a
        end
    01:
        begin
            out=b
        end
    10:
        begin
            out=c
        end
    11:
        begin
            out=d
        end
end
```



More Examples

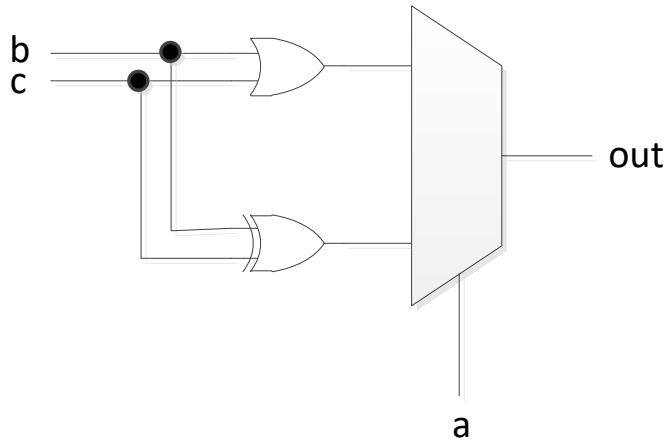
- What is the behavior and matching logic for this code fragment?

```
always @(clk or D)
   if (clock) Q <= D;
```

- What is the behavior and code for this schematic?

Combinational Logic

- Combinational logic example
- How would you describe the behavior of this function in words?



- And in code?

Other Procedural blocks

System Verilog adds the following procedural blocks:

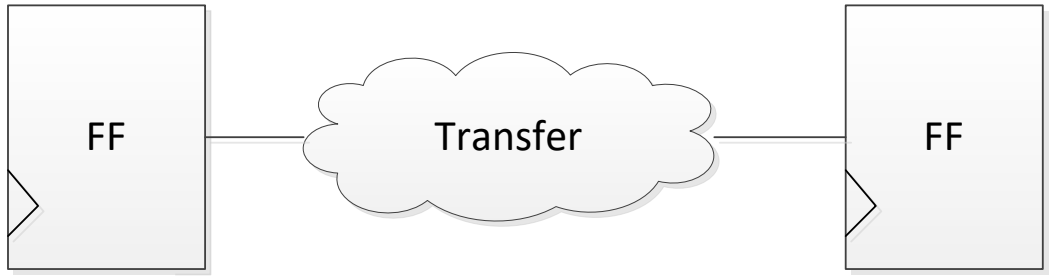
- `always_ff` Geared to infer synchronous registers. Tools can show warnings if no register is inferred
- `always_latch` Geared to infer Latches. Tools can show warnings if latch is not inferred
- `always_comb` Geared to infer combinational logic. Tools can show warnings if any storage element is inferred. Does not need sensitivity list.

Blocks targeted for simulation:

- `initial` Runs once at simulation start. Recommended for testbench initialization.

Design Before Coding

- Automatic synthesis does not relieve you of logic design
- It does relieve you of
  - Logic optimization
  - Timing calculations and control
  - In many cases, detailed logic design
- If you don't design before coding, you are likely to end up with the following
  - A very slow clock (long critical path)
  - Poor performance and large area
  - Non-synthesizable Verilog
  - Many HDL lint errors
- Must code at register transfer level
- Register and "transfer" (combinational) logic must be worked out before coding can start

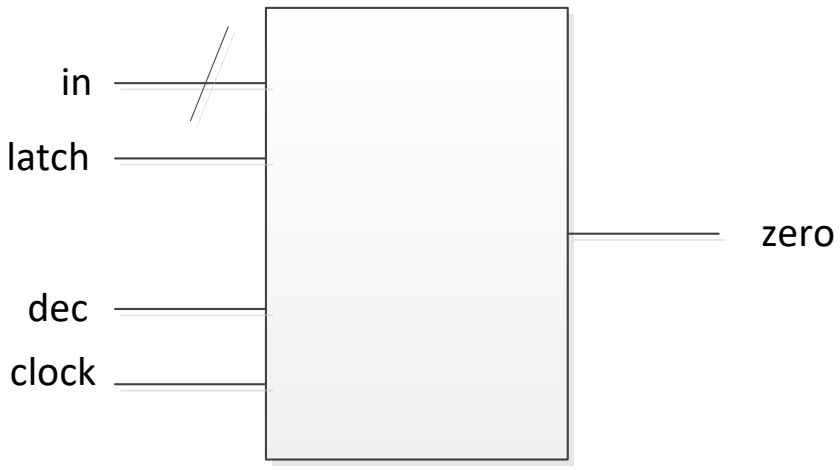


Design Before Coding – Steps in Design

1. Work out the hardware algorithm and overall strategy
2. Identify and name all registers (flip – flops)
   - Determine system timing while doing that
3. Identify the behavior of each “cloud” of combinational logic
4. Translate design to RTL
5. Verify design
6. Synthesize design

Example: Count Down Timer

- Specification
  - 4 bit counter
  - “count” value loaded from “in” on a positive4 clock edge when “latch” is high
  - “count” value decremented by 1 on a positive clock edge when “dec” is high
  - Decrement stops at 0
  - “zero” flag active high whenever count value is 0



Avoid Temptation

- Temptation # 1
  - “Verilog looks like C, so I’ll write the algorithm in C and turn it into Verilog with a few always @ statements”
  - Usual results: Synthesis problems, unknown clock level timing, too many registers
- Temptation # 2
  - “I cant work out how to design it, so I’ll code up something that looks right and let Synthesis fix it”
  - Usual result: Synthesis does not fix it
- Temptation #3
  - “Look at these neat coding structures available in Verilog, I’ll write more elegant code and get better results”
  - Usual result: Synthesis problems for neophytes, not all constructs are supported for Synthesis

What not to do

- Coding before design

```
always @(posedge clock)
   for (value = in; value >= 0; value --)
      if (value == 0) zero <= 1;
      else zero <= 0;
```

- or

