mirafra
TECHNOLOGIES

lint_rtl goals

- ❑ This goal checks basic connectivity issues in the design, such as floating inputs, width mismatch, etc.
  - ▪ These checks should be run after every change in RTL code prior to code check-in.
- ❑ This goal checks simulation issues in the design, such as
  - incomplete sensitivity list
  - incorrect use of blocking / non-blocking assignments
  - potential functional errors
  - possible simulation hang cases, and simulation race cases
    - ▪ These checks should be run, and reported messages should be reviewed prior to all simulation runs.
- ❑ This goal identifies the structural issues in the design that affect the post implementation functionality or performance of the design. Examples include multiple drivers, high fan-in mux, and synchronous/asynchronous use of resets.
  - ▪ These checks should be run once every week and before handoff to implementation.
- ❑ This goal reports unsynthesizable constructs in the design and code which can cause RTL vs. gate simulation mismatch.
  - ▪ These checks should be run twice a week, and before handoff to synthesis team.

Important issues addressed in Lint

- Un-synthesizable constructs
- Unintentional latches
- Unused declarations
- Multiple driver and undriven signals
- Race conditions
- Incorrect usage of blocking & non-blocking assignments
- Incomplete assignments in sub-routines
- Case statement style issues
- Set & Reset conflicts
- Out-of-range indexing

Important issues addressed in Lint

mirafra TECHNOLOGIES

- Differences between simulation and synthesis semantics.
- Opportunities to improve simulation performance.
- Probable simulation errors.
- Chances of matching gate level simulations with RTL simulations.
- Coding guidelines.
- FSM state reachability and coding issues.
- Network and connectivity checks for clocks, resets, and tri-state driven signals.
- Module partitioning.
- Tool flow issues in the upcoming design cycle stages.
- Possible synthesis issues. (e.g., unintended latches inferred or combo loops).

mirafra
TECHNOLOGIES

Linting tools

- Some of the available tools in the market to do Linting are:
  - Spyglass (Synopsys)
  - Realintent
  - LEDA (Synopsys)
  - SureLint

Rules in Spyglass Lint

mirafra TECHNOLOGIES

- Array Rules
- Case Rules
- Reset Rules
- Clock Rules
- Usage Rules
- Tristate Rules
- Assign Rules
- Function-Task Rules
- Delay Rules
- Latch Rules
- Instance Rules
- Synthesis Rules
- Expression Rules
- Multiple Driver Rules
- Simulation Rules
- Event Rules
- Loop Rules
- Elab Rules

