# Lab 0 — SystemVerilog

*Lecturer: PhD. Linh Tran*
*TA: Hai Cao*
*Department of Electronics*
*HCMC University of Technology, VNU-HCM*

**Abstract**

*This document helps you familiarize yourself with SystemVerilog HDL (Hardware Description Language), which you will use to design digital logic blocks, and then you will learn the basic of verifying your RTL design.*

*In case you meet an error or have any improvement in this document, please email the TA: cxhai.sdh221@hcmut.edu.vn with the subject "**[COMPARCH203: FEEDBACK]**"*

## 1 Objectives

- Familiarize yourself with basic design flow.
- Review some basics of SystemVerilog HDL with existing codes.
- Implement the design on KIT.

### 1.1 Prerequisites

You should prepare two essential softwares:

- Ubuntu
- Quartus

Additionally, you need to learn some common commands to work in UNIX/Linux environment. You may find this cheatsheet useful.

## 2 Design Flow

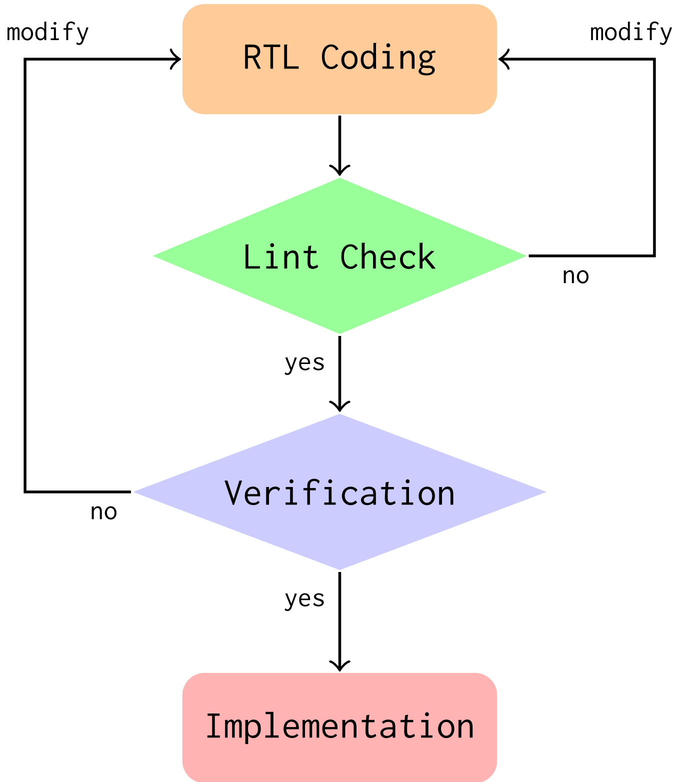


Figure 1: RTL design flow

Figure 1 shows the basic design flow of design a digital logic block. In this diagram, designers have to follow four steps:

1. **RTL Coding** is consisted of designing algorithms, FSMs, etc. to handle the requirements of the problem or expected designs. In this document, you will use SystemVerilog HDL, a versatile HDL for both design and verification.

2. **Lint Check** is a process of static code analysis to check the quality of the code. Lint Check will check your code with a list of rules, to find any violations — syntax errors or potential code lines or blocks that may cause errors or bugs when running. This step is needed to perform right after you finish RTL Coding. If there are any flags or errors, you need to read them carefully to know exactly the rules you violated to fix your code.

3. **Verification** is a process of running your code to check correctness your design. Based on the requirements, each input will have only one expected output, so you have to interpret the log or the waveform generated from this step to fix your design. After fixing your design, don't forget linting again.

4. **Implementation** is to implement the design into real hardwares. Your simulation and the reality are sometimes not the same.

