Proposal: replace Annex B with the following:

Annex B
(informative)

Functional Mismatches

This annex describes certain situations where functional differences may arise between the RTL model and its synthesized netlist.

B.1 Non-deterministic behavior

The Verilog language has some inherent sources of non-determinism. In such cases, there is a potential for a functional mismatch. For example, statements without time-control constructs (# and @ expression constructs) in behavioral blocks do not have to be executed as one event. This allows for interleaving the execution of always statements in any order. In the following example, the behavior can be interpreted such that it is free to assign a value of either 0 or 1 to register B:

```
always @(posedge CLOCK) begin
  A = 0;
  A = 1;
end

always @(posedge CLOCK) begin
  B = A;
end
```

In this case, the synthesis tool is free to assign either 0 or 1 to register B as well, causing a potential functional mismatch.

When Verilog non-deterministic behavior permits different results from two different Verilog simulators that are both IEEE compliant, this is known as a race condition. There are common Verilog guidelines that help to insure that a simulation race condition will not happen. This document does not describe these common guidelines.

Most Verilog RTL models that include race conditions can cause a mismatch between pre-synthesis and post-synthesis simulations and should be avoided.

B.2 Pragmas

Pragmas must be used wisely since they can affect how synthesis interprets certain constructs. ~~For example, if a user species the parallel case directive but the case items are not independent, the behavior of the synthesis results may not match that of the RTL model.~~ A pragma that directs the synthesis tool to do something different than what the simulator does, should either be avoided or used with great caution. Some problematic synthesis pragmas are described in the following sections.

(synthesis, full_case *)

The full_case attribute directs the synthesis tool to treat all undefined case items as synthesis don’t care cases. Verilog simulators ignore undefined case items.

In the following decoder example, the simulator correctly sets the decoder outputs to 0 when the enable signal is low. Because the pragma (* synthesis, full_case *) has been added to this model, the synthesis tool recognizes that only four of eight possible case items have been defined and treats all other cases as don't care cases. With the don't care cases defined, the synthesis tool recognizes that the output is a don't care whenever the enable is low; therefore, the enable is a don't care so the enable is optimized out of the design, changing the functionality of the design. This will cause a mis-match between the pre-synthesis and post-synthesis simulations

```
module decode4_fc (output reg [3:0] y, input [1:0] a, input en);
  always @* begin
    y=4'b0;
    (* synthesis, full_case *)
    case ({en, a})
      3'b1_00: y[a]=1'b1;
      3'b1_01: y[a]=1'b1;
      3'b1_10: y[a]=1'b1;
      3'b1_11: y[a]=1'b1;
    endcase
  end
endmodule
```

(synthesis, parallel_case *)

The parallel_case attribute directs the synthesis tool to test each case item every time the case statement is executed. Verilog simulators only test case items until there is a match between the case expression and a case item. Once a case item is matched to the case expression, the case item statement is executed and an implied break causes the simulator to ignore the remaining case items.

In the following enable-decoder example, if the en = 2'b11, the simulation will execute the first case item statement, skipping the second, while the synthesis tool will execute the first two case item statements. This will cause a mis-match between the pre-synthesis and post-synthesis simulations.

```
module endec_pc (
  output reg       en_mem, en_cpu, en_io,
  input      [1:0] en);
  always @* begin
    en_mem=1'b0;
    en_cpu=1'b0;
    en_io =1'b0;
    (* synthesis, parallel_case *)
    casez (en)
      2'b1?:   en_mem=1'b1;
      2'b?1:   en_cpu=1'b1;
      default: en_io =1'b1;
    endcase
  end
endmodule
```

`ifdef Usage

Conditionally compiling or omitting compilation of Verilog source code based on synthesis and simulation should be used with extreme caution.

In the following model, if the SYNTHESIS macro definition is not defined, the memory will be modeled with synthesizable RTL code. If the SYNTHESIS macro is defined, a vendor xram device will be instantiated into the design.

```
module ram_ifdef (
  output [7:0] q,
  input  [7:0] d,
  input  [6:0] a,
  input        clk, we);
  `ifndef SYNTHESIS
    // RTL model of a ram device for pre-synthesis simulation
    reg [7:0] mem [0:127];
    always @(posedge clk) if (we) mem[a] <= d;

    assign q = mem[a];
  `else
    xram ram1 (.dout(q), .din(d), .addr(a), .ck(clk), .we(we));
  `endif
endmodule

// Vendor ram model
module xram (
  output [7:0] dout,
  input  [7:0] din,
  input  [6:0] addr,
  input        ck, we);
  // Vendor ram model implementation
endmodule
```

Although selecting between a modeled RTL core device and an instantiated core device is one reason that conditional compilation would be used, it is the responsibility of the user to insure that there are not simulation-functional differences in the models that would cause a mis-match between pre-synthesis and post-synthesis simulations.

