Verilog Design Language

Doug Smith
Texas Instruments, Knoxville
11 / 1 / 11

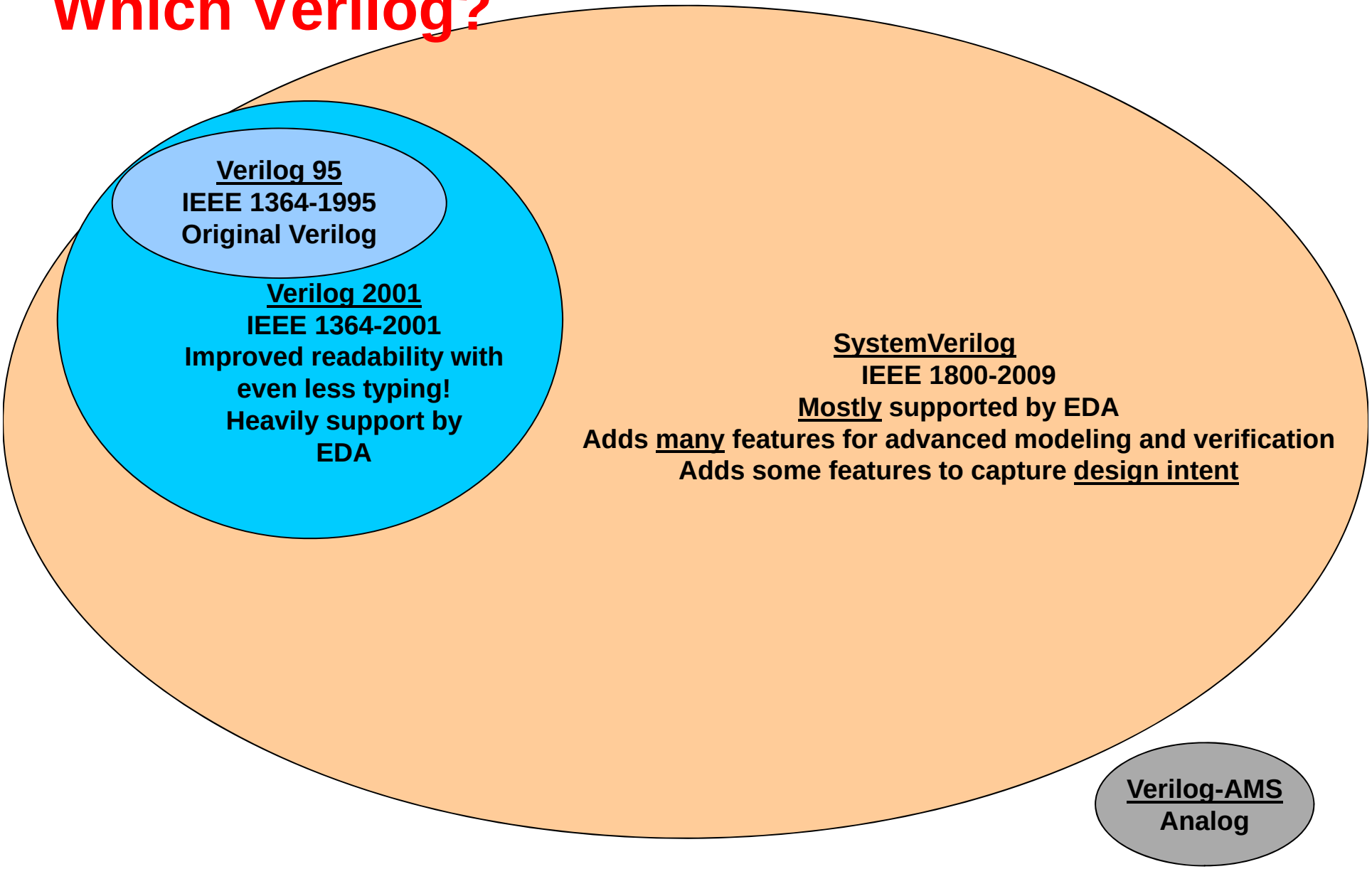
Introduction

Texas Instruments

- 2010: #5 semiconductor company by sales, ~$13B revenue
  - Acquisition of National Semi should move TI to #3
  - ~45,000 total products
- Knoxville office established in December 2005
  - Currently 9 Design Engineers in 2 Design Groups
  - Multi-cell, Battery Charge Management
    - 6 analog designers, 2 digital
    - Switch Mode Battery Charger ICs
    - Linear Chargers
    - Wireless Chargers
    - All design work performed in Knoxville
    - ~50% market share
  - Home Audio
    - 1 analog designer, 1 analog co-op
    - Class D Audio Amplifiers

Verilog Overview

Which Verilog?