# 3 Combinational Logic Modeling

## 3.1 Problem

*Design the combinational logic in Figure 2.*

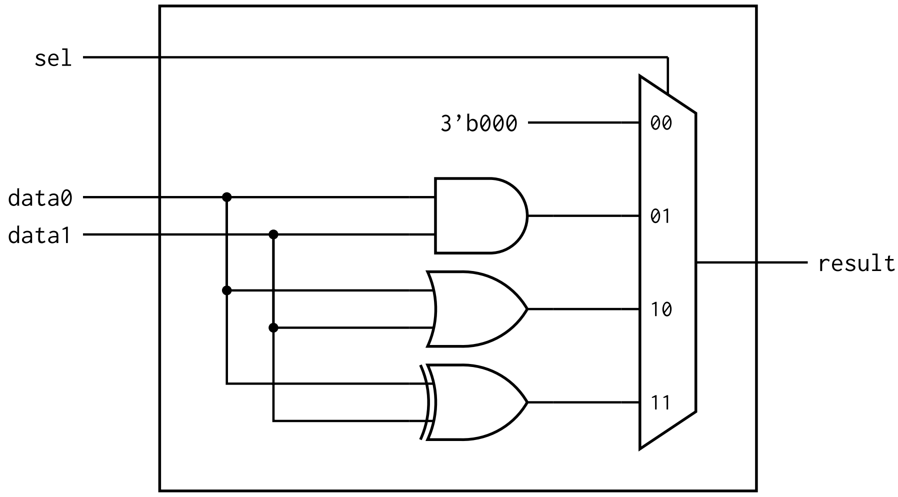


Figure 2: Sample combinational logic

**!** → The data width of `data0`, `data1`, and `result` is **3**, and that of `sel` is **2**.

## 3.2 Design

### 3.2.1 Creating Project

A project should be organized systematically to manage and maintain easily later. First, in the terminal, type the below command to download the example projects of **lab 0** from GitHub to your local computer.

```
git clone https://github.com/joachimcao/lab0
```

Inside the new-downloaded directory, `lab0`, there are two directories, `ex01` and `ex02`. The former is of this section, go to that directory by typing `cd lab0/ex01`. Type `ls` to see all files and directories in the current directory; an alternative is `tree` which displays the tree directory.

```
~/gi/c/lab0/ex01 $ ls
drwxr-xr-x@ - ioachimus 21 Feb 15:53 quartus
drwxr-xr-x@ - ioachimus 21 Feb 15:53 src
drwxr-xr-x@ - ioachimus 21 Feb 15:55 tb
```

```
~/gi/c/lab0/ex01 $ tree
.
|-- quartus
|   |-- de10_pin.qsf
|   |-- de2_pin.qsf
|   `-- wrapper.sv
|-- src
|   `-- design_1.sv
`-- tb
    |-- driver.cpp
    |-- filelist
    |-- makefile
    |-- tb_top.cpp
    `-- top.sv

4 directories, 9 files
~/gi/c/lab0/ex01 $
```

In this sample project, you may notice:

**src** contains source files — SystemVerilog files.

**tb** contains testbench files

- `makefile` is a bash script to run several command lines conveniently.
- `filelist` contains all design files' names.
- `top.sv` is to instantiate the design to be tested.
- `driver.cpp` is a C++ file to drive inputs.
- `tb_top.cpp` is a C++ testbench file.

**quartus** contains DE2/DE10 related files.

- `de2_pin.qsf` is a pin assignments file for DE2 with its pins named according to this file.
- `de10_pin.qsf` is a pin assignments file for DE10 with its pins named according to this file.
- `wrapper.sv` is to connect the design's pins to those of DE2/DE10.

! → All design files have to be placed in `src`.

! → In `tb`, only `filelist`, `top.sv`, and `driver.cpp` will be configured. In `quartus`, only `wrapper.sv` will be edited.

**Question**

Why do you need `wrapper.sv`?