```
always @(posedge clock)
   for (value = in; value >= 0; value --)
      @ (posedge clock)
         if (value == 0) zero <= 1;
         else zero <= 0;
```

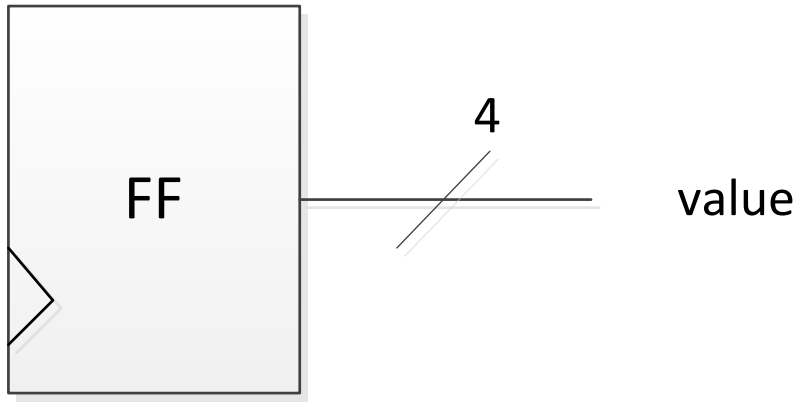
Strategy

- 1. Work out the hardware algorithm and overall strategy

- Strategy
  - Load “in” into a register
  - Decrement value of register while “dec” is high
  - Monitor register value to determine when zero

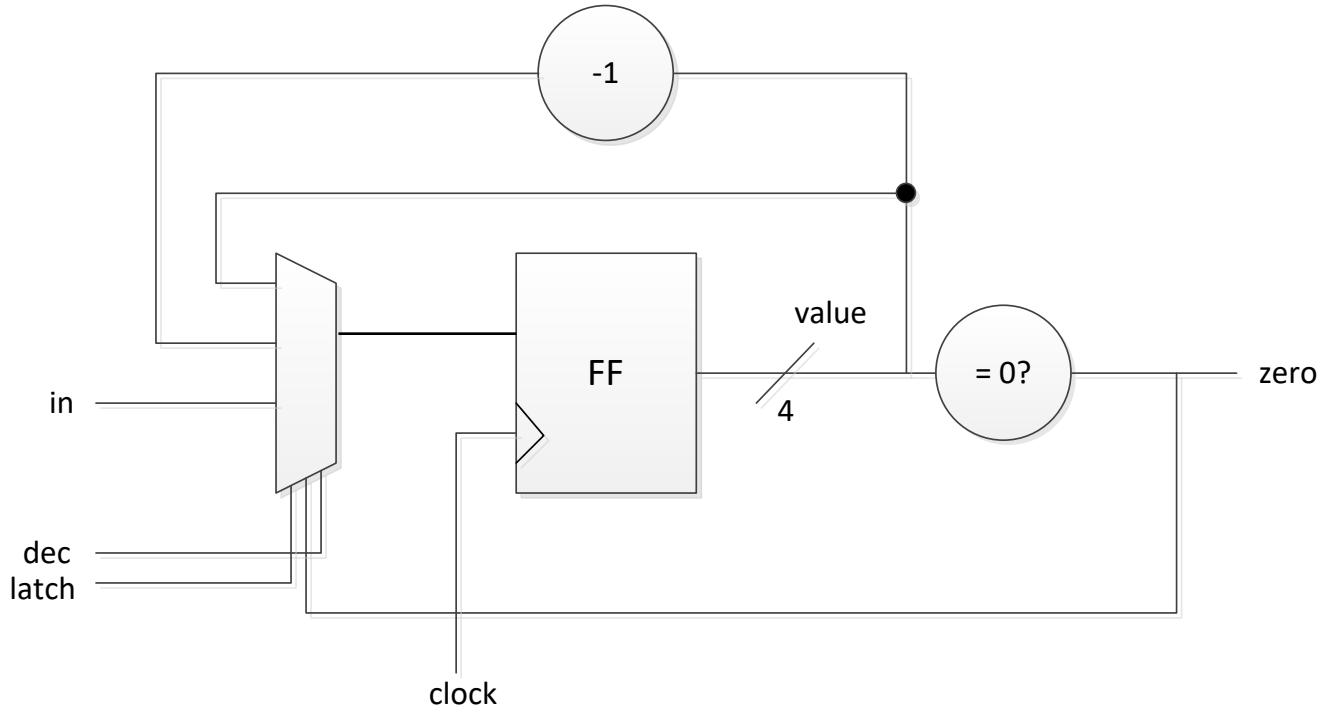
Design

- 2. Identify and name all the registers (flip – flops)



Design (cont'd)

- 3. Identify the behavior of each "cloud" of combinational logic



4. Translate Design into RTL

```
module counter (clock, in, latch, dec, zero);

   input clock;  //clock
   input [3:0] in;  //starting count
   input latch;  //latch 'in' when high
   input dec;    //decrement count when 'dec' high
   output zero;  //high when count down to zero

   reg [3:0] value  //current count value

   always @(posedge clock) begin
      if (latch) value <= in;
      else if (dec && !zero) value <= value -1'b1;
   end
   assign zero = (value == 4'b0);