Why Verilog?

- Widely used by industry in US
  - VHDL more common in defense industry, Europe and IBM
  - EDA tools used to work better for Verilog, but have improved for VHDL
    - Netlists are in Verilog
- VHDL and Verilog are converging
  - SystemVerilog adds many features of VHDL
  - New VHDL spec?
  - Easy to co-simulate VHDL and Verilog
- More compact
  - Much less typing → faster to implement
  - Easier to auto generate code
- More freedom
  - Free to make mistakes
  - Not as restrictive as VHDL

Basic Concepts

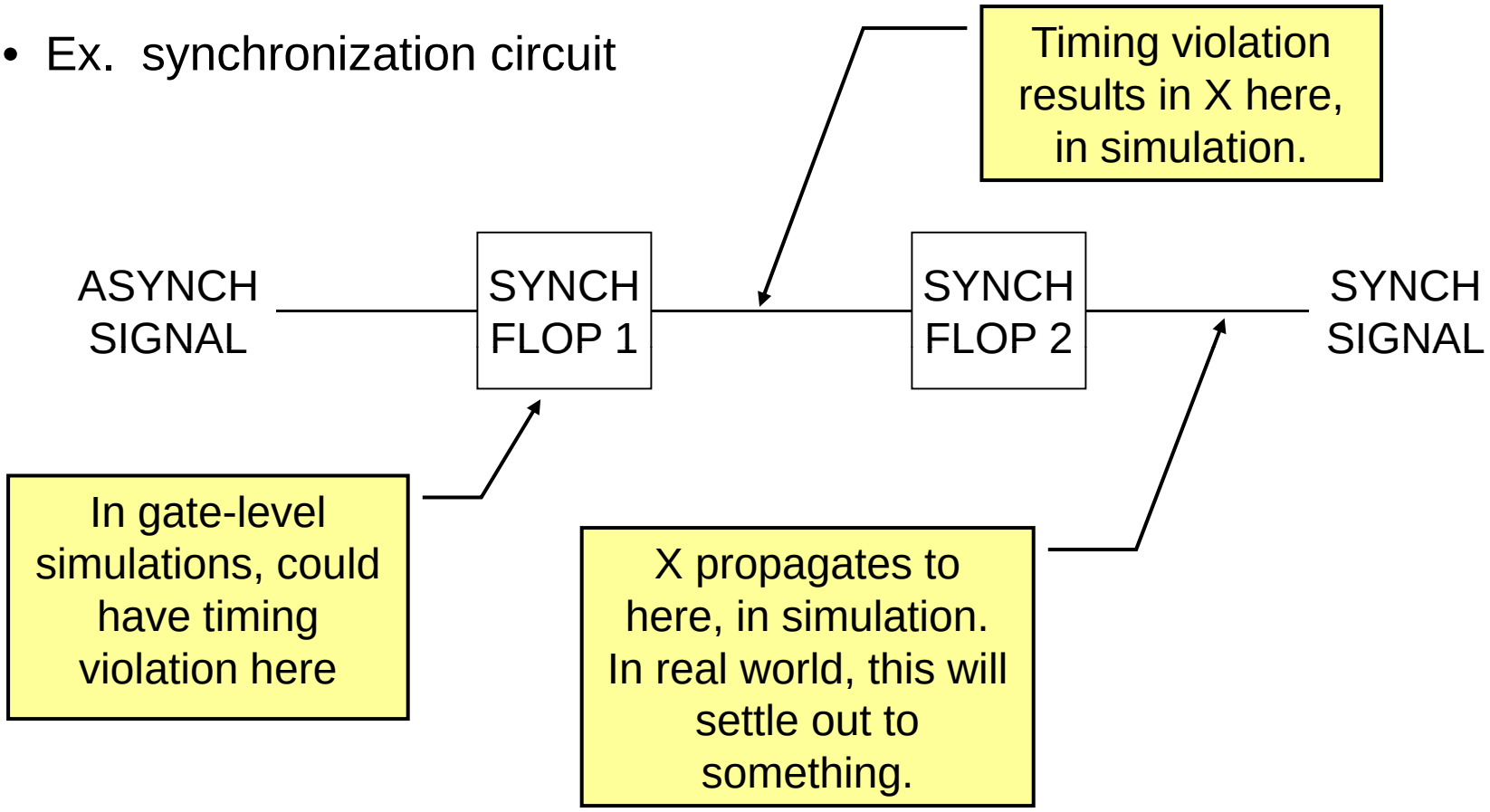
Logic Values

- Verilog logic data types are 4-state: 0, 1, X, Z
  - reg, wire, integer
  - 0 = logic low
  - 1 = logic high