In the following model, the conditionally compiled code will clearly cause a mis-match between pre-synthesis and post-synthesis simulations. When the SYNTHESIS macro is defined, the y output is set equal to the bitwise-or of the a and b inputs. When the SYNTHESIS macro is not defined, the y output is set equal to the bitwise-and of the a and b inputs.

```
module and2_ifdef (output y, input a, b);
  `ifdef SYNTHESIS
    assign y = a | b;
  `else
    assign y = a & b;
  `endif
endmodule
```

Incomplete Sensitivity List

An incomplete sensitivity list on a combinational always block will typically cause a mis-match between pre-synthesis and post-synthesis simulations.

The following model is coded correctly to model a 2-input and gate. Both and-gate inputs are listed in the combinational sensitivity list. There will be no mis-match between pre-synthesis and post-synthesis simulations using this model.

```
module myand1b (output reg y, input a, b);
  always @(a or b)
    y = a & b;
endmodule
```

In the following model, the b-input is missing from the combinational sensitivity list. This model typically synthesizes to a 2-input and gate but does not simulate correctly whenever the b-input changes. This will cause a mis-match between pre-synthesis and post-synthesis simulations.

```
module myand1c (output reg y, input a, b);
  always @(a)
    y = a & b;
endmodule
```

In the following model, both inputs are missing from the combinational sensitivity list. This model typically synthesizes to a 2-input and gate but does not simulate correctly. If this model is simulated, the simulator will hang as it loops in zero time, continuously executing the statement, y = a & b. This will cause a mis-match between pre-synthesis and post-synthesis simulations (the pre-synthesis simulation hangs)..

```
module myand1d (output reg y, input a, b);
  always
    y = a & b;
endmodule
```

A feature added to the Verilog-2001 Standard is the @* combinational sensitivity list. This shorthand feature was added to reduce redundant typing and to greatly reduce the number of errors that are introduced by coding incomplete sensitivity lists.

```
module myand1a (output reg y, input a, b);
  always @*
    y = a & b;
endmodule
```

Assignment statements mis-ordered

If assignment statements are mis-ordered in a combinational always block, synthesis tools typically build the logic as if the statements had been ordered correctly but the pre-synthesis simulation will be wrong.

The following model is coded correctly to model an and-or gate where the a and b inputs are anded together, and the result is ored with the c input. There will be no mis-match between pre-synthesis and post-synthesis simulations using this model.

```
module andor1a (output reg y, input a, b, c);
  reg tmp;

  always @* begin
    tmp = a & b;
    y   = tmp | c;
  end
endmodule
```

The following model is coded incorrectly to model an and-or gate where the a and b inputs are anded together, and the result is ored with the c input. The pre-synthesis simulation will not correctly update the

ored output y after changes on the a and b inputs. There will be a mis-match between pre-synthesis and post-synthesis simulations using this model.

```
module andor1b (output reg y, input a, b, c);
  reg tmp;

  always @* begin
    y   = tmp | c;
    tmp = a & b;
  end
endmodule
```

Flip-flop with both asynchronous reset and asynchronous set

There is a small problem with the pre-synthesis model of a flip-flop with both asynchronous reset and asynchronous preset signals. The correct synthesizable model for this type of flip-flop is shown below.

```
module dffaras (output reg q, input d, clk, rst_n, set_n);
  always @(posedge clk or negedge rst_n or negedge set_n)
    if      (!rst_n) q <= 1'b0;
    else if (!set_n) q <= 1'b1;
    else             q <= d;
endmodule
```

The problem occurs when both reset and preset are asserted at the same time and reset is removed first. When reset is removed (posedge rst_n), the always block is not activated. This means that the output will continue to drive the reset output to ‘0’ until the next rising clock edge. A real flip-flop of this type would immediately drive the output to ‘1’ because the set_n signal is an asynchronous preset. This potentially could cause a mis-match between pre-synthesis and post-synthesis simulations using this model.

It should be noted that it is rare to design flip-flops with both asynchronous set and asynchronous reset, it is even more rare to use this type of flip-flop in a design where both reset and preset are permitted to be asserted at the same time and even more rare to allow reset to be removed before the preset is removed. It is estimated that fewer than 1% of all designs would ever be subject to this mis-match.

For the rare designs that do require assertion of both reset and preset and must permit removal of reset first, the following conditionally compiled code can be added to the above simulation model to correct the simulation problem. Note that this is only a rare simulation problem, not a synthesis problem.

```
`ifndef SYNTHESIS
  always @(rst_n or set_n)
    if (rst_n && !set_n) force q = 1'b1;
    else                 release q;
`endif
```

Functions

In general, synthesis tools always synthesize Verilog functions to combinational logic, even if the simulation behaves like a latch. The following is a correct model for a simple D-latch.

```
module latch1a (output reg y, input d, en);
  always @*
    if (en) y <= d;
endmodule
```

If the latching code is placed into a Verilog function, as shown in the following model, the simulation still behaves like a function but synthesis tools generally infer combinational logic causing a mis-match between pre-synthesis and post-synthesis simulations.

```
module latch1b (output reg y, input d, en);
  always @*
    y <= lat(d, en);

  function lat (input d, en);
    if (en) lat = d;
  endfunction
endmodule
```

Using Verilog functions should be done with caution since there is no warning from a synthesis tool that latch-behavior coding will be synthesized to combinational logic.