mirafra
TECHNOLOGIES

Examples of some major violations

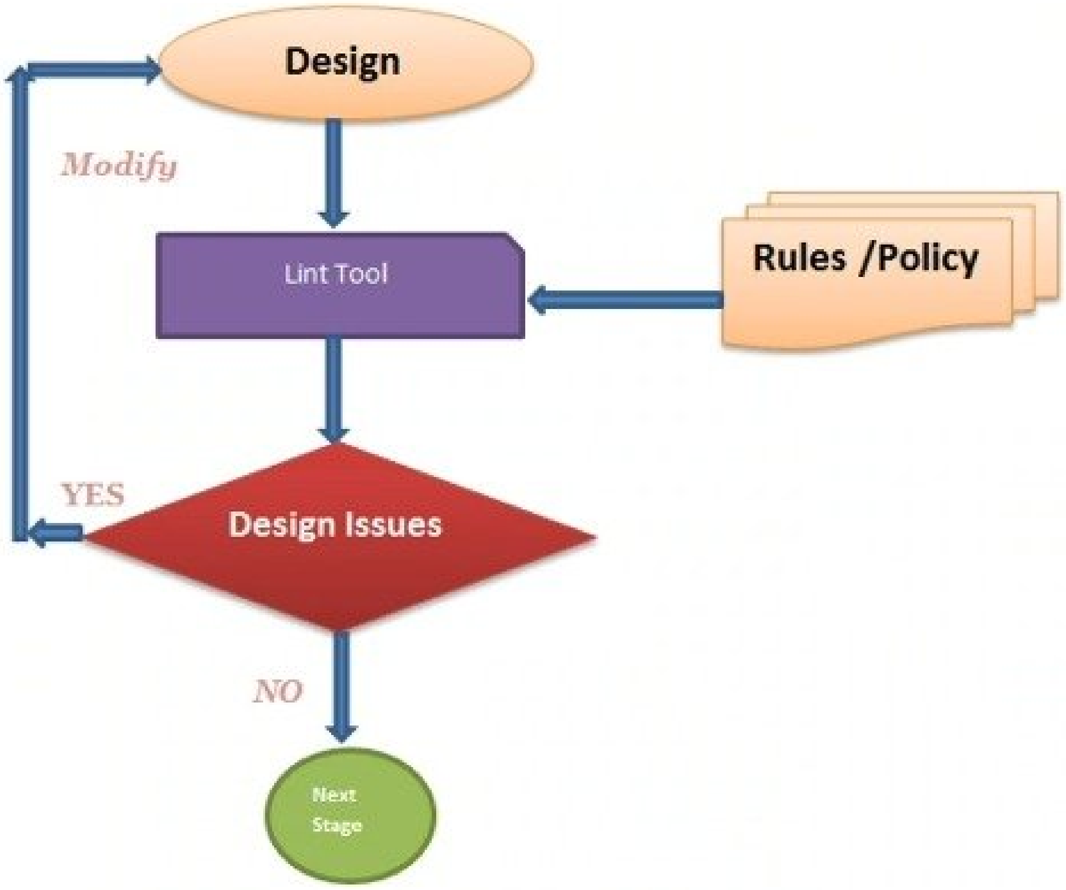
Introduction

- Static code analysis checks used to flag
  - Programming errors
  - Stylistic errors
  - Suspicious constructs
- Need for linting?
  - Synthesized output of RTL code is input to multiple stages of VLSI design flow
    - Any coding issues, can result multiple iterations of RTL update, synthesis
- Linting checks the cleanliness and portability of HDL code for the later stages of VLSI design flow.

mirafra
TECHNOLOGIES

How does it work?

- There are some set of rules defined in the lint tool. Rule means a condition that has to be checked on your design. User can enable and disable the required rules as per his/her requirement. Once these rules are run on the design, and if the design source code does not conform to a rule, violation will be reported.



Example: 1

```
module example_1 (a, b, c);
    input a;
    input b;
    output c;
    assign c = a + b;
endmodule
```

- This example is completely ok with compiler. Compiler does not generate warning.
- In this code by default the output variable ‘c’ is a wire. It is not necessary to declare it as a wire.
- But some (not every) synthesis tools may need it to be declared it as a wire.
- If Lint tool applied on the code before the synthesis , it points the above mistake.
- This is just simple example.

mirafra TECHNOLOGIES

Example: 2

```
module example_2 (out, in);
output [1:0] out;
input [2:0] in;
assign out = in[1:0];
endmodule
```

- In this example, the Spyglass Lint tool reports a violation because all bits of the array in are not read.

W111: Not all elements of an array are read

mirafra TECHNOLOGIES

Example 3: Latch inference

```
always@(in1 or in2 or in3 or mem)
begin
case(in3)
2'b00 ; out[0] = in1 || mem[0];
2'b01 : out[1] = in1 && mem[1];
2'b11 : out[2] = in2 && mem[3];
endcase
end
```

default

❑**Fix:**

▪add case clauses (or a default clause) for those cases that are not covered and specify the appropriate behavior in that default clause.

▪add pragma full_case to the case statement, implying you know that the remaining cases can never happen.

- In this example where the case construct does not have all possible cases described (case in3=2'b10 is not described) and also does not have a default clause

❑ ***Consequences of Not Fixing:***

- A latch may be inferred, if the case construct does not describe all possible states and also does not have a default clause.
- While you may have intended to infer a latch, it is advisable to examine such cases to avoid creation of unexpected logic.