  - Z = tri-state, high-impedance
  - X = unknown value
  - In real world there is no such thing as X!

- SV introduced 2-state types: 0, 1
  - bit, shortint, int, longint, byte
  - Uses less memory and evaluates faster
  - For verification only
  - Potential to mask X's

No Xs in Real World

- Ex. synchronization circuit



REG != Flop

```
reg dout;
wire sel;
wire din;

always @(sel or din)
if (sel)
  dout = din;
else
  dout = 1'b0;
```

Synthesizes to combinational logic

```
reg dout;
wire sel;
wire din;

always @(sel)
if (sel)
  dout = din;
```

Synthesizes to latch

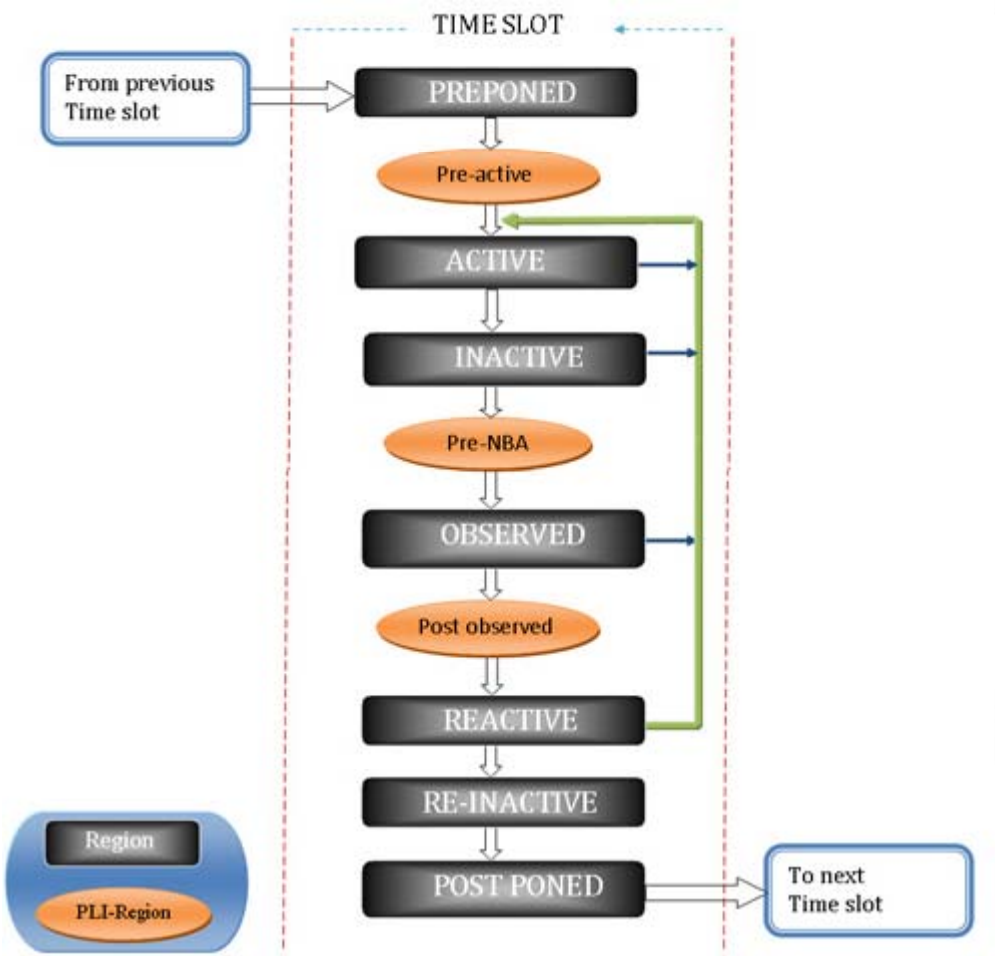
```
reg dout;
wire sel;
wire din;

always @(posedge sel)
if (sel)
  dout  = din;
```