endmodule
```

5. Verify Design

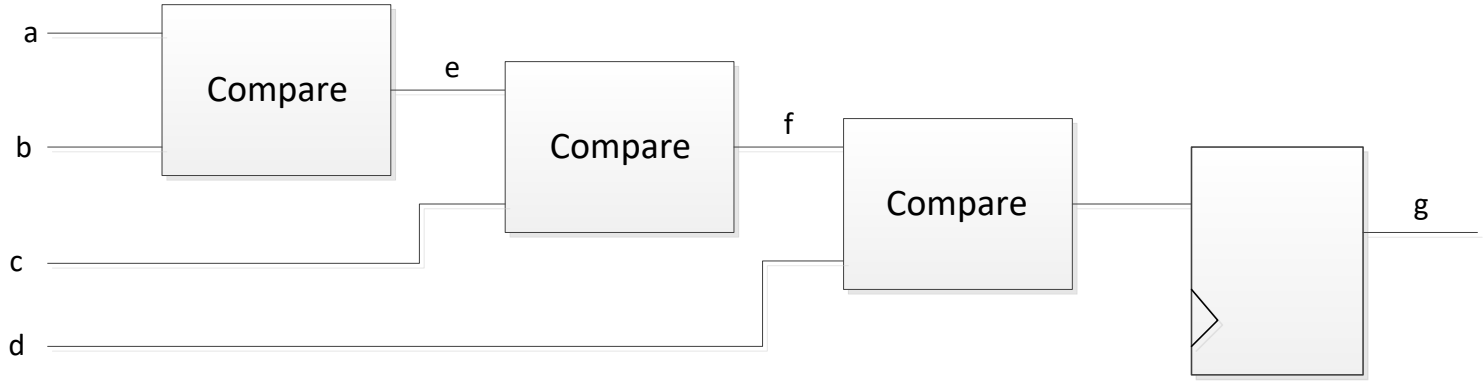
- Achieved by designing a test fixture to exercise design
- Will cover later

6. Synthesis

- After verifying correctness, the design can be synthesized to optimized logic
- The result is a gate level design (netlist)
- Will cover later

Exercise

- What do these look like in Verilog

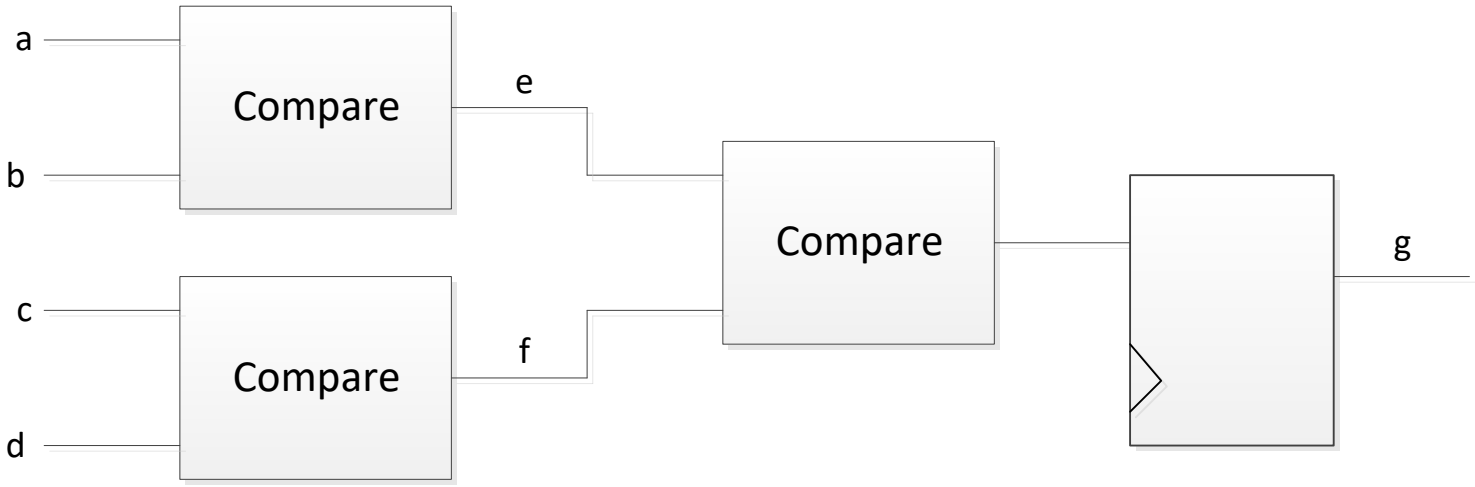



```
always @(a or b or c) begin  //or
always_comb
    if (a > b) e = a; else e = b;
    if (c > e) f = e; else f = c;
end
always @(posedge clock)
    if (d > f) g <= d; else g <= f;
```

Why 2 always?

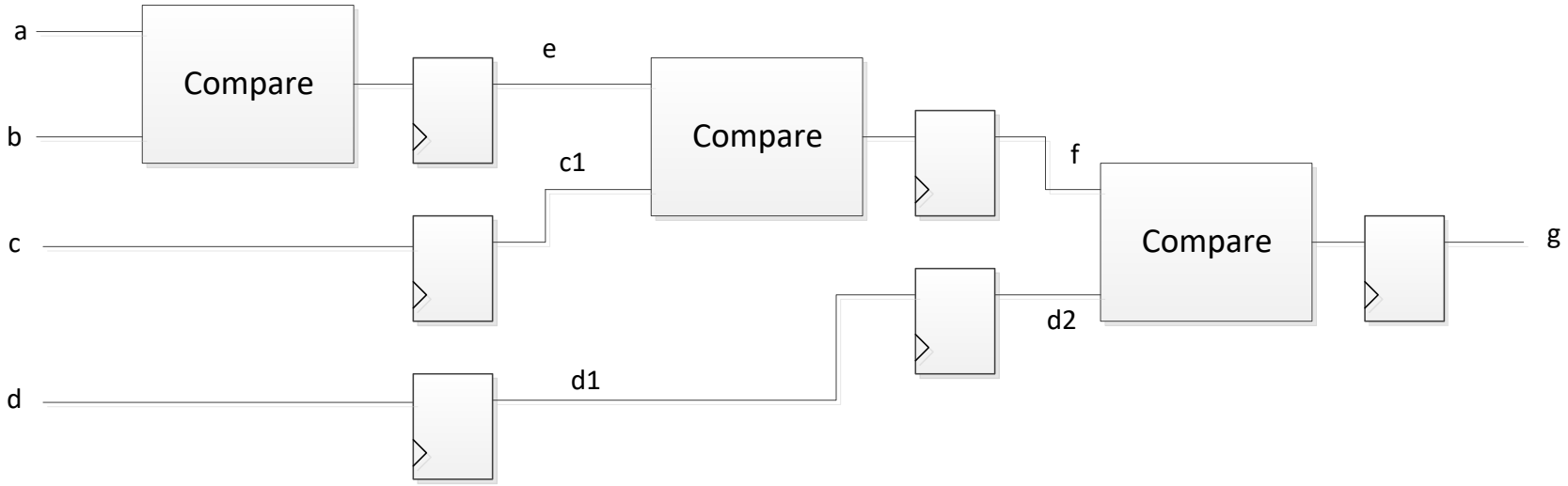
Exercise

- Produce a Verilog code fragment for the following schematic (use continuous assignment)



Exercise

- And for this



Note: All FFs have to be named

Blocking vs. Non-Blocking

- Blocking
  - Assignment of C blocked until A = B completed → They executed in sequence
- Non-Blocking
  - Assignment of C not blocked until A = B completed → They executed in parallel

```
begin
    A = B;
    C = D;
end
```

```
begin
    A <= B;
    C <= D;
end
```

Blocking vs. Non-Blocking

- Examples
  - Blocking
    - Sum in the second statement will have the value of the result of the first statement
  - Non-blocking
    - Sum in the second statement will have the value that has when the always block started its evaluation, not the result of sum <= a + b;

```
begin
    sum = a + b;
    prod = sum * factor;
    result = prod - c;
end
```