Casex

The Verilog casex statement treats all z , x , and ? bits as don't cares, whether they appear in the case expression or in the case item being tested. In the following model, if the en (enable) goes unknown during simulation, the en_mem output will be driven high. In a synthesized gate-level model, the outputs would most likely go unknown indicating a design problem. This is a mis-match between pre-synthesis and post-synthesis simulations.

```
module endec_x (output reg en_mem, en_cpu, en_io, input [1:0]
en);
  always @* begin
    en_mem=1'b0;
    en_cpu=1'b0;
    en_io =1'b0;
    casex (en)
      2'b1?:   en_mem=1'b1;
      2'b01:   en_cpu=1'b1;
      default: en_io =1'b1;
    endcase
  end
endmodule
```

It is too easy for a pre-synthesis simulation to have startup problems that cause signals to go unknown and to be treated as a don't care by the casex statement. For this reason, in general, the casex statement should be avoided for synthesis RTL coding.

Casez

The Verilog casez statement treats all z and ? bits as don't cares, whether they appear in the case expression or in the case item being tested. In the following model, if both en (enable) bits go high during simulation, the en_mem output will be driven high. In a synthesized gate-level model, the outputs would most likely go unknown indicating a design problem. This is a mis-match between pre-synthesis and post-synthesis simulations.

```
module endec_z (output reg en_mem, en_cpu, en_io, input [1:0]
en);
  always @* begin
    en_mem=1'b0;
```

```
    en_cpu=1'b0;
    en_io =1'b0;
    casez (en)
      2'b1?:   en_mem=1'b1;
      2'b01:   en_cpu=1'b1;
      default: en_io =1'b1;
    endcase
  end
endmodule
```

It is unlikely (but not impossible) that a pre-synthesis simulation would experience stray high impedance values on most design signals. For this reason, in general, the casez statement is safe to use but, noting the above potential for problems, they should be used with caution.

Making ‘X’ Assignments

Making a Verilog x-assignment to a signal tells the simulator to treat the signal as having an unknown value and tells the synthesis tool to treat the signal as a don’t care. The synthesis tool will build a gate-level design using optimized gates that will not drive an unknown output on the signal. This means there is a mis-match between pre-synthesis and post-synthesis simulations for all x-assigned signals.

In the following 3-to-1 mux model, the output is initialized to an x-value and then updated based on the value of the sel signals. This design assumes that sel should never be equal to 2’b11. If the pre-synthesis simulation permits the sel signals to briefly pass through the 2’b11 pattern, the simulation will drive ‘X’ to the output until the sel signals take on a valid select-pattern.

```
module mux3_x (output reg y, input [2:0] a, input [1:0] sel);
  always @* begin
    y = 1'bx; // synthesis "don't-care"
    case (sel)
      2'b00: y=a[0];
      2'b01: y=a[1];
      2'b10: y=a[2];
    endcase
  end
endmodule
```

The x-output can be useful to help find bugs in the design during pre-synthesis simulations. It can also help direct the synthesis tool to optimize the design based on a don’t care assignment. If the pre-synthesis simulation tests the output while it is unknown and if that unknown output is compared to a post-synthesis simulation, there will be a mis-match.

In general, the unknown output is short (unless there is a real design problem) and the output will be tested closer to a clock edge when the signal has had time to propagate to a known and correct value. Designers should just recognize that there is potential for a mis-match between pre-synthesis and post-synthesis simulations using this technique.

Assignments in variable declarations

Initializing variables in the variable declaration was added to Verilog-2001. Making assignments in the declaration forces the signal to a known value for pre-synthesis simulations. In general, no such initialization occurs in an actual gate level design. This can cause a mis-match between pre-synthesis and post-synthesis simulations and in general should be avoided.

```
// module dff_init (output reg q=1'b0, input d, clk, rst_n);
module dff_init (q, d, clk, rst_n);
  output q;
  input  d, clk, rst_n;
  reg    q=1'b0;
  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Timing delays

Synthesis tools ignore time delay in a model. Adding time delays to a Verilog pre-synthesis simulation can cause a mis-match between pre-synthesis and post-synthesis simulations and in general should be avoided.

In the following delay-line model, the latch enable output is delayed in a pre-synthesis simulation but the delay will be removed from the post-synthesis implementation, potentially causing a delayed latch signal to be enabled too soon.

```
`timescale 1ns/1ns
module delay1 (output reg latchendly, input latchen);
  always @*
    latchendly <= #25 latchen;
endmodule
```

Adding delay elements to a synthesized model typically requires instantiation of the delay element in the pre-synthesis RTL model.