W69: Ensure that a Case statement specifies all possible cases & has a default clause

Inferred latches (contd ....)

mirafra TECHNOLOGIES

- It is a major issue that can cause a design to malfunction.
- This happens when a Verilog code does not account for the value held by a variable at certain times.
- It occurs due to incomplete usage of certain constructs.
- Example: ***If condition without else block***

```
always @ (posedge clk)
if (a > b)
c <= a;
```

- If this is all the given code, how do we know what happens to “c” when “a” > “b” condition fails?
- This is called ***inferred latch.*** When the tool infers a latch under such circumstances, then the previous value of the variable is maintained.

mirafra
TECHNOLOGIES

Inferred latches (contd ...)

- To avoid the tool making such an inference, it is always desirable for a designer to explicitly mention what happens in such a case.
- Even if the value is maintained, it can be explicitly coded as shown:

```
always @ (posedge clk)
if (a > b)
c <= a;
else
c <= c;
```

- This will ensure that the latches are not 'inferred' by the tool.

mirafra
TECHNOLOGIES

Example 4: Set & Reset conflicts

```
always @( posedge C_SCLK1 or posedge
    C_SRST1 )
begin
if (C_SRST1)
Q1 = 2'b11;
else
Q1 = DataIn[1:0];
end
always @( posedge C_SCLK1 or negedge
    C_SRST1 )begin
if (!C_SRST1)
Q2 = 2'b11;
else
Q2 = DataIn[1:0]:
end
```

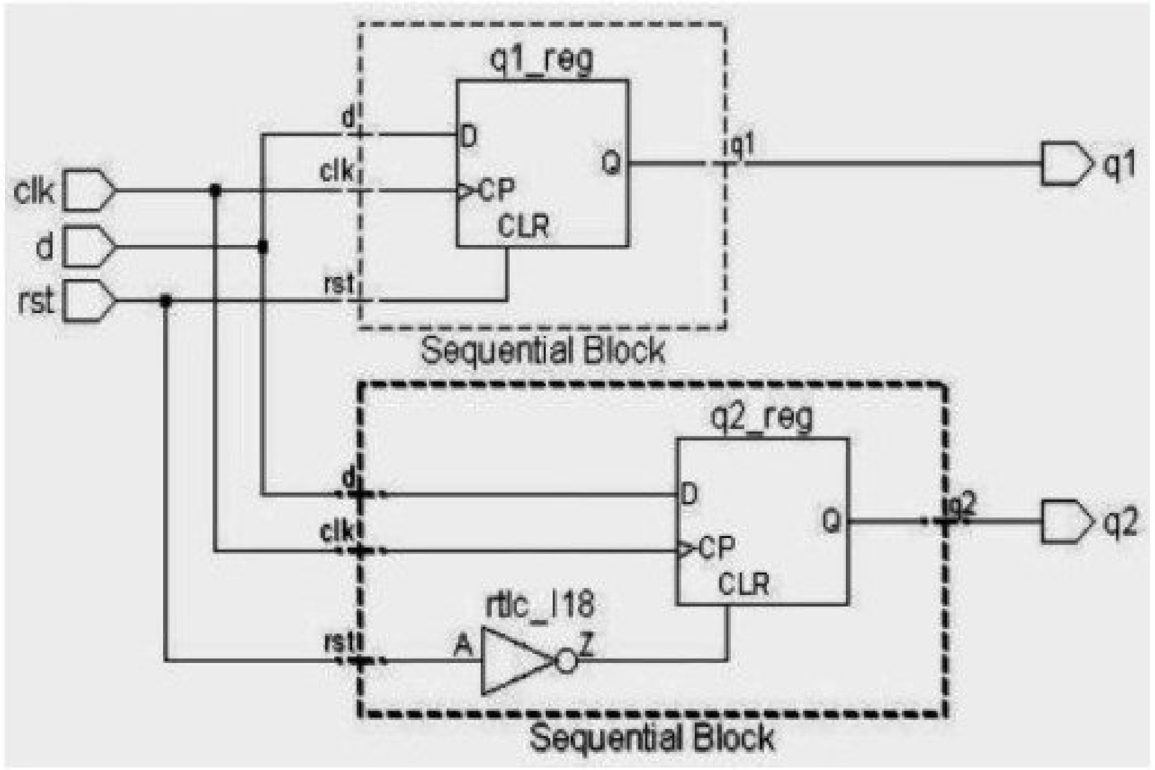
❑ ***Consequences of Not Fixing***

- When both polarities of reset/set signal are used, one logic block always remain in a reset/set state. If a reset/set signal is high, the logic that operates on positive reset/set polarity would be reset/set.
- However, when the reset/set signal is de-asserted, some other portion of logic would be reset/set. The usage leads to mutually exclusive blocks of logic.