Synthesizes to flop

Simulator Event Scheduling

- Simulating parallel hardware …
- …on a sequential machine.
- Keep in mind that every time slot is made up of multiple steps to evaluate.
- Very important to remember that even concurrent assignments are actually performed sequentially by simulator.



Assignments

- Blocking (=)
  - Assignment is made before execution continues
- Non-blocking (<=)
  - Assignment is schedule to be made with all other non-blocking assignments

```
initial
  begin
  A = 3;
  #1;
  A = A + 1;
  B = A + 1;
```

A is assigned 4, then B is assigned 5

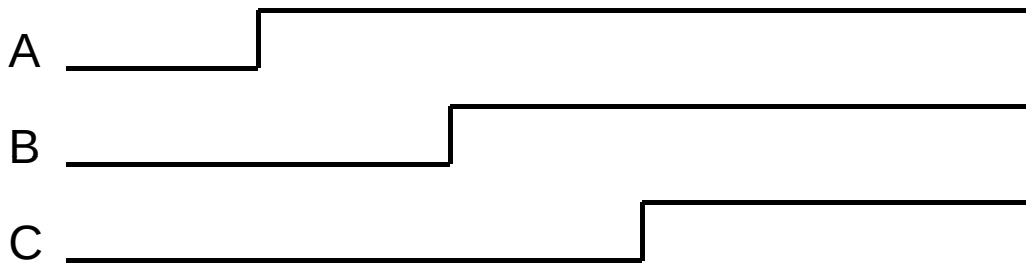
```
initial
  begin
  A = 3;
  #1;
  A <= A + 1;
  B <= A + 1;
```

A is assigned 4, when B is assigned 4

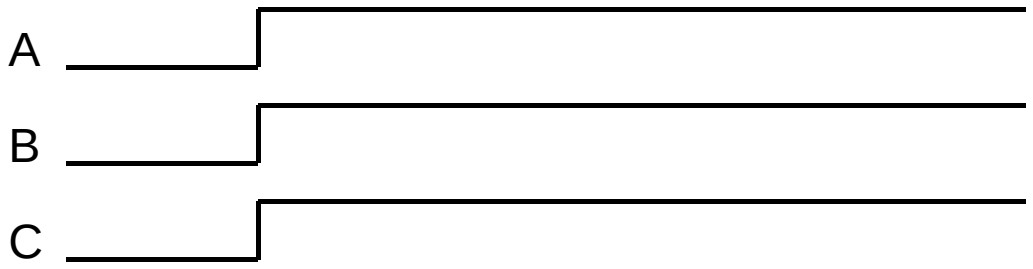
begin-end versus fork-join

- begin-end timing is sequential, fork-join is parallel

```
initial
   begin
   #10ns A = 'b1;
   #10ns B = 'b1;
   #10ns C = 'b1;
   end
```




```
initial
   fork
   #10ns A = 'b1;
   #10ns B = 'b1;
   #10ns C = 'b1;
   join
```



Verilog has no Concept of Time

- simulators are event driven
  - Doubling your clock will NOT shorten your simulation time
  - Same number of events
  - Same number of evaluation steps
- time values are only for your benefit

Verilog has Multiple Purposes

- Verification
  - Essentially, entire language is available for use
- Design
  - Can only use synthesizable sub-set
- Gate-level Modeling
  - Used to create standard cell simulation libraries

Synthesizable vs. Simulatable

- All Verilog commands can be simulated, but not all can be synthesized
- Synthesizable code
  - Keep it simple
  - Generally stick with Verilog-2001
  - Utilize the few SV features which capture intent
- Simulatable code
  - Test benches, models, monitors
  - SV very powerful
    - classes, data structures, queues, associative arrays, random sequences, assertions
  - S/W background is useful