```
begin
    sum <= a + b;
    prod <= sum * factor;
    result <= prod - c;
end
```

Blocking vs. Non Blocking

- Which describes better what you expect to see in a sequential circuit?
  - Non blocking assignment
- Note
  - Use non blocking for flip flops
  - Use blocking for combinational logic
  - Don’t mix them in the same procedural block

Common Problems and Fixes

- Unintentional latches
  - Detected after reading the design, extra messages can be generated if always_ff is used
  - How to fix: make sure every variable is assigned for every way code is executed (except for flip flops)
  - If unfixed: you can have glitches on “irregular clock” to latch cause set up and hold problems in actual hardware (transient failures)

Problem code

```
always @(a or b) begin
   if (a) c = ~b;
   else d = |b;
end
```

Common Problems and Fixes

- Incomplete sensitivity list
  - Detected after reading the design
  - How to fix: all logic inputs have to appear in sensitivity list, use always @(*) (Verilog 2001), or use always_comb(SystemVerilog, recommended)
  - If unfixed: since simulation results won't match what actual hardware will do, bugs can remain undetected

Problem code

```
always @(a or b) begin
   if (a) c = b ^ a;
   else c = d & e;
   f = c | a;
end
```

Common Problems and Fixes

- Unintentional multi-driven nets
  - Detected after reading the design
  - How to fix: redesign hardware so that every signal is driven by only one piece of logic (or redesign s a tri-state buffer if that is the intention)
  - If unfixed: undesired results, this is a symptom of not designing before coding

Problem code

```
assign c = |b;
assign c = ^e;
```

Common Problems and Fixes

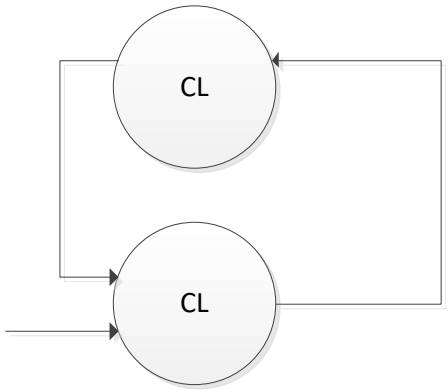
- Improper startup
  - Cannot be detected
  - How to fix: make sure don’t cares are propagated
  - If unfixed: possible undetected bug in reset logic

Problem code

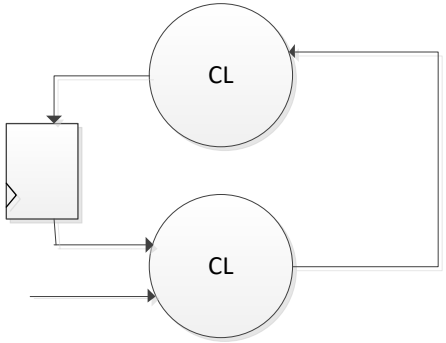
```
always @(posedge clock)
   if (a) q <= d;