W392: Reports reset or set signals used with both positive & negative polarities within the same design unit

mirafra TECHNOLOGIES

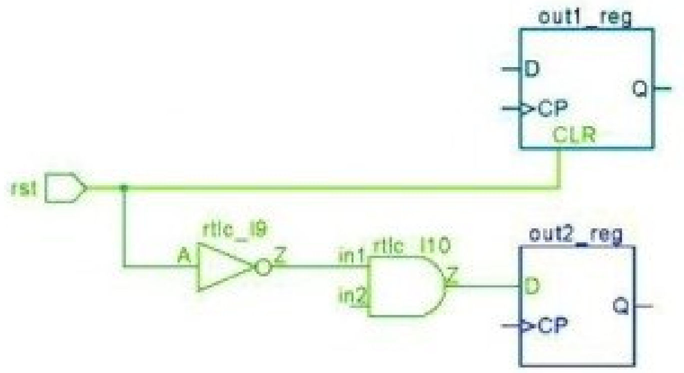
Example 4: Set & Reset conflicts



Example 5: Sync & Async reset conflicts

```
always @(posedge clk or
   posedge rst)
begin
if(rst) out1 <= 0;
else out1 <= in;
end
always@(posedge clk)
begin
if(rst) out2 <= 0;
else out2 <= in;
end
```

- In this example where signal rst is used both as synchronous reset as well as asynchronous reset



W448: Reset/set is used both synchronously and asynchronously

mirafra
TECHNOLOGIES

Example 6: Pos-edge & Neg-edge conflicts

```
module example_6 (out1, out2, in1, in2, clk);
input in1, in2, clk;
output out1, out2;
reg out1, out2;
always @(posedge clk)
out1 = in1;
always @(negedge clk)
out2 = in2;
endmodule
```

❑ ***Consequences of Not Fixing***

- As a result of using both the edges, the behavior of that module gets dependent on the duty cycle of the clock.

W391: Reports modules driven by both edges of a clock

Example 7: Undriven nets

mirafra TECHNOLOGIES

```
module example_7(in1, clk, out1);
input [2:0] in1;
input clk;
output [2:0] out1;
reg [2:0] out1;
wire [2:0] net;
always@(posedge clk)
out1 <= in1 && net;
endmodule
```

- ***Consequences of Not Fixing***

- Reading an uninitialized value may result in undriven nets in post synthesis.

W123: Identifies the signals and variables that are read but not set

mirafra
TECHNOLOGIES

Example 8: Data Loss

```
module example_8(clk, out);
input clk;
output out;
reg out;
always @(posedge clk) begin
out = 1'b101;
if(out == 2'b0101) begin
end
end
endmodule
```

❑ ***Consequences of Not Fixing***

- When constant value is wider than the width of the constant, it results in truncation of extra bits. This in turn leads to data loss and unexpected code behavior.
- //Constant 1'b101 will be truncated
- //Constant 2'b0101 will be truncated

W19: Reports the truncation of extra bits

mirafra
TECHNOLOGIES

Example 9: Blocking assignment usage

```
module example_9 (clk, reset, d, q);
input clk, reset, d;
output q;
reg q;
always @(posedge clk or negedge
    reset)
begin
if (!reset)
q = 1'b0;
else
q = d;
end
endmodule
```

❑ ***Consequences of Not Fixing***

- When a blocking assignment is used in a sequential block, inherent sequence of operation is implied in simulation. However, the synthesized hardware may behave in a concurrent fashion. Therefore, there is no assurance that the gate-level simulations match RTL level simulations.

❑ ***Fix***

- Use only non-blocking assignments in sequential blocks.