Synthesizable Code

Synthesizable Code

- Typical synthesizable block
- Simple = fewer mistakes and better results

```
module counter (
       input                rst_n, clk, enable,
       output reg    [7:0]  count,
       output               alarm
);

assign alarm  = (count == 8'h3F);

always_ff @(posedge clk, negedge rst_n)
  if (!rst_n)         count  <= 'h00;
  else if (alarm)     count  <= 'h00;
  else if (enable)    count  <= count + 'b1;

endmodule
```

Commonly Used Synthesizable Code

MOST COMMON

- module, endmodule
- input, output, inout (at top-level)
- reg, wire, logic
- always, always_ff, always_comb, always_latch
- assign
- posedge, negedge, or
- case…endcase, if … else, ? … :
- &, ^, |, ~
- !, &&, ||
- begin … end

LESS COMMON

- parameter, defparam, localparam
- `define, `ifdef, `ifndef, etc.
- casez, casex
- typedef, enum

EVEN LESS COMMON

- interface, package
- unique, priority
- generate, endgenerate, genvar

Common Coding Mistakes

Wire name typos

- By default, new signal names are assumed to be 1-bit wires

```
timer i1 (.enable(timerEnable), .alarm(timerAlarm));
control i2 (... .enableTimer(enableTimer));
```

- Design is incorrect, but no error will be reported.
- **`default_nettype none** forces declaration of every wire
- SV **.name** and **.*** do not infer implicit wires

```
timer i1 (.enable, .alarm);
control i2 (.*);
```

Wrong OR in sensitivity list

- Legal to use either “or” separator, or “|” operation
- However, they operate differently

```
always @(a | b)
       sum = a + b;
```

If a=1 and b changes from 0 to 1, always will <u>not</u> trigger

```
always @(a or b)
       sum = a + b;
```

If a=1 and b changes from 0 to 1, always will trigger

- Instead of “or”, use “,” or “*”, or always_comb

```
always @(a, b)
always @*
always_comb
```

Vector in sequential logic sensitivity list

- Legal to use vector, but wrong
- Vector in sensitivity list of combinational logic is fine

```
wire   [7:0]  vector;
always @(vector)
       ...
```

always block triggers when any bit in vector changes

```
wire   [7:0]  vector;
always @(posedge vector)
       ...
```

always block triggers when only when LSB in vector changes

- Posedge/negedge should only have 1-bit arguments

```
wire   [7:0]  vector;
always @(posedge vector[7])
       ...
```

Incomplete sensitivity list

- Legal, but will not implement intended design
- Tools usually displays warning message

```
always @(A)
  C = A & B;
```

always block triggers only when A changes. C will not be updated when B changes.

```
always @(A or B)
  C = A & B;
```

always block triggers when either A or B changes. C will update properly.

- SV introduces @* and always_comb

```
always @*
  C = A & B;
```

```
always_comb
  C = A & B;
```

Unintentionally infer latch

- reg / always block can be flip-flop, latch or combinational logic
- Latch usually result of incomplete case or if statement

```
reg dout;
wire sel;
wire din;

always @(sel, din)
if (sel) dout = din;
else     dout = 1'b0;
```

Fully defined, so results in combinational logic

```
reg dout;
wire sel;
wire din;

always @(sel, din)
if (sel)
       dout   = din;
```

Not defined for sel = 0, so results in latch

- Good practice to specify default value when coding combo

Unintentionally infer latch

- SV adds always_comb, always_ff and always_latch to capture intent

```
logic dout;
wire sel;
wire din;

always_comb
if (sel) dout = din;
else     dout = 1'b0;
```

Results in error if result is not combinational. No sensitivity list needed.

```
reg dout;
wire sel;
wire din;

always_latch
if (sel)
      dout   = din;
```

Results in error if result is not a latch. No sensitivity list needed.

Improperly nested IF statements

- Good practice to use begin…end with nested IF statements

```
if (A >= 5)
  if (A <= 10)
    C = 1'b1;
else
  C = 1'b0;
```

Confusing code. Implies that C is set to 0, when A < 5. But actually sets C to 0 when A > 10.

```
if (A >= 5)
  begin
  if (A <= 10)
    C = 1'b1;
  end
else
  C = 1'b0;
```

C is set to 0, when A < 5.