### 3.2.2 RTL Coding

As all design files are placed into `src`, type `gedit src/design_1.sv` to open that file while you're in directory `ex01`. If you want to use `vim`, you could check this website for some basic commands in vim.

```
module design_1 (
  // input
  input logic [2:0] data0_i,
  input logic [2:0] data1_i,
  input logic [1:0] sel_i,

  // output
  output logic [2:0] result_o
);

  // local declaration
  logic [2:0] and_tmp; // temporary for and result
  logic [2:0]  or_tmp; // temporary for  or result
  logic [2:0] xor_tmp; // temporary for xor result

  assign and_tnp = data0_i & data1_i;
  assign  or_tmp = data0_i | data1_i;
  assign xor_tmp = data0_i ~^ data1_i;

  always_comb begin : proc_mux
    case (sel_i)
      2'b00: result_o =      '0;
      2'b01: result_o = and_tmp;
      2'b10: result_o =  or_tmp;
      2'b10: result_o = xor_tmp;
    endcase
  end

endmodule : design_1
```

### 3.2.3 Lint Check

To perform lint check, go to directory `tb` by using `cd tb`. In this directory, all design files are listed in `filelist`. This example only has `design_1.sv` in `src` directory, the content of `filelist` is:

```
../src/design_1.sv
```

Then, in `tb` directory, run `make lint`, powered by Verilator here.

```
~/gi/c/l/ex01/tb $ make lint
------------------> LINT CHECK <------------------
%Warning-IMPLICIT: ../src/design_1.sv:16:10: Signal definition not found, creating implicitly: 'and_tnp'
                                            : ... Suggested alternative: 'and_tmp'
   16 |   assign and_tnp = data0_i & data1_i;
      |          ^~~~~~~
                    ... For warning description see https://verilator.org/warn/IMPLICIT?v=5.003
                    ... Use "/* verilator lint_off IMPLICIT */" and lint_on around source to disable this message.
%Warning-WIDTH: ../src/design_1.sv:16:18: Operator ASSIGNW expects 1 bits on the Assign RHS, but Assign RHS's AND generates 3 bits.
                                         : ... In instance design_1
   16 |   assign and_tnp = data0_i & data1_i;
      |                  ^
%Warning-UNDRIVEN: ../src/design_1.sv:12:15: Signal is not driven: 'and_tmp'
                                            : ... In instance design_1
   12 |   logic [2:0] and_tmp;
      |               ^~~~~~~
%Warning-UNUSEDSIGNAL: ../src/design_1.sv:16:10: Signal is not used: 'and_tnp'
                                                : ... In instance design_1
   16 |   assign and_tnp = data0_i & data1_i;
      |          ^~~~~~~
%Warning-CASEOVERLAP: ../src/design_1.sv:25:7: Case values overlap (example pattern 0x2)
   25 |       2'b10: result_o = xor_tmp;
      |       ^~~~~
%Warning-CASEINCOMPLETE: ../src/design_1.sv:21:5: Case values incompletely covered (example pattern 0x3)
   21 |     case (sel_i)
      |     ^~~~
%Error: Exiting due to 6 warning(s)
make: *** [makefile:47: lint] Error 1
~/gi/c/l/ex01/tb $
```

The example code has 6 warnings. For the first one — `%Warning-IMPLICIT`, the log specifically states the violation, a suggestion to tackle it, and also a link for more information on that particular warning. The original design, indeed, has a typo in line 16, fix it and lint again.

```
~/gi/c/l/ex01/tb $ make lint
------------------> LINT CHECK <------------------
%Warning-CASEOVERLAP: ../src/design_1.sv:25:7: Case values overlap (example pattern 0x2)
   25 |       2'b10: result_o = xor_tmp;
      |       ^~~~~
                      ... For warning description see https://verilator.org/warn/CASEOVERLAP?v=5.003
                      ... Use "/* verilator lint_off CASEOVERLAP */" and lint_on around source to disable this message.
%Warning-CASEINCOMPLETE: ../src/design_1.sv:21:5: Case values incompletely covered (example pattern 0x3)
   21 |     case (sel_i)
      |     ^~~~
%Error: Exiting due to 2 warning(s)
make: *** [makefile:47: lint] Error 1
~/gi/c/l/ex01/tb $
```

This time, linting alerts the violation of `CASEOVERLAP`, which is, according to the log, "case values overlap" because of the value `2'b10` is repeated twice. Fix this error at line 25 and lint again, and there are no warnings or errors.

```
~/gi/c/l/ex01/tb $ make lint
------------------> LINT CHECK <------------------
~/gi/c/l/ex01/tb $
```

Now, the design is free of syntax errors and potential bugs, but, still, the next step will prove its correctness.