W336: Blocking assignment should not be used in a sequential block (may lead to shoot through)

mirafra
TECHNOLOGIES

Example 10: Blocking & Non-blocking assignment usage

Case 1

```
out1 <= in1 & in2;
out2 = in3 & in4;
```

Case 2

```
out1 <= in1 & in2;
out2 <= in3 & in4;
```

Case 3

```
out1 = in1 & in2;
out2 <= in1 & in2;
```

Case 4

```
reg [3:0] reg_a;
reg_a <= 1 ;
reg_a = 2 ;
```

❑ ***Consequences of Not Fixing***

- Not fixing the violation may result in unexpected code behavior.
- Synthesis require that the same variable or signal should not be assigned in both blocking mode and non-blocking mode.
- Otherwise, the pre- and post-synthesis simulation behavior of the RTL and gate-level design may differ.

W414: Reports non-blocking assignments in a combinational block

mirafra
TECHNOLOGIES

Example 11: Latch inference

```
process (reset, d)
begin
if (reset = '1') then
q <= d;
end if;
end process;
```

- ❑ Latch inferred for signal 'q'

W18: Do not infer latches

mirafra
TECHNOLOGIES

Example 12: Width mis-match

```
inst IN1 (.in1(a[2:0]+b[2:0]));

// Width of expression,
   a[2:0]+b[2:0], will be 4 (7+7=14,
   4 bits wide)
```

- ***Consequences of Not Fixing***
  - Connection of a bus which is wider than the port: Excess bits are ignored for the input bus and floated for the output bus.
  - Connection of a bus which is narrower than the port: Missing bits are driven unknown for the input bus and ignored for the output bus.

W110: Identifies a module instance port connection that has incompatible width as compared to the port definition

Example 12: Width mis-match (contd ....)

- One of the more common issues that can be discovered only by lint - ***Mismatches in the width of signals being used on LHS as well as RHS. Example:***

```
wire [3:0] a;
wire [7:0] b;
assign a = b;
```

  - This will cause a width mismatch.
  - The width of signal “a” is 4-bits while the width of “b” is 8-bits.
  - When we assign “b” to “a”, “a” will be able to store only the least significant 4-bits of “b”, ie., b[3:0].
  - The remaining 4-bits of “b” will be lost.
  - Solution – To avoid potential data loss, we should ensure that the signals used in the LHS as well as RHS of an expression is of the same width.

Example 13: Non-synthesizable constructs

```
module example_13(in1, in2, z);
input in1;
input in2;
output z;
real r = 0.025;
assign z = r + in1 + in2;
endmodule
```

❑ ***Consequences of Not Fixing***

- Objects with real values have no physical equivalent and therefore may not be synthesizable by some synthesis tools.
- Real variables should only be used in testbenches and simulation models.

W294:Reports real variables that are unsynthesizable

mirafra
TECHNOLOGIES

Example 13: Non-synthesizable constructs (contd ....)

```
module example_14(in1, in2, clk, out1);
input [1:0] in1, in2;
input clk;
output [1:0]out1;
reg [1:0]out1;
wire [1:0] count;
assign count = (in1 === 1'b1)? 2'b10 : 2'b01;
always @(posedge clk)
begin
out1 <= (in2 !== count) ? in1 : in2;
end
endmodule
```

- Case equal operator (===) and case not equal (!==) operators may not be synthesizable

W339a:
Case equal operator (===) and case not equal (!==) operators may not be synthesizable

mirafra
TECHNOLOGIES

Example 13: Non-synthesizable constructs (contd ....)

```
module example_15(in1,clk,out1,out2);
input in1,clk;
output reg out1,out2;
initial
begin
out1 = 1'b0;
out2 = 1'b0;
end
endmodule
```

❑ ***Consequences of Not Fixing***

- The initial constructs have no physical equivalent. Therefore, such constructs are Unsynthesizable by some synthesis tools.