Using wrong NOT, AND, OR

- Logical operators (!,&&,||) versus bitwise operators (~,&,|)

```
reg A;
reg [1:0] B, C:
initial
   begin
   A = 1'b1;
   B = 2'b01;
   C = 2'b10;
   end
```

```
if (!B)          -- evaluates to FALSE
if (~B)          -- evaluates to TRUE

if (B && C)      -- evaluates to TRUE
if (B & C)       -- evaluates to FALSE
```

- Always use logical operators when looking for true/false result (if statements, etc.)

UART Example

UART_RX

```
module uart_rx #(
  parameter DBIT = 8,
            SB_TICK = 16
) (
  input wire clk, reset,
  input wire rx, s_tick,
  output reg rx_done_tick,
  output wire [7:0] dout
);
```

```
module uart_rx #(
  parameter DBIT    = 8,
            SB_TICK = 16
) (
  input         clk, reset,
                rx, s_tick,
  output reg    rx_done_tick,
  output [7:0]  dout
);
```

No need to explicitly declare inputs and outputs as wire.

Not necessary to declare input on each line. Continues from previous line.

UART_RX

```
// symbolic state declaration
  localparam [1:0]
    idle  = 2'b00,
    start = 2'b01,
    data  = 2'b10,
    stop  = 2'b11;

// signal declaration
reg [1:0] state_reg,
          state_next;
reg [3:0] s_reg, s_next;
reg [2:0] n_reg, n_next;
reg [7:0] b_reg, b_next;
```

```
typedef enum reg [1:0] {
  idle,
  start,
  data,
  stop
} state_t;

// signal declaration
state_t state_reg, state_next;
reg [3:0] s_reg, s_next;
reg [2:0] n_reg, n_next;
reg [7:0] b_reg, b_next;
```

typedef enum approach records variables using names instead of bit values

UART_RX

```
always @(posedge clk,
         posedge reset)
  if (reset)
    begin
    state_reg <= idle;
    s_reg <= 0;
    n_reg <= 0;
    b_reg <= 0;
    end
  else
    begin
    state_reg <= state_next;
    s_reg <= s_next;
    n_reg <= n_next;
    b_reg <= b_next;
    end
```

```
always_ff @(posedge clk,
            posedge reset)
  if (reset)
    begin
    state_reg <= idle;
    s_reg <= 'b0;
    n_reg <= 'b0;
    b_reg <= 'b0;
    end
  else
    begin
    state_reg <= state_next;
    s_reg <= s_next;
    n_reg <= n_next;
    b_reg <= b_next;
    end
```

Use always_ff to specify intent

UART_RX

```
// FSMD next-state logic
always @*
  begin
  state_next = state_reg;
  rx_done_tick = 1'b0;
  s_next = s_reg;
  n_next = n_reg;
  b_next = b_reg;
```

```
// FSMD next-state logic
always_comb
  begin
```

Use always_comb to specify intent.

Dangerous to assign values that are re-assigned later. Could create false event. Assign all variables in every case condition. Add default case.

UART_RX

```
case (state_reg)
  idle:
    if (~rx)
      begin
      state_next = start;
      s_next = 0;
      end
```

```
case (state_reg)
  idle:
    begin
    rx_done_tick = 1'b0;
    n_next = n_reg;
    b_next = b_reg;

    if (~rx)
      begin
      state_next = start;
      s_next = `b0;
      end
    else
      begin
      state_next = state_reg;
      s_next = s_reg;
      end
    end
```

Assign all variables.

UART_RX

```
case (state_reg)
  start:
    if (s_tick)
      if (s_reg==7)
        begin
        state_next = data;
        s_next = 0;
        n_next = 0;
        end
      else
        s_next = s_reg + 1;
```

Remove nested if statements.
Assign all variables.

```
case (state_reg)
  start:
    begin
    rx_done_tick = 1'b0;
    b_next = b_reg;
    if (!s_tick)
      begin
      state_next = state_reg;
      s_next = s_reg;
      n_next = n_reg;
      end
    else if (s_reg==7)
        begin
        state_next = data;
        s_next = 'b0;
        n_next = 'b0;
        end
    else
      begin
      state_next = state_reg;
      s_next = s_reg + 'b1;
      n_next = n_reg;
      end
    end
```

UART_RX

```
data:
  if (s_tick)
    if (s_reg==15)
      begin
      s_next = 0;
      b_next = {rx,
              b_reg[7:1]};
      if (n_reg==(DBIT-1))
        state_next = stop ;
      else
        n_next = n_reg + 1;
      end
    else
      s_next = s_reg + 1;
```

```
data:
  begin
  rx_done_tick = 1'b0;

  if (!s_tick)
    begin
    state_next = state_reg;
    s_next = s_reg;
    n_next = n_reg;
    b_next = b_reg;
    end
```

Remove nested if statements.
Assign all variables.