always @(q or e)
   case (q)
      0: f = e;
      default: f = 1;
   endcase
```

Common Problems and Fixes

- Feedback in combinational logic
  - Either results in
    - Latches, when the feedback path is short
    - Timing Arcs when feedback path is convoluted
  - Fix by redesigning logic to remove feedback
    - Feedback can only be through FFs



WRONG



OK

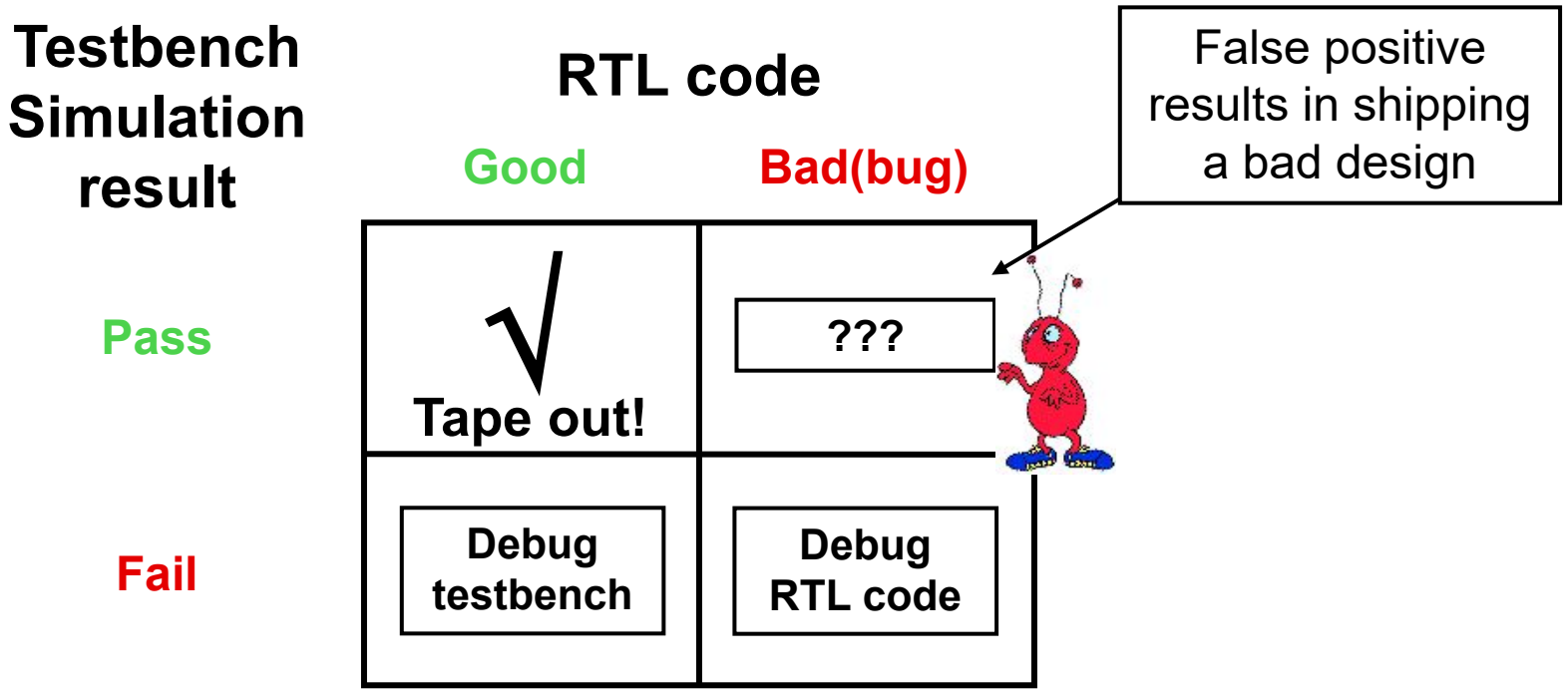
Common Problems and Fixes

- Incorrect use of FOR loops
  - Only correct use is to iterate through an array of bits
- Unconstrained timing
  - To calculate permitted delay, synthesis must know where the FFs are
  - If you have a path from input port to output port that does not path through a FF, synthesis cannot calculate timing
  - To fix: revisit modules partitioning (see later) to include FFs in all paths

Simulation Overview

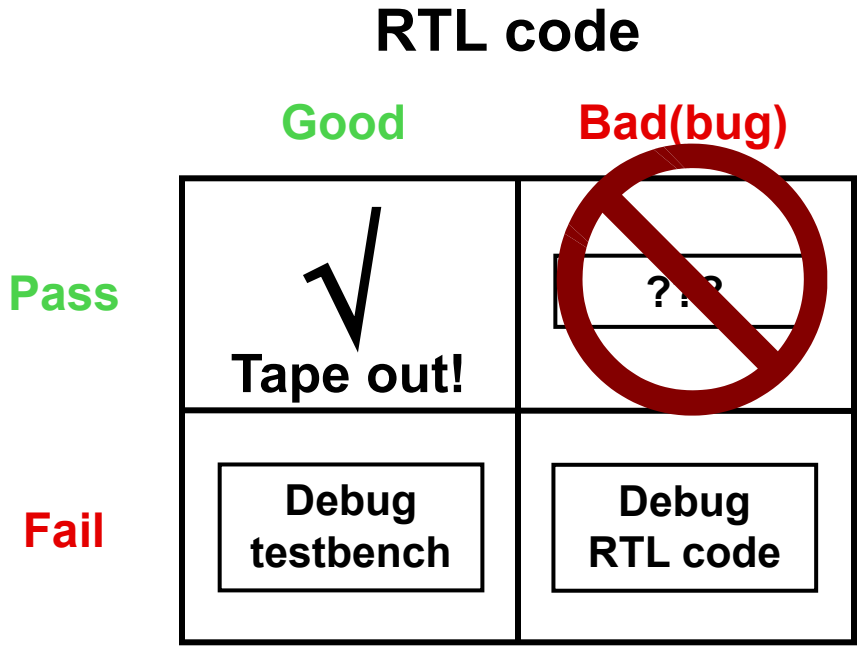
Verification Goals

- Verify RTL design code
  - Fully conform with specifications
- Must avoid false positives (Untested functionalities)



