SYNOPSYS®

CUSTOMER EDUCATION SERVICES

SystemVerilog for RTL Design Workshop

Lab Guide

50-I-054-SLG-005   2019.03

Synopsys Customer Education Services
690 E. Middlefield Road
Mountain View, California 94043

Workshop Registration: https://training.synopsys.com

Copyright Notice and Proprietary Information

Document Order Number: 50-I-054-SLG-005
SystemVerilog for RTL Design Workshop Lab Guide

1

User Logic Intent

Learning Objectives

After completing this lab, you should be able to:

- Resolve synthesis/simulation mismatch with new SystemVerilog construct

- Avoid unintentional latches with new SystemVerilog construct

- Implement SystemVerilog enum data type to create better self-documenting state machine code

Lab Duration:
30 minutes

Lab Overview

To implement high Quality of Result (QoR) RTL design, one must first make sure that the RTL code written is interpreted by the synthesis tool as intented.

In this lab, you will see that one can better achieve this goal with SystemVerilog.

In the terminal window, you will find a **labs** directory.
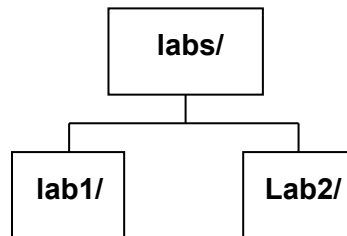
Figure 1. **Lab Directory Structure**

For each individual lab, you will work in the specified lab directory.

The general work flow for each section of this lab is illustrated as follows.
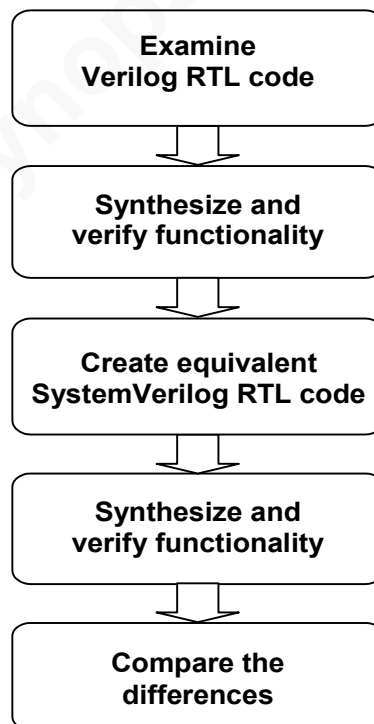
Figure 2. **Lab 1 Flow Diagram**

RTL/Gate Simulation Mismatch

Task 1.   Go into Working Directory

1.   Go into working directory:

```
> cd labs/lab1
```

Task 2.   See the Effects of Incomplete Event List RTL

A common problem associated with Verilog is that synthesizer ignores the event list whereas simulator obeys them.

For this first section, you will run simulation on a Verilog RTL code and log the result. Then, synthesize the RTL and run the simulation at the gate level. Comparing the result of RTL v.s. gate level simulation, you will see that they do not match.

1.   Take a look at the Verilog file called **mismatch.sv**

```
> less rtl/mismatch.sv
```

Note that the signal **C** is left off the Verilog event list:

```
module mismatch (...);
  ...// code not shown
  // This is the Verilog behavioral code
  always@(A, B) begin
    D = (A & B) | C;
  end
  ...
endmodule
```

Missing **C** in sensitivity

2.   Compile and simulate this RTL code

```
> make sim rtl=mismatch
```

In the opened waveform window, you can clearly see when **C** is "1", the output **D** errorenously still shows "0".
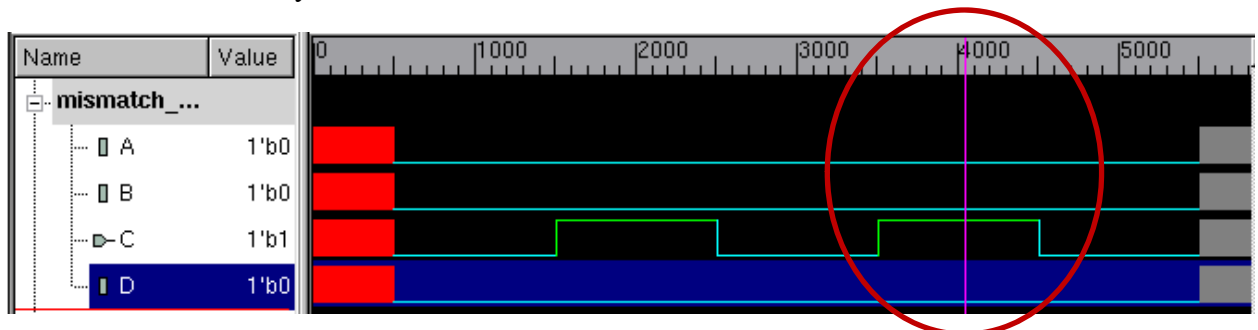
3. Synthesize the design

> **make syn rtl=mismatch**

4. Run simulation at the gate level

> **make gate rtl=mismatch**

This time, in the opened waveform window, you see that when **C** is "1", the output **D** correctly shows "1".
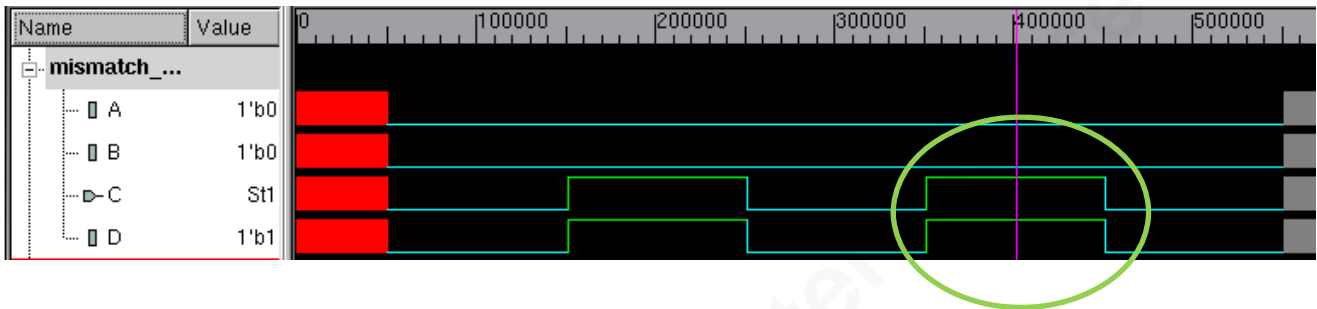
5. You can also see that the synthesis tool completely ignores the event list by looking at the resulting gates.