UART_RX

```
data:
  if (s_tick)
    if (s_reg==15)
      begin
      s_next = 0;
      b_next = {rx,
              b_reg[7:1]};
      if (n_reg==(DBIT-1))
        state_next = stop ;
      else
        n_next = n_reg + 1;
      end
    else
      s_next = s_reg + 1;
```

```
else if (s_reg==15)
    begin
    s_next = 'b0;
    b_next = {rx,
            b_reg[7:1]};
    if (n_reg==(DBIT-1))
      begin
      state_next = stop ;
      n_next = n_reg;
      end
    else
      begin
      state_next =
                state_reg;
      n_next = n_reg+'b1;
      end
    end
else
  begin
  state_next = state_reg;
  s_next = s_reg + 'b1;
  n_next = n_reg;
  b_next = b_reg;
  end
```

UART_RX

```
  stop:
    if (s_tick)
      if (s_reg==(SB_TICK-1))
        begin
        state_next = idle;
        rx_done_tick =1'b1;
        end
      else
        s_next = s_reg + 1;
endcase
end
```

Remove nested if statements.
Assign all variables.

```
  stop:
    begin
    n_next = n_reg;
    b_next = b_reg;
    if (!s_tick)
      begin
      state_next = state_reg;
      rx_done_tick = 'b0;
      s_next = s_reg;
      end
    else
    if (s_reg==(SB_TICK-1))
        begin
        state_next = idle;
        rx_done_tick = 'b1;
        s_next = s_reg;
        end
    else
        begin
        state_next= state_reg;
        rx_done_tick = 'b0;
        s_next = s_reg + `b1;
        end
```

UART_RX

```
endcase
end
```

```
  default:
      begin
      state_next = state_reg;
      rx_done_tick = 'b0;
      s_next = s_reg;
      n_next = n_reg;
      b_next = b_reg;
      end

endcase
end
```

Put default values in default case, or else condition when using if statement.

Simulation Race Conditions

Simulator can have race conditions

- Not referring to gate-level race conditions
- Race conditions occur when 2 events are scheduled to happen at same simulation time and the order of execution by scheduler is not deterministic.
- Race conditions within simulator, can lead to unpredictable results, and even differing results on different simulators.
- Race conditions are hard to debug because they occur in “zero” time and don’t readily appear on waveforms.
- Race conditions can even result in an oscillation which will hang the simulator

Race example

- Writing variable from more than one block
- Final value of A depends on which block is evaluate last
- Order is not specified in standard, so is unpredictable

```
always @(posedge clk) a = 1;
always @(posedge clk) a = 5;
```

Race example

- More realistic example

```
module some_model (
input d, clock,
output reg q);

always @(posedge clock)
  q = d;

endmodule
```

```
module testbench;

DUT dut_i (d,clk,q);

initial
  begin
  @(posedge clk);
  d = 1;
  if (q != d)
    $display (“ERROR”);
  end
endmodule
```

Race example

- Initial and always blocks execute at time 0
- Value of CLK depends whether initial block was executed before always assignment was scheduled.
- Order is not specified in standard, so is unpredictable

```
initial clk = 0;
always
    clk = #5ns ~clk;
```

- Better solution

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

Worse than race condition

- It is possible to create oscillations
- Simulation continues to run, but never advances simulation time
- Occurs when 2 blocks trigger each other

```
always @(posedge A, posedge X)
  begin
  A <= 'b0;
  B <= 'b1;
  end

always @(posedge B)
  begin
  A <= 'b1;
  B <= 'b0;
  end
```

Avoiding race conditions
Clifford E. Cummings, Sunburst Design, Inc.