W430: The "initial" statement is not synthesizable

Example 16: Multiple drivers

```
always @(*) begin
multi_dr <= a & b;
end


always @(*) begin
   multi_dr = c & d;
end
```

W553: Same signal is driven by different combinational blocks

❑ ***Consequences of Not Fixing***

- A net that is not a tristate net should have a unique driver. If such nets are driven by more than one net, there is a possibility of an electrical conflict.
- For example, consider a net driven by outputs of two different AND gates. If one output is 0 and other output is 1, the driven net is in a non-digital state.
- Such cases lead to excessive current flows from one AND gate to another AND gate causing chip electrical failures, causing chip malfunction.
- The effect of a non-digital net can also propagate to its fan-out points. For such fan-out points, the net behaves as if it was floating resulting in further chip electrical and functional failures.

mirafra
TECHNOLOGIES

Example 17: Incorrect Sensitivity list

```
module example_17 ( in1, in2, out1 );
input in1, in2;
output out1;
reg out1;
always @(in1)
begin
out1 = in1&in2;
end
endmodule
```

W122:A signal is read inside a combinational always block but is not included in the sensitivity list (Verilog)

❑ ***Consequences of Not Fixing***

- Signals missing from the sensitivity list can lead to a mismatch between the pre- and post-synthesis simulations.
- The simulation only evaluates the changes in the combinational logic when the signals in the sensitivity list change before synthesis.
- The synthesis process generates a logic which reads all the required values, whether they are present in the sensitivity list or not.
- Effectively, the missing signals are added to the sensitivity list. Therefore, the post-synthesis simulation results are different from the pre-synthesis simulation results.

mirafra TECHNOLOGIES

Example 18: Incorrect Sensitivity list (contd ....)

```
module example_18(out1,inp1);
output out1;
input inp1;
reg out1;
always@(inp1 or out1)
begin
out1 <= inp1;
end
```

❑ ***Consequences of Not Fixing***

- Updating a signal from sensitivity in the same always block may result in erroneous simulation results.

W502: Ensure that a variable in the sensitivity list is not modified inside the always block

Example 20

// Top-level SGDC file

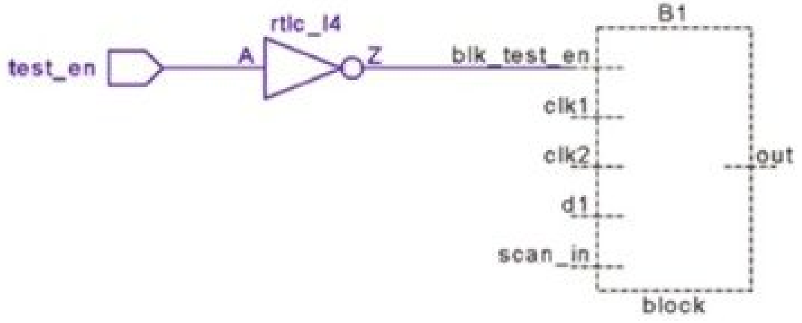
```
current_design top
set_case_analysis -name
    top.test_en -value 0
```

// Block-level SGDC file

```
current_design block
set_case_analysis -name
    block.blk_test_en -value 0
```

❑ ***Consequences of Not Fixing***

- If you do not fix this violation, the design may not operate in the desired mode.



Example 21: Combinational loops

- They are strictly not allowed in Verilog. Example of a combinational loop snippet:

```
always @ *
a = a & b;
```

  - This computes the AND operation of "a" with "b" and sends it back to "a". This will lead to an infinite loop, completely occurring at zero simulation time.
- Incase, we really need to perform this operation, we need to use non-blocking assignments:

```
always @ (posedge clock)
a <= a & b;
```

- This ensures that "a" AND "b" is computed first, and then updated to "a" only in the next clock cycle.
  - Each clock cycle, "a" AND "b" is computed and the result is updated in "a" in the following clock cycle.

mirafra
TECHNOLOGIES

Summary

- Unwanted latch inference
- Set and reset conflicts
- Synchronous & Asynchronous reset conflicts
- Pos-edge and Neg-edge conflicts
- Undriven nets
- Data loss
- Blocking assignment usage
- Blocking & Non-blocking assignment usage
- Width mis-match
- Non-synthesizable constructs
- Multiple drivers
- Incorrect sensitivity list
- Out of range indexing
- Combinational loops
- Conflicts in SGDC files