> **make dv ddc=mismatch_mapped**

6. In the opened Design Vision window, double click on the design, you will see the synthesized gates
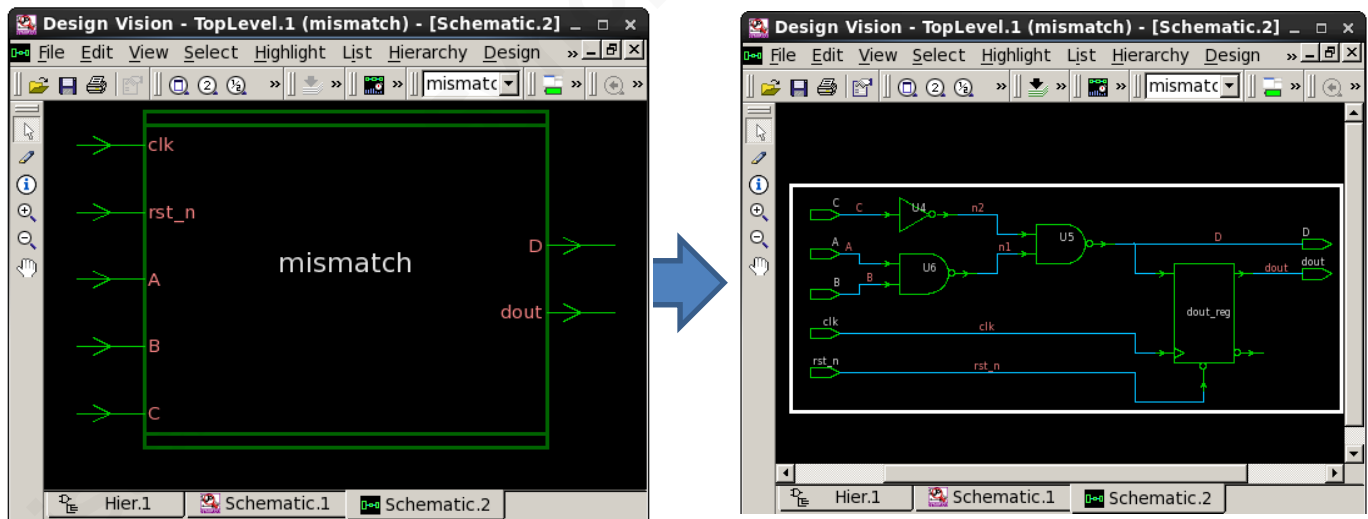
With Verilog, this type of simulation mismatches can be very frustrating to debug. This type of error can be completely eliminated with SystemVerilog

7. Open the existing **mismatch.sv** file with an editor (if not already opened):

    > **gvim rtl/mismatch.sv**

8. Look for the ToDo comment line and enter the following SystemVerilg code:

    ```
    module mismatch (...);
      ...// code not shown
      initial $display("SVERILOG");
      // Lab 1 Task 2 Step 8
      // ...
      // ToDo:
      always_comb begin
        D = (A & B) | C;
      end

    endmodule
    ```

9. Compile and simulate this RTL code

    > **make sim rtl=mismatch lang=sverilog**

    You should see the expected behavior:
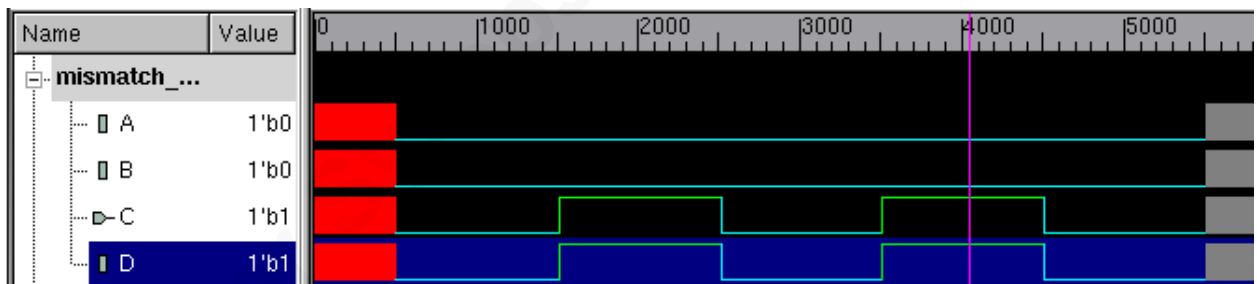
10. Synthesize and run simulation at the gate level

    > **make syn rtl=mismatch lang=sverilog**

    > **make gate rtl=mismatch**

    You should see that the gate level simulation produces the same result when using the SystemVerilog **always_comb** feature.

11. Check the synthesized gates in Design Vision

    > **make dv ddc=mismatch_mapped**

    You should see the same gates as before.

    The difference between the Verilog vs. the SystemVerilog code is the elimination of simulation mismatch.

Unintentional Latch

Task 3.   See the Effects of Incomplete Branch in RTL

Another issue associated with Verilog is the creation of unintentional latches.

In this section, you will run simulation at RTL and gate level to see that if RTL branch code is not complete, both the simulator and synthesizer will treat the code as latch.

1.   Take a look at a RTL file with unintentional latch code:

> **`less rtl/unintentional_latch.sv`**

The branch on **`selA`** is incomplete for both the Verilog and SystemVerilog code:

```
`ifndef sverilog
  always@(selA, A) begin
    if (selA) begin
      B = A;
    end
  end
`else
  always_comb begin
    if (selA) begin
      B = A;
    end
  end
`endif
```

Missing **`else`**

2.   Compile and simulate this RTL code

> **`make sim rtl=unintentional_latch`**

When **`selA`** is 1, **`B`** is the **`A`** value otherwise **`B`** should be 0. You can see that this is not the result. When **`selA`** is "0", the output **`B`** stayed at the captured value (latched) when **`selA`** returned to 0. Also, **`B`** should never be x!
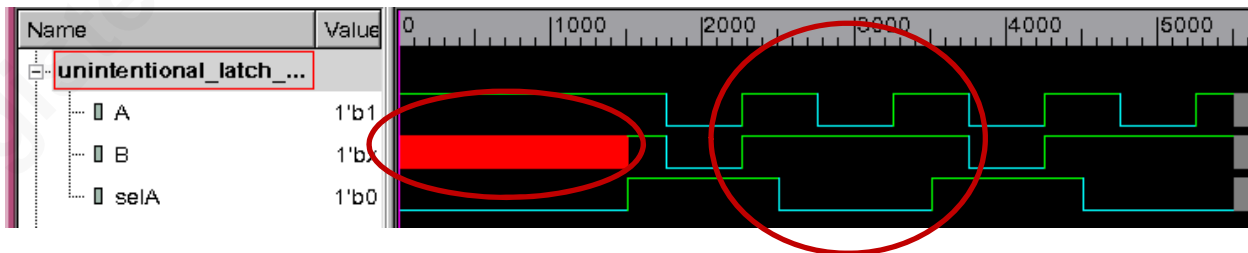
This is the classic unintended latch problem. Simulate with **`lang=sverilog`**, you will still see the same result. The difference between Verilog (**`always`**) and SystemVerilog (**`always_comb`**) is the synthesis report.

3. Synthesize the Verilog RTL code (with **always**)

> **make syn rtl=unintentional_latch**

4. Take a look at the synthesis log file

> **less unintentional_latch_run.log**

Scroll down, you should see that a latch was created and no warning issued:

```
    in routine unintentional_latch line 11 in file
        './rtl/unintentional_latch.sv'.
=======================================================================
|    Register Name    | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
=======================================================================
|       B_reg         | Latch |   1   |  N  |  N |  N |  N |  - |  - |  - |
=======================================================================
...
Presto compilation completed successfully.
```

5. Synthesize the SystemVerilog RTL code (with **always_comb**)

> **make syn rtl=unintentional_latch lang=sverilog**

6. Take a look at the synthesis log file

> **less unintentional_latch_run.log**

You should now see a warning message:

```
    in routine unintentional_latch line 40 in file
        './rtl/unintentional_latch.sv'.
=======================================================================
|    Register Name    | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
=======================================================================
|       B_reg         | Latch |   1   |  N  |  N |  N |  N |  - |  - |  - |
=======================================================================

Inferred memory devices in process
    in routine unintentional_latch line 68 in file
        './rtl/unintentional_latch.sv'.
=======================================================================
| Register Name  |    Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=======================================================================
|   dout_reg     | Flip-flop  |   1   |  N  |  N |  Y |  N |  N |  N |  N |
=======================================================================
Warning:  ./rtl/unintentional_latch.sv:40: Netlist for always_comb block
contains a latch. (ELAB-974)
Presto compilation completed successfully.
```

You will need to modify the RTL code to resolve the problem.