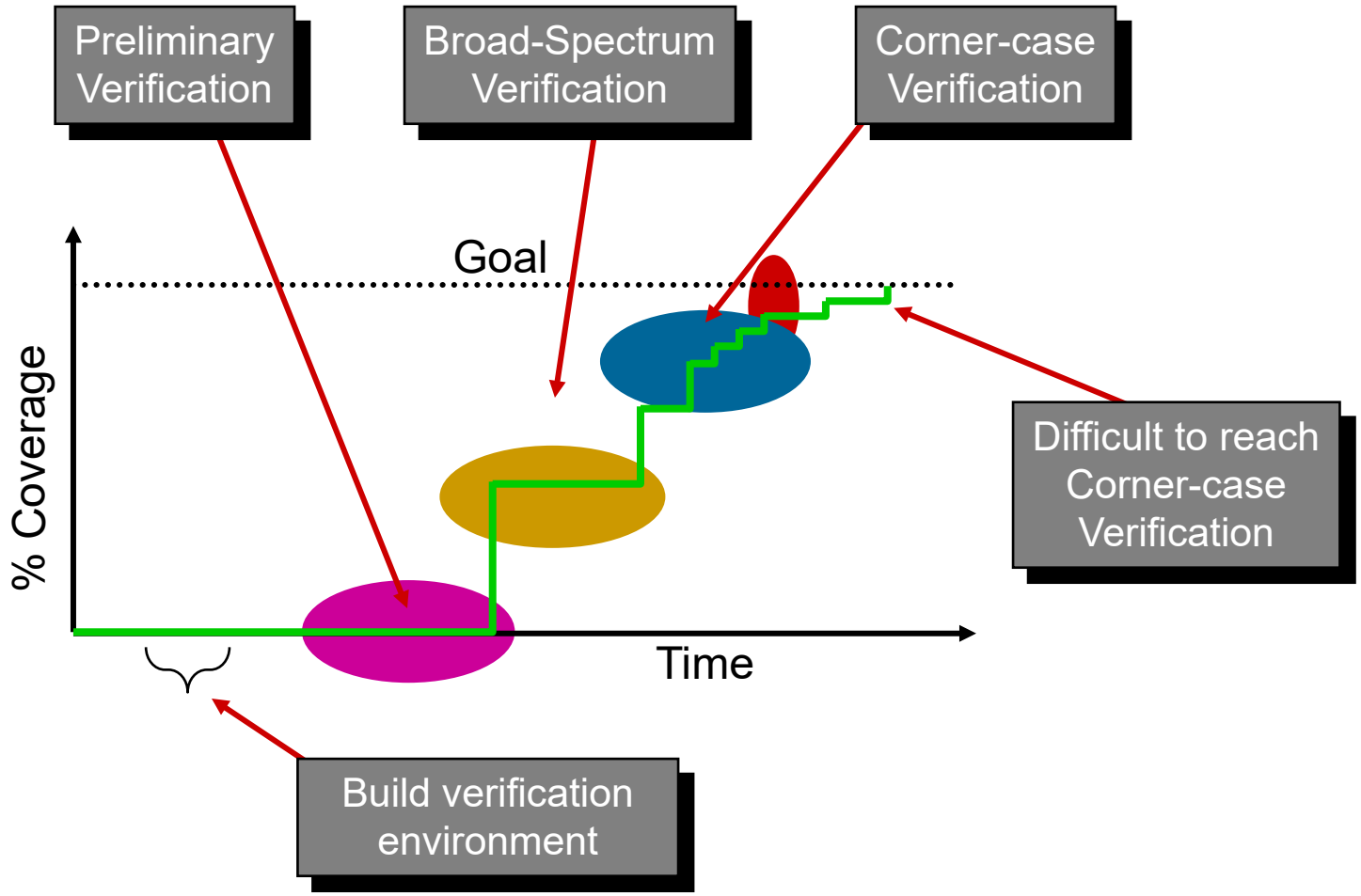
Verification Goals (Cont'd)

- Test environment must
  - Be structured for debug
  - Avoid false positives
- Test must
  - Achieve functional coverage
    - Prevent untested regions
  - Reach corner cases
    - Anticipated cases
    - Error injection
      - Environment error
      - DUT error
    - Unanticipated errors
      - Random tests
    - Be robust, reusable, scalable



Process of reaching Verification Goals

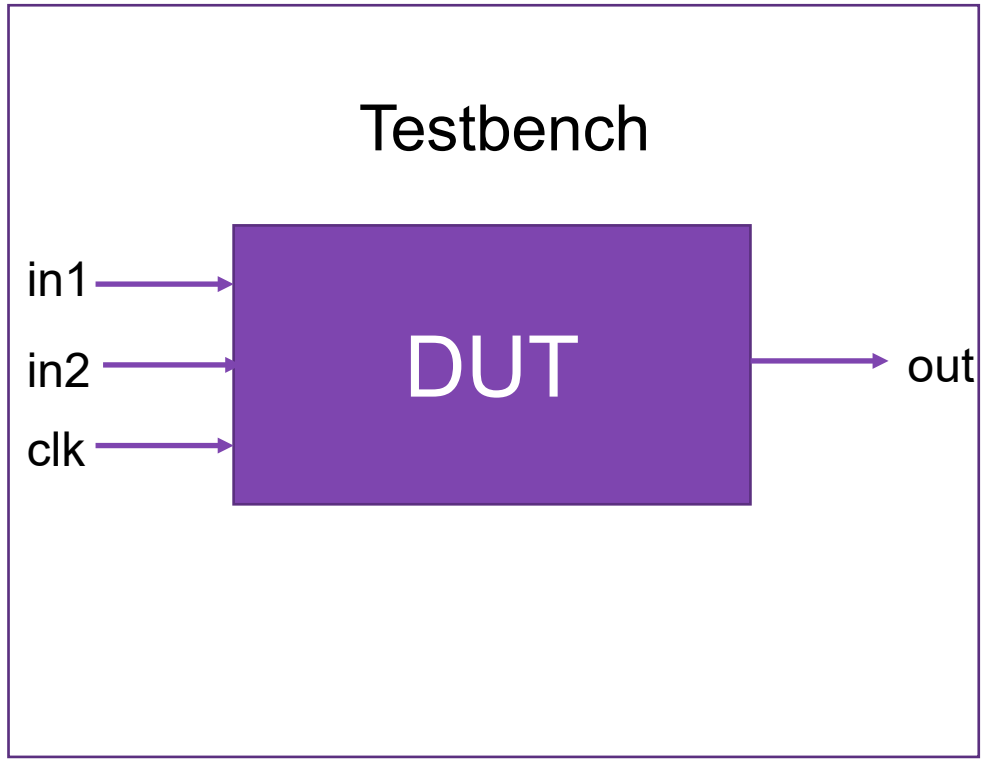
Phases of verification



Testbenches

- A test bench specifies a sequence of inputs to be applied by the simulator to a Verilog-based design.
- The test bench uses an initial block and delay statements and procedural statement.
- Verilog has advanced “behavioral” commands to facilitate this:
  - Delay for n units of time
  - Full high-level constructs: if, while, sequential assignment.
  - Input/output: file I/O, output to display, etc.

Example Testbench

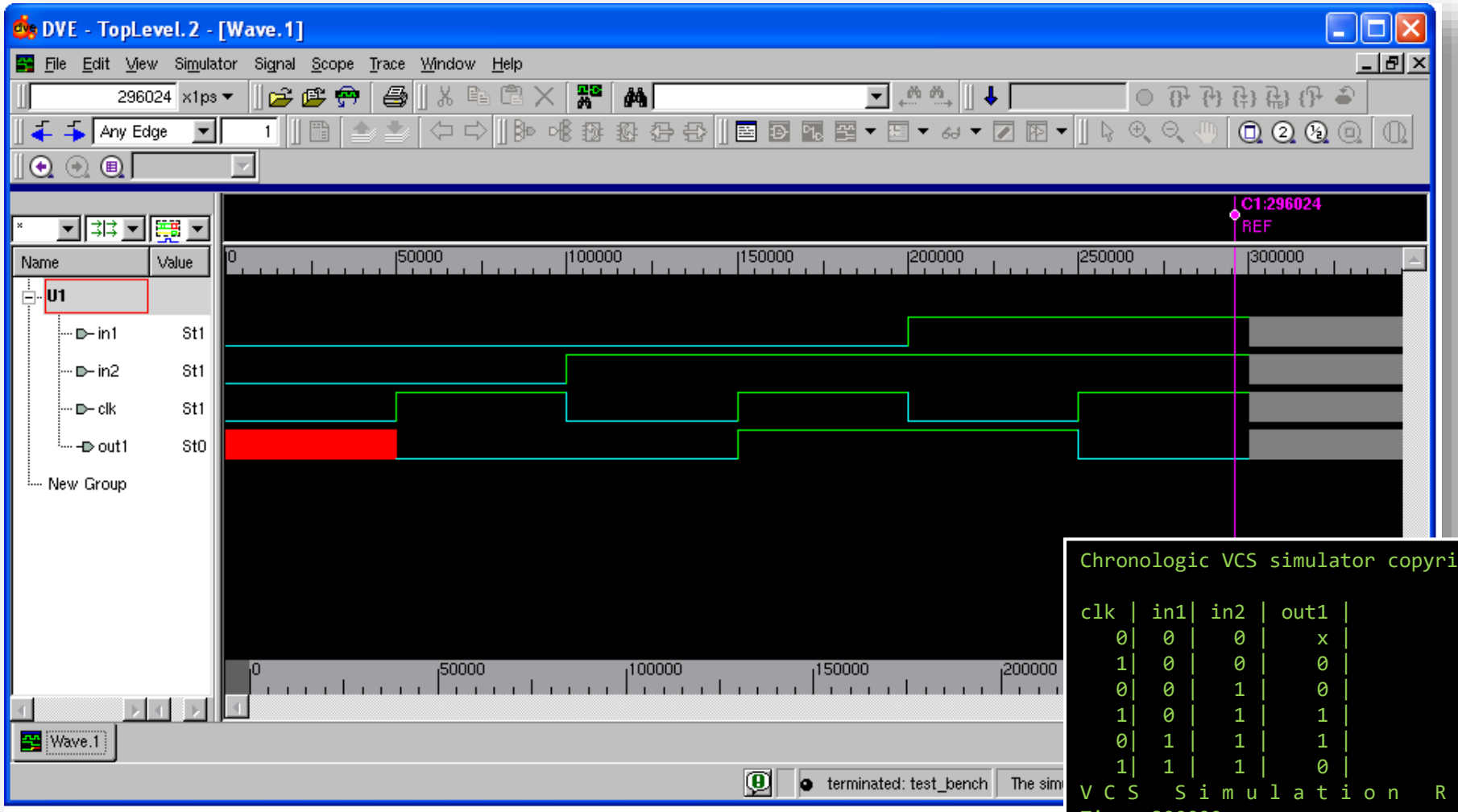