- When modeling sequential logic, use nonblocking assignments.
- When modeling latches, use nonblocking assignments.
- When modeling combinational logic with an always block, use blocking assignments.
  - Flops and latches use <=, combo use =
- When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.
  - Don't do this
- Do not mix blocking and nonblocking assignments in the same always block.
- Do not make assignments to the same variable from more than one always block.
- Use $strobe to display values that have been assigned using nonblocking assignments.
  - Kind of outdated
- Do not make assignments using #0 delays.
  - Guilty

Systemverilog

SV Enhancements

- Have already mentioned for synthesizable code
- Time literals can now have units
  - No requirement for `timescale

```
#5ns
```

- Enumerated types
- User defined types
  - Can declare variables as new data type
  - Introduces strict typing

```
typedef enum reg [1:0] {idle, start, data, stop}
 states;
```

- Structs and unions

SV Enhancements

- Queues
  - Variable-size ordered collection of homogenous elements

```
int      q[$];
q.push_back(10);
q.pop_front();
```

- Associative Arrays
  - Dynamic array that implements lookup table.
  - Good for modeling large, sparsely data space, such as RAMs

```
bit [31:0] RAM[integer];
RAM[‘h0000_0000]      = ‘hDEAD_BEEF;
RAM[‘hFFFF_FFFF]      = `hFACE_CAFE;
```

SV Enhancements

- $urandom, $urandom_range
  - Generate random number

```
A = $urandom(); // returns an unsigned 32-bit number
A = $urandom_range(15,0); // returns number 0 to 15
```

- randcase
  - Generate a weighted random variable

```
randcase
  1: x = 1;  // 20% of time
  3: x = 2;  // 60% of time
  1: x = 3;  // 20% of time
endcase
```

SV Enhancements

- randsequence
  - Generate a random sequence of events
  - Conditional branching based on prior results

```
randsequence( main )
  main   : first second done ;

  first  : add := 3 | dec := 2 ;
  second : pop | push ;
  done   : { $display("done"); } ;

  add    : { $display("add"); } ;
  dec    : { $display("dec"); } ;
  pop    : { $display("pop"); } ;
  push   : { $display("push"); } ;
endsequence
```

SV Enhancements

- fork … join_none
- fork … join_any
- disable fork;

Example: fork … join_any / disable fork

```
@(posedge some_signal);
…
```

Above could hang simulation if signal never transitions from 0 to 1.

```
fork
  begin
  @(posedge some_signal);
  error = ‘b0;
  end

  begin
  #100ns;
  error = ‘b1;
  end
join_any
disable fork;

if (error)
  $display (“ERROR: No signal”);
…
```

Using fork…join, a timeout error can be added

SV Enhancements

- Classes
- Packages
- Assertions
- Coverage

References

Resources

- What's New in Verilog 2001
  - http://www.sutherland-hdl.com/papers/2000-HDLCon-presentation_Verilog-2000.pdf
- Online Verilog-1995 Quick Reference Guide
  - http://www.sutherland-hdl.com/online_verilog_ref_guide/vlog_ref_top.html
- Standard Gotchas, Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know!
  - http://www.sutherland-hdl.com/papers/2006-SNUG-Boston_standard_gotchas_presentation.pdf
- Programming Gotchas
  - http://www.deepchip.com/items/0466-06.html
- full_case & parallel_case the Evil Twins of Verilog Synthesis
  - http://www.deepchip.com/posts/0332.html
- SystemVerilog Saves the Day—the Evil Twins are Defeated! "unique" and "priority" are the new Heroes
  - http://www.sutherland-hdl.com/papers/2005-SNUG-paper_SystemVerilog_unique_and_priority.pdf

Resources

- **How to raise the RTL abstraction level and design conciseness with SystemVerilog - Part 1**
  - http://www.eetimes.com/design/eda-design/4015170/How-to-raise-the-RTL-abstraction-level-and-design-conciseness-with-SystemVerilog--Part-1
- **Verilog always@ Blocks**
  - http://www.scribd.com/doc/30553026/Verilog-Notes-John-Wawrzynek
- **Race Condition**
  - http://www.testbench.in/TB_16_RACE_CONDITION.html
- **Scheduling Semantics**
  - http://www.systemverilog.in/scheduling-semantics.php
- **Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!**
  - http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA.pdf
- **SystemVerilog enhancements for all chip designers**
  - http://www.eetimes.com/electronics-news/4154793/SystemVerilog-enhancements-for-all-chip-designers