7. Open `unintentional_latch.sv` file

> `gvim rtl/unintential_latch.sv`

8. Look for the ToDo's (two of them) and add the missing branch to the source code.

9. Run simulation at Verilog RTL level

> `make sim rtl=unintentional_latch`

The result should be correct now – including the elimination of x!

10. Synthesize the modified Verilog RTL code and view the simulation results

> `make syn rtl=unintentional_latch`

> `make gate rtl=unintentional_latch`

The result should be correct.

If you set the language to `sverilog`, you will also see that with the else statement added, all are working correctly.

11. If you are interested in seeing the gate logic:

> `make dv ddc=unintentional_latch_mapped`

Continue on to the last part of the lab: state machine.

SystemVerilog Enum in State Machine

Task 4.   Implement enum Data Type for State Machines

One of the most useful data type in SystemVerilog for state machine development is the new enum data type. To illustrate the benefit of the enum data type, let's use a very simple traffic light state machine as an example.
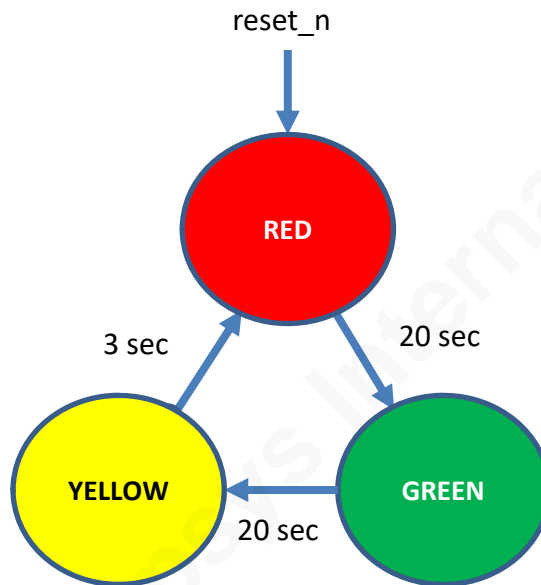
1.   Take a look at the **traffic_light.sv** file to see the state machine

> **less rtl/traffic_light.sv**

The state machine coding style is one-hot with 3 bits of logic.

```
1  module traffic_light(input logic clk, reset_n, output logic green_on, yellow_on, red_on);
2  logic [2:0] state, nxt_state;
3
4  logic [5:0] count_seconds;
5  logic       reset_count;
6
7  always_comb begin
8    nxt_state    = state;
9    reset_count  = 1'b0;
10   red_on       = 1'b0;
11   green_on     = 1'b0;
12   yellow_on    = 1'b0;
13   unique case(1'b1)
14     state[0]: begin
15               red_on = 1'b1;
16               if (count_seconds >= 20) begin
17                 nxt_state    = 3'b010;
18                 reset_count  = 1'b1;
19               end
20             end
21     state[1]: begin
22               green_on = 1'b1;
```

This works, but trying to deciphor the code takes a little getting use to. The effect is more pronounced in the waveform window.

2.   Simulate and open the waveform window

```
> make sim rtl=traffic_light
```

This is what you see:
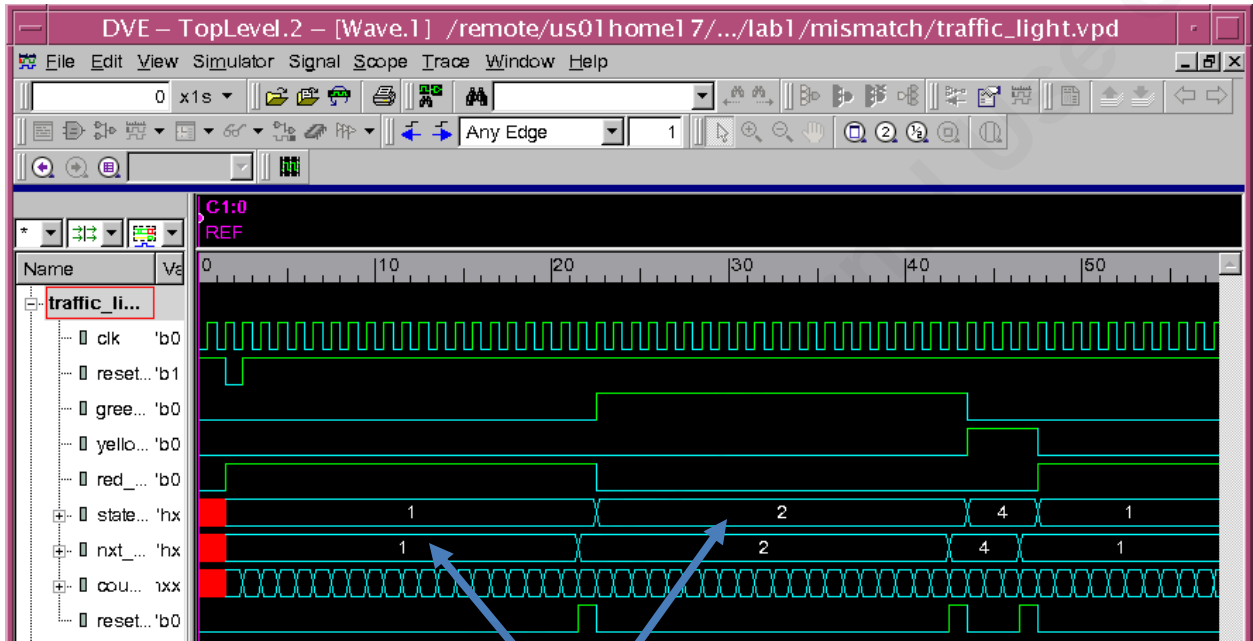
It is very hard to tell what these numbers mean.

Let's convert the RTL state machine to use enum data type.