```
`timescale 10ns/1ps
module test_bench;
// Interface to communicate with the
logic a, b, clk;
logic c;
// Device under test instantiation
DUT U1 (.in1(a), .in2(b), .clk(clk), .out1(c));
initial
begin  // Test program
  test1 ();
  $finish;
end
initial
begin
  clk = 0;
  forever #5 clk = ~clk;
end
initial
begin // Monitor the simulation
  $display ("clk | in1| in2 | out1 |");
  $monitor ("  %b| %b |  %b |   %b |",clk, a, b, c);
end
endmodule
```

```
task test1 ();
begin
      a = 0;    b = 0;
  #10 a = 0;    b = 1;
  #10 a = 1;    b = 1;
  #10 a = 1;    b = 0;
end
endtask
```

```
module DUT (in1, in2, clk,
out1);
input in1, in2;
input clk;
output logic out1;
always_ff @(posedge clk)
 out1 = in1^in2;
endmodule
```

Simulation results




```
Chronologic VCS simulator copyright 1991-2008

clk | in1| in2 | out1 |
   0|  0 |   0 |    x |
   1|  0 |   0 |    0 |
   0|  0 |   1 |    0 |
   1|  0 |   1 |    1 |
   0|  1 |   1 |    1 |
   1|  1 |   1 |    0 |
V C S   S i m u l a t i o n   R e p o r t
Time: 300000 ps
CPU Time:      0.010 seconds;       Data structure size:   0.0Mb
```

Writing Verilog Testbenches

Precision macro – `timescale

- Defines the time units (specified by #<number> and simulation precision (smallest increment)

```
`timescale Time_Unit/Precision_Unit
```

- Time : `1 10 100`
- Units: `ms us ns ps fs`
- The precision unit must be less than or equal to the time unit
- Example:

```
`timescale 10ns/1ps
```

Initial block

- Contains a statement or block of statement which is executed only once, stating at the beginning of the simulation
- Each block is executed concurrently before simulation time 0 (ignored by synthesis)
- No sensitivity list

```
initial
begin
    X = 1'b0;
end
```

Forever block

- Cause one or more statements to be executed in an infinite loop.
- Example: clock signal generation

```
initial
begin
  clk = 0;
  forever #5 clk = ~clk;
end
```

Delay (#)

- Specifies the delay time units before a statement is executed during simulation
- Regular delay control

```
#<num> y = 1;
```

  - Regular delay control is used when a non-zero (`<num>`) value is specified
- Zero delay control

```
#0;
```

  - Ensure that statements are executed at the end of time `0`
- Intra-assignment delay control

```
y = #<num> x+z;
```

Delay example

```
parameter sim_cycle = 6;
initial
begin
  x = 0;
  #10 y = 1;              //assignment is delayed 10 time units
  #(sim_cycle/3) x = 2; //delay number defined from a parameter value
end
initial
begin
  p = 0; q = 0;
  r = #5 p+q;          //Take the value of p and q at time 0, evaluate
                       //p+q and wait 5 time units to assign value to r
end
initial
begin
  #0 x = 1;            //x=0,p=0;q=0,x=1 are executed a time 0 but x=1 is
                       //executed at the end
end
```

Task

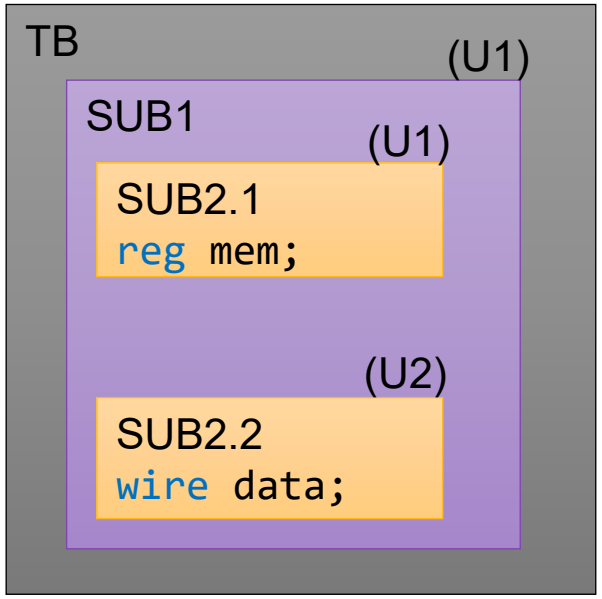
- Task helps to simplify the test bench
- Can include timing control
  - Inputs
  - Delays (#) and regular event control (@)

```
task load (input [7:0] data,
           input enable);
begin
  #2  rst_n = 1’b0;
  #10 data_in = data;
  #2  read_ena = enab;
end
endtask
```

```
task reset;
begin
  rst_n = 1'b0;
  repeat(3)
  @(negedge clk);
  rst_n = 1'b1;
end
endtask
```

Hierarchical names

- Hierarchical name references allows us to denote every identifier in the design with a unique name
- Hierarchical name is a list of identifier separated by dots (“.”)




```
module TB ();
//...
$monitor (“%b”, TB.U1.U1.mem);
$monitor (“%b”, TB.U1.U2.data);

endmodule
```

System tasks and functions

- System task are tool specific tasks and functions.