3.   Open **traffic_light.sv** file

```
> gvim rtl/traffic_light.sv
```

4.   Declare the following enum data type in the beginning of the file:

```
typedef enum logic[2:0] {RED = 3'b001, GREEN = 3'b010,
YELLOW = 3'b100} state_e;
```

5.   Change the **state** and **nxt_state** variables to enum data type

```
// logic [2:0] state, nxt_state;

state_e state, nxt_state;
```

6. Change the following highlighted code to use the **enum** data type values

```
27 always_comb begin
28   nxt_state     = state;
29   reset_count   = 1'b0;
30   red_on        = 1'b0;
31   green_on      = 1'b0;
32   yellow_on     = 1'b0;
33 //   case(1'b1)
34   case(state)
35 //     state[0]: begin
36     RED: begin
37                 red_on = 1'b1;
38                 if (count_seconds >= 20) begin
39 //                    nxt_state     = 3'b010;
40                    nxt_state     = GREEN;
41                    reset_count   = 1'b1;
42                 end
43             end
44 //       state[1]: begin
45     GREEN: begin
46                 green_on = 1'b1;
47                 if (count_seconds >= 20) begin
48 //                    nxt_state     = 3'b100;
49                    nxt_state     = YELLOW;
50                    reset_count   = 1'b1;
51                 end
52             end
53 //       state[2]: begin
54     YELLOW: begin
55                 yellow_on = 1'b1;
56                 if (count_seconds >= 3) begin
57 //                    nxt_state     = 3'b001;
58                    nxt_state     = RED;
59                    reset_count   = 1'b1;
60                 end
61             end
62   endcase
63 end
64
65 always_ff @(posedge clk or negedge reset_n) begin
66   if (!reset_n) begin
67 //     state      <= 3'b001;
68     state      <= RED;
69   end else begin
70     state      <= nxt_state;
71   end
72 end
```

Lab 1

7. Re-run the simulation with the modified RTL code

> `make sim rtl=traffic_light`

This is what you see now:
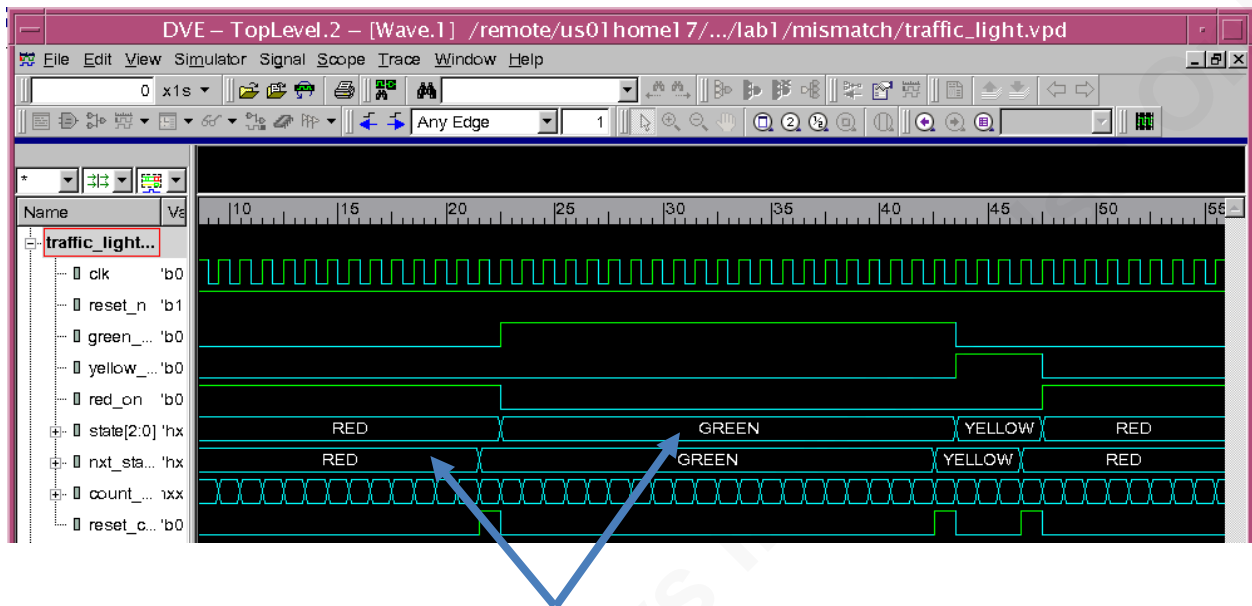
Isn't this a lot more readable? The RTL source code is also a lot easier to read.

This benefit, exist only for the RTL simulation though. After synthesis, at the gate level, they are once again just ones and zeros.

You are done with Lab 1!

Answers / Solutions

`mismatch.sv` Solution:

```
module mismatch (clk, rst_n, A, B, C, D, dout);

  input  clk, rst_n;
  input  A, B, C;
  output D;
  output dout;
  logic  D, dout;

`ifndef sverilog

  // This is the Verilog behavioral code
  always@(A, B, C) begin
    D = (A & B) | C;
  end

`else

  always_comb begin
    D = (A & B) | C;
  end

`endif

  always@(posedge clk, negedge rst_n) begin
    if (!rst_n) begin
      dout <= 0;
    end else begin
      dout <= D;
    end
  end

endmodule
```

<u>**unintentional latch.sv** Solution:</u>

```systemverilog
module unintentional_latch (clk, rst_n, A, B, selA, dout);

  input  clk, rst_n;
  input  A, selA;
  output B, dout;
  logic  B, dout;

`ifndef sverilog

  // This is the Verilog behavioral code
  always@(selA, A) begin
    if (selA) begin
      B = A;
    end

    else begin
      B = 0;
    end
  end

`else

  // When the branch code is incomplete, SystemVerilog also treats it
as a latch code
  always_comb begin
    if (selA) begin
      B = A;
    end

    else begin
      B = 0;
    end
  end

`endif

  always@(posedge clk, negedge rst_n) begin
    if (!rst_n) begin
      dout <= 0;
    end else begin
      dout <= B;
    end
  end

endmodule
```

traffic light.sv Solution:

```systemverilog
typedef enum logic[2:0] {RED = 3'b001, GREEN = 3'b010, YELLOW = 3'b100} state_e;
module traffic_light(input logic clk, reset_n, output logic green_on, yellow_on,
red_on);
state_e state, nxt_state;
logic [5:0] count_seconds;
logic       reset_count;
always_comb begin
  nxt_state    = state;
  reset_count  = 1'b0; red_on = 1'b0; green_on = 1'b0; yellow_on = 1'b0;
  case(state)
    RED:     begin
              red_on = 1'b1;
              if (count_seconds >= 20) begin
                nxt_state    = GREEN;
                reset_count  = 1'b1;
              end
             end
    GREEN:   begin
              green_on = 1'b1;
              if (count_seconds >= 20) begin
                nxt_state    = YELLOW;
                reset_count  = 1'b1;
              end
             end
    YELLOW: begin
              yellow_on = 1'b1;
              if (count_seconds >= 3) begin
                nxt_state    = RED;
                reset_count  = 1'b1;
              end
             end
  endcase
end
always_ff @(posedge clk or negedge reset_n) begin
  if (!reset_n) begin
    state      <= RED;
  end else begin
    state      <= nxt_state;
  end
end
always_ff @(posedge clk or negedge reset_n) begin
  if (!reset_n) begin
    count_seconds   <= '0;
  end else begin
    if (reset_count) begin
      count_seconds <= '0;
    end else begin
      count_seconds <= count_seconds + 1;
    end
  end
end
endmodule
```

User Logic Intent   **Lab 1-15**
SystemVerilog RTL Design Workshop

This page was intentionally left blank.

2 SystemVerilog Interface

Learning Objectives

After completing this lab, you should be able to:

- Implement modport in SystemVerilog interface

- Synthesize RTL code with parameterized interface

- Create synthesis script to generate gate level netlist for simulation

- Create synthesis script to integrate lower level module with parameterized interface

Lab Duration: 45 minutes

Lab Overview

Of the new features in SystemVerilog, the one that makes the biggest impact to the RTL coding style is the new interface mechanism. But, because backend tools do not support SystemVerilog interface, synthesis tools deconstruct the interface into individual port listings. This results in a problem that one must resolve when integrating gate level netlist into higher level blocks (bottom up approach to synthesis) and developing testbench for gate level simulation.

This lab will take you through how to manage the interfaces (especially with parameters) for gate level block integration and simulation.

The general work flow for each section of this lab is illustrated as follows.

```
┌─────────────────────────────┐
│          Modify             │
│    Verilog RTL to use       │
│  SystemVerilog Interface    │
└─────────────────────────────┘
              │
              ▼
┌─────────────────────────────┐
│       Synthesize and        │
│     verify functionality    │
└─────────────────────────────┘
```
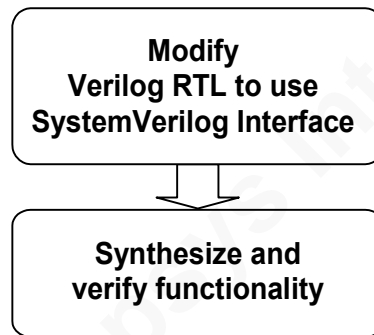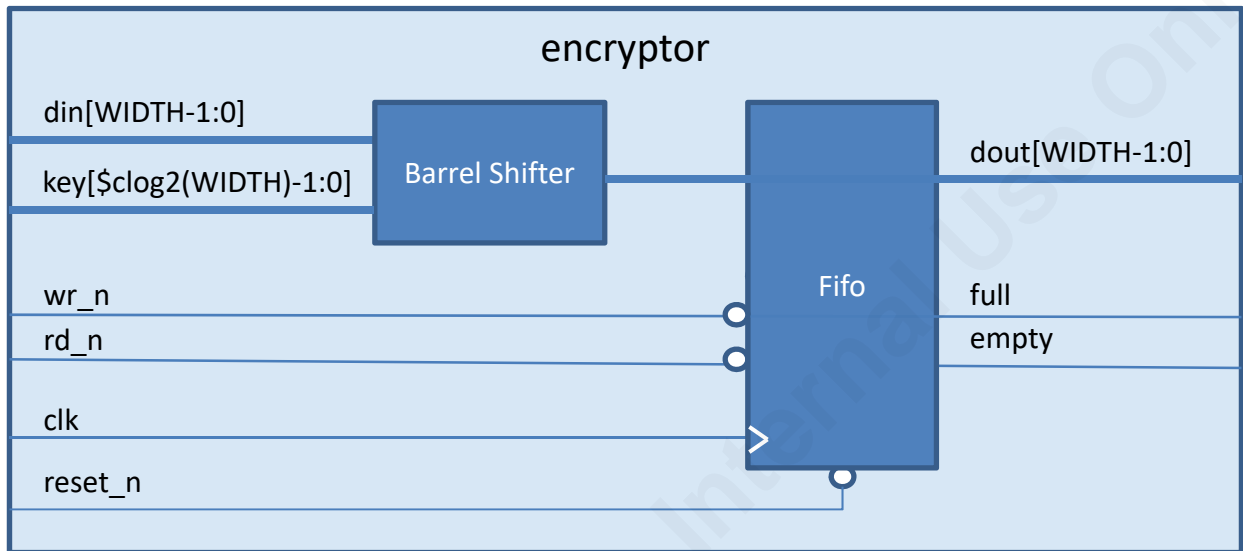
Figure 1.   Lab 2 Flow Diagram

Implement Interface and Verify RTL at Block level

For this lab, the design is an encryptor with a **fifo** embedded.

You will take a bottom up approach to the integration.

- First, verify the **fifo** module implemented with Verilog style port list

- Then, replace the port list of signals of the **fifo** module with a SystemVerilog interface

- Develop DC script to synthesize the **fifo** module and generate files needed for integration and simulation

- Verify the **fifo** gate level netlist using the generated definition

- Develop DC script to synthesize the top-level **encryptor** using the **fifo** block gate-level ddc generated with the previous synthesis run

- Verify that the synthesized top-level **encryptor** is functionally correct

Task 1.   Go Into Lab Directory

1.   Change directory to **lab2/interface** directory:

```
> cd ../lab2
```

Task 2.   Examine and Verify the `fifo` Module

1.   Take a look at the **fifo** module

> **less rtl/fifo.sv**

You should see that the module is parameterized and the port list of the module is coded without SystemVerilog interface.

2.   Take a look at the **fifo** testbench

> **less test/fifo_test_top.sv**

```
`define StringConvert(s) `"s`"

module fifo_test_top;
  parameter WIDTH    = 8;
  parameter BUF_SIZE = 16;

  string           design = `StringConvert(`DESIGN);
  bit              clk    = 0;
  logic            reset_n;

  fifo_io#(WIDTH) fifo_if(clk);

`ifdef GATE

  ...

`else
  ..
  fifo#(WIDTH, BUF_SIZE) dut(.*,
                           .rd_n  (fifo_if.rd_n),
                           .wr_n  (fifo_if.wr_n),
                           .full  (fifo_if.full),
                           .empty (fifo_if.empty),
                           .din   (fifo_if.din),
                           .dout  (fifo_if.dout));
`endif
```

Notice that even though the RTL code did not implement an interface for the **fifo** module, the testbench still makes use of the SystemVerilog interface.

The reality is that, in the verification world, testbenches make use of the SystemVerilog interface to simplify the development of the testbench device drivers even if the RTL code does not. This is one of the reasons why design engineers must understand the SystemVerilog interface mechanism.

3.   Verify that this code works correctly before playing with the SystemVerilog interface

> **make sim rtl=fifo**

You should see all expected values matched.

Task 3. Add a `modport` Declaration in the `interface`

1. Open the existing **fifo** interface file with an editor

> **gvim rtl/fifo_io.sv**

The clocking blocks in the interface is for simulation only. They are typically specified by the verification engineer.

Design engineer is typically responsible for the **modport** declaration.

2. Search for the **ToDo** comment (at end of the file) and implement a **modport** for the **fifo** module (for reference see the directions specified in **fifo.sv**)

Task 4. Modify `fifo` to Use SystemVerilog Interface

1. Open the **fifo** RTL file with an editor

> **gvim rtl/fifo.sv**

2. Modify the port list to use the **fifo_io** interface and the **modport**

Change the existing code from:

```
module fifo #(WIDTH=8, BUF_SIZE=16) (input logic clk,
reset_n, rd_n, wr_n, logic[WIDTH-1:0] din, output logic
empty, full, logic[WIDTH-1:0] dout);
```

To: (DO NOT change **clk** and **reset_n**)

```
module fifo #(WIDTH=8, BUF_SIZE=16) (input logic clk,
reset_n, fifo_io.fifo fifo_if);
```

The final step in the RTL conversion is a bit of a headache. You will need to add **fifo_if** in front of all modified port signals with a dot notation.

3. Locate all the modified port signals (rd_n, wr_n, din, empty, full and dout) and change them to – **fifo_if.rd_n**, **fifo_if.wr_n**, **fifo_if.din**, **fifo_if.empty**, **fifo_if.full** and **fifo_if.dout**.