```
$display, $write // utility to display information
$monitor         // monitor, display
$clog2           // log base 2, useful to determine array sizes
$time, $realtime // current simulation time
$finish          // exit the simulator
$stop            // stop the simulator
$timeformat      // format for printing simulation
$random          // random number generation
```

$display - $strobe - $monitor

- Print message to a simulator (similar to C printf)
- $display (*format*, *args*)
  - Display information before RHS evaluation (before non-blocking assignments)
- $strobe (*format*, *args*)
  - Display information after RHS evaluation (after non-blocking assignments)
- $monitor (*format*, *args*)
  - Print-on-change after RHS and non-blocking assignments whenever one argument change

Verilog string and messages

```
initial
begin
$display (“Results\n”);
$monitor (“\n Time=%t  X=%b”, $time , X);
end
```

Useful format specifiers:

- %h hex integer
- %d decimal integer
- %b binary
- %c single ASCII character
- %s character string
- %t simulation time
- %u 2-value data
- %z 4-value data
- %m module instance name

$random

- Random number generation
- Returns as 32-bit signed integer

```
reg [31:0] rand1, rand2, rand3;
rand1 = $random;     // generates random numbers
rand2 = $random % 60;   // random numbers between -59 and 59
rand3 = {$random} % 60; // random positive values
                // between 0 and 59
```

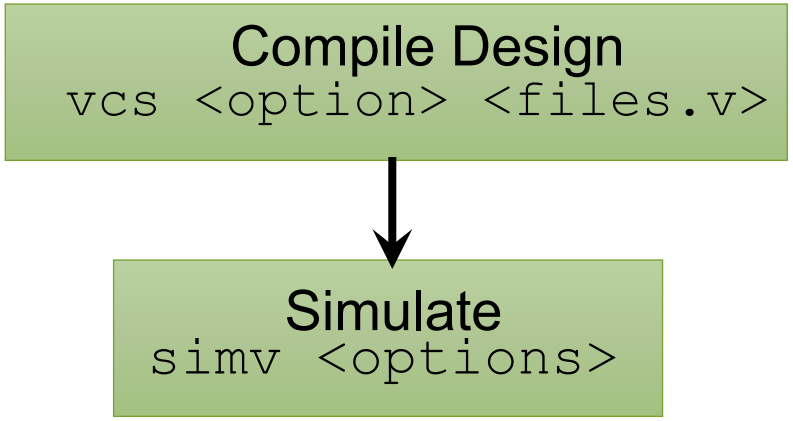
VCS Setup and Use Model

What is VCS

- Complete verification environment
- Multi-language simulator
  - VHDL
  - Verilog
  - SystemC
  - SystemVerilog
  - C/C++

Pure Verilog Flow (2-steps)

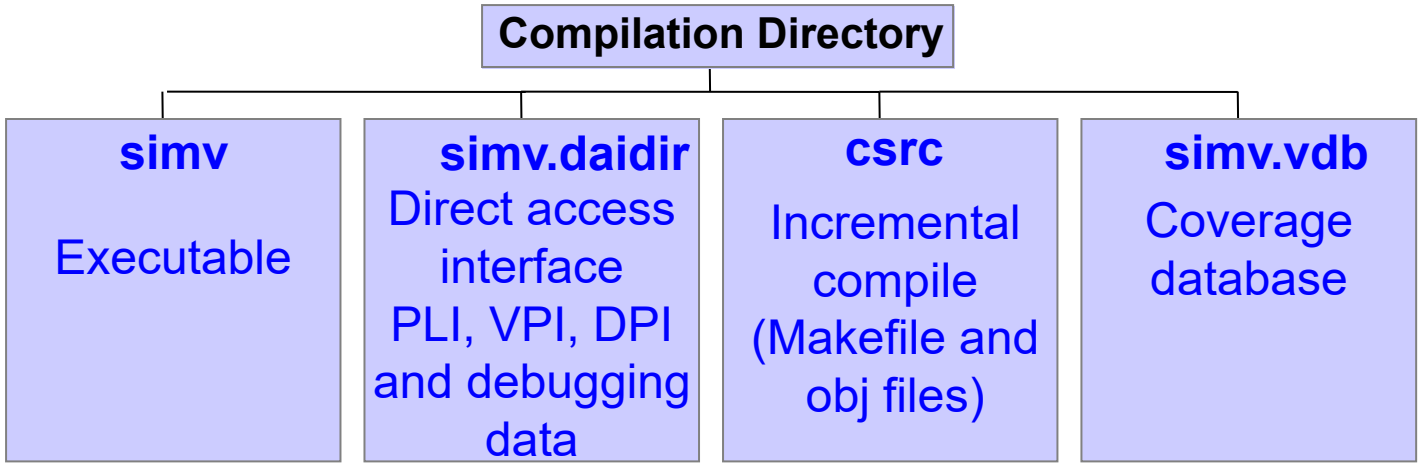
- Elaborate/Compile the design
  - Specify all Verilog source code
  - Command: **`vcs`**
- Simulate the design
  - Command: **`simv`**
- Notes:
  - No setup file is needed
  - Verilog has no concept of logical libraries
- Analyze Verilog source
  - Verilog has no strict order dependencies
  - For pure Verilog designs, all files can be given on a single compile line



Elaboration and Compilation

- VCS command performs elaboration and compile
- Methodology: Elaborate/Compile once, run multiple times
  - The following tasks are performed
    - Generic and parameter resolution – resolves generic/parameters in the design with their specified values.
    - File handle resolution – opens file handles for reading and/or writing files.
    - Design hierarchy and architecture resolution – resolves multiple architectures.
    - Code generation – generates C code for the entire design.
    - Elaboration/Compilation and Linking – elaborates the generated code and statically links all objects to generate the simulation executable.

VCS Directory Structure



Compiling for Debug

- Goal: Perform interactive or post-simulation debug
  - Want to keep design visibility
  - Generate waveforms
- Typical compile for post-simulation debug:
  - Enables waveform generation (VPD, VCD, etc)
  - Minimum requirement for `$dumpvars` or `$vcdpluson`

```
%> vcs -debug_pp <options>
```

- Typical compile for interactive debug:
  - Enables full interactive Tcl and GUI debugging

```
%> vcs -debug_access+all <options>
```

- Visibility into complex types
  - Enables debug of multi-D arrays and other complex types

```
%> vcs -debug +memcbk <options>
```

  - Use carefully!
    - Dumping every memory in a large design can cause simulation to seemingly hang
    - Commands allow users to specify scopes and specific objects for debugging

Debugging with VCS

- Debugging flow
  - Interactive debug
  - Post process debug
- Two steps flow
  - `vcs -kdb -debug_access+all <other option> testbench_file description_files`
  - Compile design and generated elaborated KDB to ./simv.daidir
- Debug with simulator (interactive mode)
  - `./simv -verdi`
- Post processing mode
  - `verdi -ssf my.fsdb`

SYNOPSYS®

Thank You