(In gvim, it would be something like the following)

> **:50,$s/rd_n/fifo_if.rd_n/g**

> **:50,$s/wr_n/fifo_if.wr_n/g**

> **:50,$s/din/fifo_if.din/g**

> **:50,$s/dout/fifo_if.dout/g**

> **:50,$s/empty/fifo_if.empty/g**

> **:50,$s/full/fifo_if.full/g**

Task 5.   Modify testbench to Use SystemVerilog Interface

1.   Open the testbench file with an editor

> **gvim test/fifo_test_top.sv**

2.   Modify the port list to use the **fifo_io** interface and the **modport**

Change the existing code from:

```
`else
  fifo#(WIDTH, BUF_SIZE) dut(.*,
                            .rd_n (fifo_if.rd_n),
                            .wr_n (fifo_if.wr_n),
                            .full (fifo_if.full),
                            .empty(fifo_if.empty),
                            .din  (fifo_if.din),
                            .dout (fifo_if.dout));
`endif
```

To:

```
`else
  fifo#(WIDTH, BUF_SIZE) dut(.*);
`endif
```

The connectivity of modules using the SystemVerilog interface simplifies tremendously.
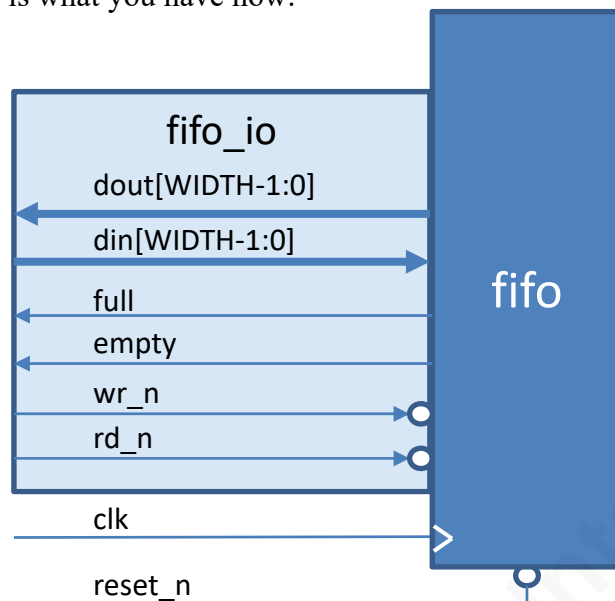
3.   Make sure that nothing is broken through this conversion process

> **make sim rtl=fifo**

You should see all expected values matched. You are now ready to synthesis the **fifo** into a gate level netlist.

Task 6.   Synthesize `fifo` and Examine Gate Level Netlist

This is what you have now:

1.   Take a look at the synthesis script to make sure you see what's being executed

> **`less script/fifo_run.tcl`**

2.   Synthesize this into gate level

> **`make syn rtl=fifo`**

3.   Take a look at the elaborated netlist

> **`less unmapped/fifo_unmapped.v`**

Notice after elaboration, the module name and the port list changed.

Was:

```
module fifo #(WIDTH=8, BUF_SIZE=16) (input logic clk,
reset_n, fifo_io.fifo fifo_if);
```

At synthesis, the parameters become part of the module name and the interface is decomposed into individual signal sets:

```
module fifo_WIDTH8_BUF_SIZE16 ( clk, reset_n,
\fifo_if.rd_n ,  \fifo_if.wr_n , \fifo_if.din ,
\fifo_if.empty , \fifo_if.full , \fifo_if.dout  );
```

Furthermore, the whole point of applying parameters in modules and interfaces is for the flexibility of module and interface reuse. In this case, the interface is also parameterized. But notice that the parameter of the interface does not show up in the code at all.

This can be a huge problem. To see the issue, execute the following steps.

4.   Synthesize `fifo` with a parameter that's different from the default value

> `make syn rtl=fifo WIDTH=16`

5.   Check the gate level netlist result

> `less unmapped/fifo_unmapped.v`

You should see that the name of the module reflects the new parameter. But, bit width of the content of the interface is wrong!

```
module fifo_WIDTH16_BUF_SIZE16 ( clk, reset_n, \fifo_if.rd_n , \fifo_if.wr_n ,
        \fifo_if.din , \fifo_if.empty , \fifo_if.full , \fifo_if.dout  );
  input [7:0] \fifo_if.din ;
  output [7:0] \fifo_if.dout ;
  input clk, reset_n, \fifo_if.rd_n , \fifo_if.wr_n ;
  output \fifo_if.empty , \fifo_if.full ;
```

You may attemp to get around the issue by adding the parameter to the interface reference:

```
module fifo #(WIDTH=8, BUF_SIZE=16) (input logic clk, reset_n,
                          fifo_io#(WIDTH).fifo fifo_if);
```

Unfortunately, this is illegal in SystemVerilog and will not compile.

If you read the SystemVerilog LRM, it tells you to use a generic interface:

```
module fifo #(WIDTH=8, BUF_SIZE=16) (input logic clk, reset_n,
                          interface fifo_if);
```

This gets even worse!

The synthesis tool now has no idea what kind of interface `fifo_if` is supposed to be when the module is synthesized as the current design in a bottom up synthesis approach.

How does one get around this problem?

Answer: create a wrapper module.

Task 7. Create a Wrapper Module

1. Create a **fifo** wrapper module file

> **gvim rtl/wrapper_fifo.sv**

2. Enter the following code:

```
module wrapper_fifo#(WIDTH=8, BUF_SIZE=16) (input clk, reset_n);
  fifo_io#(WIDTH)            fifo_if();
  fifo   #(WIDTH, BUF_SIZE)   fifo_inst (.*);
endmodule
```

By instantiating the interface with parameter, DC will recognize the user's intent. You do need to adjust the synthesis script to use this wrapper module.

Task 8. Adjust Synthesis Script to Generate Correctly Mapped Gate Level Logic

1. Open the **fifo** synthesis script

> **gvim script/fifo_run.tcl**

2. Make the following adjustments

```
# fifo_run.tcl
source ../../../script/common_setup.tcl
source ../../../script/dc_setup.tcl
set param_list "WIDTH=$_width, BUF_SIZE=$_size"

analyze -format sverilog { fifo_io.sv fifo.sv wrapper_fifo.sv }
elaborate wrapper_fifo -param $param_list

current_design [get_designs fifo*]

link
write_                                         ed/fifo_unmapped.v
write_                                         ifo_unmapped.ddc
```

> Add wrapper file to be analyzed

> Elaborate the wrapper NOT the fifo

> Add this line to synthesize the fifo, NOT the wrapper

The purpose of the wrapper is to get the synthesizer to recognize the interface parameters at the elaboration phase. Once elaborated, you will only deal with the **fifo** module for synthesis. The wildcard (*****) is needed because the name of the module is no longer **fifo**, but the expanded name.

3. Use the new script to synthesize the **fifo**

> **make syn rtl=fifo WIDTH=16**

4. Check the new module name

> **less unmapped/fifo_unmapped.v**

You should see that the name of the module now reflects not only the paramerter of the module, but the parameterized interface as well.

The bit width of the content of the interface are also correct.

```
module fifo_WIDTH16_BUF_SIZE16_I_fifo_if_fifo_io__16 ( clk, reset_n,
        \fifo_if.clk , \fifo_if.rd_n , \fifo_if.wr_n , \fifo_if.empty ,
        \fifo_if.full , \fifo_if.din , \fifo_if.dout );
    input [15:0] \fifo_if.din ;
    output [15:0] \fifo_if.dout ;
    input clk, reset_n, \fifo_if.clk , \fifo_if.rd_n , \fifo_if.wr_n ;
    output \fifo_if.empty , \fifo_if.full ;
```

You now have a correctly synthesized module.

One more issue. You may need to verify the operation of this gate level netlist. The last thing you want to do is to hand code all these changes into the testbench.

Let's adjust the synthesis script one more time to generate an instance of the gate level module that you can copy and paste into the testbench.

Task 9.   Adjust Synthesis Script to Generate Correctly Mapped Gate Level Instance for Testbench

1. Open the **fifo** synthesis script

> **gvim script/fifo_run.tcl**

2. Make the following adjustments (end of file)

```
# The following is for simulation
# Procedure for retrieving design from memory
proc get_design_from_inst { inst } {
  return [get_attribute [get_cells $inst] ref_name]
}
current_design [get_designs wrapper_fifo*]
set dut [get_design_from_inst fifo_inst]
write_file -format svsim -output wrapper/fifo_wrapper.sv $dut

exit
```

> Need the wrapper to generate the correct mapping

> Retrieve the instantiated module from within the wrapper and write out the instance in SystemVerilog

SystemVerilog Interface
SystemVerilog RTL Design Workshop

3. Rerun the synthesis

> **make syn rtl=fifo WIDTH=16**

4. Open the generated file

> **gvim wrapper/fifo_wrapper.sv**

You should see the instantiated **fifo** module with the correct module name and mapping of module port signals.

```
fifo_WIDTH16_BUF_SIZE16_I_fifo_if_fifo_io__16 fifo_WIDTH16_BUF_SIZE16_I_fifo_if_fifo_io__16(
        {>>{ clk }}, {>>{ reset_n }}, , {>>{ fifo_if.rd_n }},
        {>>{ fifo_if.wr_n }}, {>>{ fifo_if.empty }}, {>>{ fifo_if.full }},
        {>>{ fifo_if.din }}, {>>{ fifo_if.dout }} );
```

5. Also open the testbench file

> **gvim test/fifo_test_top.sv**

6. Copy the instantaiated code into the testbench

```
module fifo_test_top;
  parameter WIDTH    = 8;
  parameter BUF_SIZE = 16;

  string            design = `StringConvert(`DESIGN);
  bit               clk     = 0;
  logic             reset_n;

  fifo_io#(WIDTH) fifo_if(clk);

`ifdef GATE
  // Lab 2 Task 9 Step 6
  //
  // Add the remapped gate-level module here:
  //
  // ToDo:
  fifo_WIDTH16_BUF_SIZE16_I_fifo_if_fifo_io__16 fifo_WIDTH16_BUF_SIZE16_I_fifo_if_fifo_io__16(
    {>>{ clk }}, {>>{ reset_n }}, , {>>{ fifo_if.rd_n }},
    {>>{ fifo_if.wr_n }}, {>>{ fifo_if.empty }}, {>>{ fifo_if.full }},
    {>>{ fifo_if.din }}, {>>{ fifo_if.dout }} );

`else
```

A couple of thing to notice: the gate level instance name no longer matches the RTL instance name. If naming consistency is important, you will need to manually change the instance to match the RTL instance name (**dut**). And, this testbench is no useable for other gate-level netlists with different parameters.

7. Execute the gate level verification

> **make gate rtl=fifo WIDTH=16**

The gate simulation should pass. The major caution here is that with gate level simulation, each testbench can only handle one variation of the parameter. You may want to develop a script to generate the testbench as you need it.

SystemVerilog Interface   **Lab 2-11**
SystemVerilog RTL Design Workshop

Integrate `fifo` into the `encryptor` module

You have successfully synthesize the **fifo** module.

Because of the parameters and interfaces of the module, integrating this into a higher level block (bottom up synthesis approach) requires a little tweak to the synthesis script.

For this part of the lab, the RTL design is an **encryptor** embedding the **fifo**:
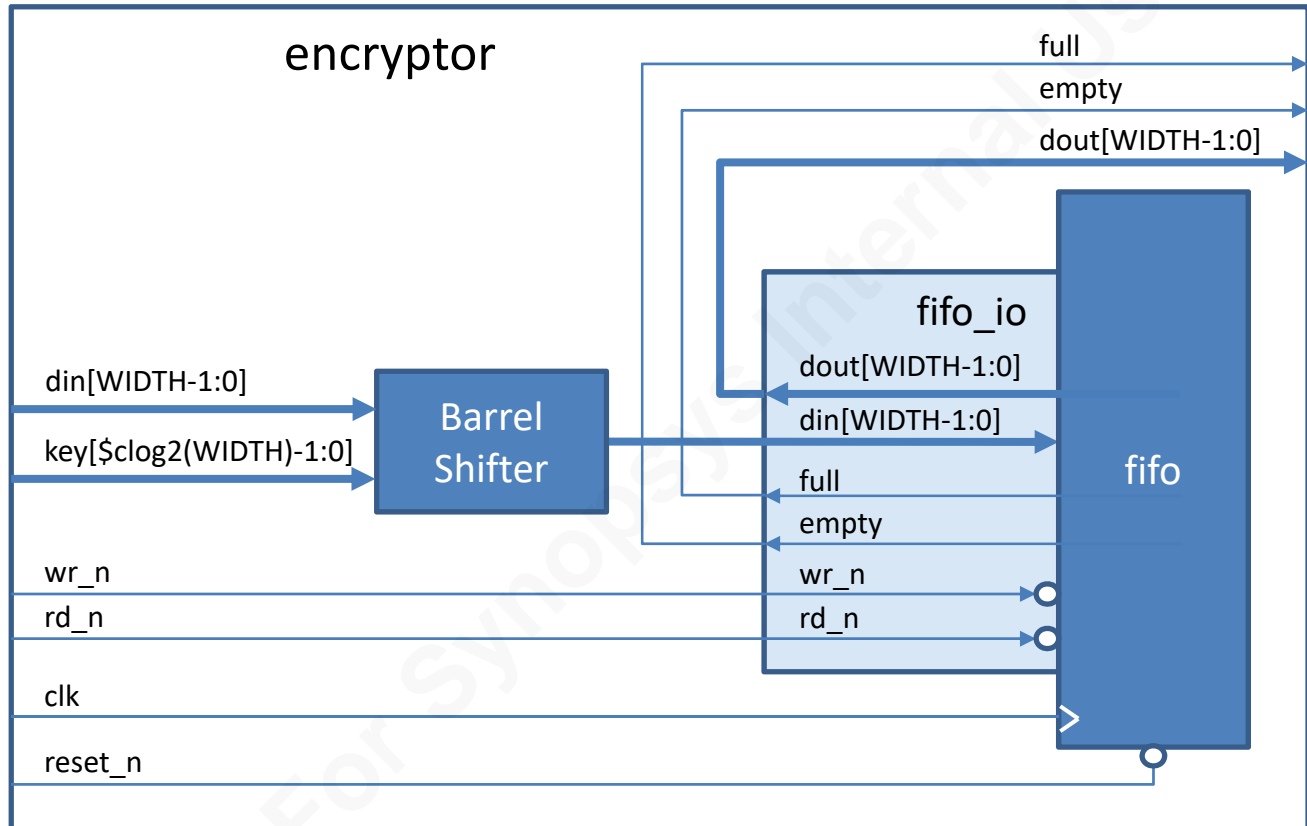
Task 10.   Examine the `encryptor` Module

This **encryptor** module is very simple. The simplicity let us focus more on the task at hand: integration of a module with parameterized interface.

1.   Take a look at the **encryptor** module code:

```
> less rtl/encryptor.sv
```

You should see that the module has parameters. But, to keep focus on just the integration of lower level block with parameterized interfaces, the **encryptor** module port list is kept at the Verilog port list style. Inside the module there is one instance of **fifo_io** interface and one instance of **fifo** module.

2. Simulate the module to make sure the **encryptor** is working correctly:

> **make sim rtl=encryptor WIDTH=16**

You should see that the RTL passed simulation.

3. Take a look at the **encryptor** synthesis script:

> **less script/encryptor_run.tcl**

The content is similar to the **fifo_run.tcl** script. The key difference between the two scripts are two lines:

```
1 # encryptor_run.tcl
2
3 source ../../script/common_setup.tcl
4 source ../../script/dc_setup.tcl
5
6 set param_list "WIDTH=$_width, BUF_SIZE=$_size"
7
8 analyze -format sverilog { fifo_io.sv encryptor.sv wrapper_encryptor.sv }
9 elaborate wrapper_encryptor -param $param_list
10 link
11
12 remove_design [get_designs fifo*]
13 read_ddc fifo_mapped.ddc
```

If you already have a synthesized module that you want to use. You need to remove the newly read in code from memory with the **remove_design** command. Then, you need to read the saved synthesized module into the memory with the **read_ddc** command.

4. Synthesize the **encryptor** module

> **make syn rtl=encryptor WIDTH=16**

To make sure everything at gate level works properly, modify the testbench to enable gate level verification

5. Open the testbench file

> **gvim test/encryptor_test_top.sv**

6. Copy the **encryptor** instance from **wrapper/encryptor_wrapper.sv** into the testbench

7. Verify the the **encryptor** module

> **make gate rtl=encryptor WIDTH=16**

Now you know how to deal with SystemVerilog interface and parameters for a bottom up synthesis approach.

You are done with Lab 2!

Answers / Solutions

fifo io.sv Solution:

```
`ifndef SYNTHESIS
interface fifo_io #(WIDTH = 8) (input clk);
`else
interface fifo_io #(WIDTH = 8) (); // RTL does not need clk
`endif
  logic            rd_n,
                   wr_n,
                   empty,
                   full;

  logic [WIDTH-1:0] din,
                    dout;

`ifndef SYNTHESIS
  clocking drvWrClk @(posedge clk);
    default input #1ns output #1ns;
    output wr_n;
    output din;
    input  full;
  endclocking

  clocking drvRdClk @(posedge clk);
    default input #1ns output #1ns;
    output rd_n;
    input  empty;
    input  dout;
  endclocking

  clocking monWrClk @(posedge clk);
    default input #1ns output #1ns;
    input  wr_n;
    input  din;
  endclocking

  clocking monRdClk @(posedge clk);
    default input #1ns output #1ns;
    input  rd_n;
    input  dout;
  endclocking
`endif

  modport fifo(input rd_n, wr_n, din, output empty, full, dout);

endinterface
```

<u>`fifo.sv` Solution:</u>

```systemverilog
 module fifo #(WIDTH=8, BUF_SIZE=16) (input logic clk, reset_n,
fifo_io.fifo fifo_if);

logic [WIDTH-1:0]           reg_buffer [BUF_SIZE];
logic [$clog2(BUF_SIZE):0]   count;
logic [$clog2(BUF_SIZE)-1:0] wr_address,
                             rd_address;

assign fifo_if.dout    = reg_buffer[rd_address];
assign fifo_if.empty   = ((count == 0) && fifo_if.wr_n)          ||
((count == 1)         && fifo_if.wr_n && !fifo_if.rd_n);
assign fifo_if.full    = ((count == BUF_SIZE) && fifo_if.rd_n) ||
((count == BUF_SIZE-1) && !fifo_if.wr_n && fifo_if.rd_n);

always_ff @(posedge clk or negedge reset_n) begin
  if (!reset_n) begin
    wr_address <= 0;
    rd_address <= 0;
    count <= 0;
  end else begin
    case ({fifo_if.wr_n, fifo_if.rd_n})
      2'b00: begin
               reg_buffer[wr_address] <= fifo_if.din;
               wr_address <= wr_address + 1;
               rd_address <= rd_address + 1;
             end
      2'b01: begin
               reg_buffer[wr_address] <= fifo_if.din;
               wr_address <= wr_address + 1;
               count <= count + 1;
             end
      2'b10: begin
               rd_address <= rd_address + 1;
               count <= count - 1;
             end
      2'b11: ;
    endcase
  end
end

endmodule
```

fifo run.tcl Solution:

```
# fifo_run.tcl

source ../../../script/common_setup.tcl
source ../../../script/dc_setup.tcl

set param_list "WIDTH=$_width, BUF_SIZE=$_size"

#analyze -format sverilog { fifo_io.sv fifo.sv }
#elaborate fifo -param $param_list
analyze -format sverilog { fifo_io.sv fifo.sv wrapper_fifo.sv }
elaborate wrapper_fifo -param $param_list
current_design [get_designs fifo*]

link

write_file -format verilog -output unmapped/fifo_unmapped.v
write_file -format ddc -output unmapped/fifo_unmapped.ddc

check_design -html check_design.html

source ../../../script/constraint.tcl

compile_ultra

write_file -format verilog -output mapped/fifo_mapped.v
write_file -format ddc -output mapped/fifo_mapped.ddc

# The following is for simulation

# Procedure for retrieving design from memory
proc get_design_from_inst { inst } {
  return [get_attribute [get_cells $inst] ref_name]
}

current_design [get_designs wrapper_fifo*]
set dut [get_design_from_inst fifo_inst]
write_file -format svsim -output wrapper/fifo_wrapper.sv $dut

exit
```

encryptor run.tcl Solution:

```
# encryptor_run.tcl

source ../../../script/common_setup.tcl
source ../../../script/dc_setup.tcl

set param_list "WIDTH=$_width, BUF_SIZE=$_size"

analyze -format sverilog { fifo_io.sv encryptor.sv wrapper_encryptor.sv }
elaborate wrapper_encryptor -param $param_list
remove_design [get_designs fifo*]

link

read_ddc fifo_mapped.ddc

current_design [get_designs encryptor*]

write_file -format verilog -output unmapped/encryptor_unmapped.v
write_file -format ddc -output unmapped/encryptor_unmapped.ddc

check_design -html check_design.html

source ../../../script/constraint.tcl

compile_ultra

write_file -format verilog -output mapped/encryptor_mapped.v
write_file -format ddc -output mapped/encryptor_mapped.ddc

# The following is for simulation

# Procedure for retrieving design from memory
proc get_design_from_inst { inst } {
  return [get_attribute [get_cells $inst] ref_name]
}

current_design [get_designs wrapper_encryptor*]

set dut [get_design_from_inst encryptor_inst]
write_file -format svsim -output wrapper/encryptor_wrapper.sv $dut

exit
```

This page was intentionally left blank.