**Question**

Change the value `[2:0]` to `[1:0]` in the line 13 of `design_1.sv`?
What is the error that lint reports?
Click on the link that Verilator provides and read the information.

### 3.3 Verification

#### 3.3.1 Setup

This process is to verify the correctness of the design — for each set of inputs, only one set of outputs is right. Although designers could drive a list of inputs and compare the corresponding outputs they've already computed with the design's outputs — directed tests, verifying a design with a vast number of inputs will be improbable and tedious. Hence, randomly driving the inputs and automatically monitoring the outputs will provides a more accurate and reliable verification methodology as in Figure 3.

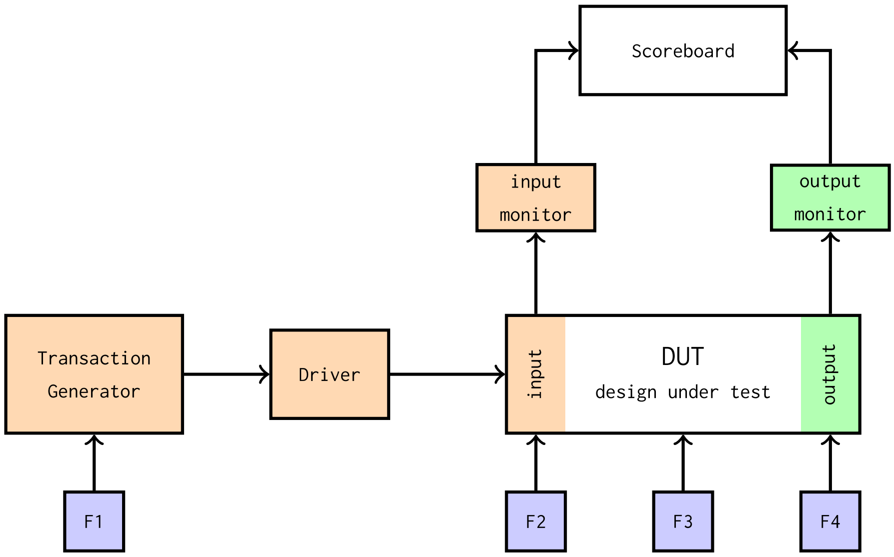


Figure 3: Verification methodology

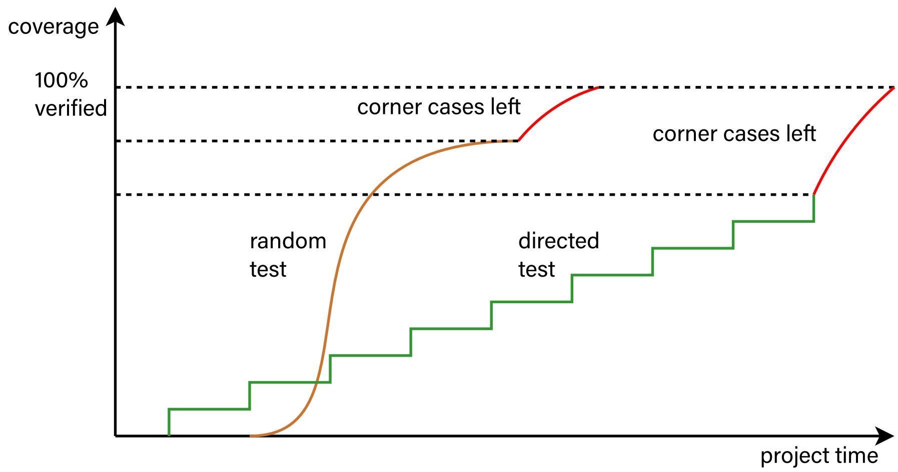


Figure 4: Compare time to verify a design

**!** → Do NOT modify `tb_top.cpp`. You may do so later.

First, preparing `top.sv`, which is alreay written and placed into `tb` directory.

```
1  module top (
2    // inputs
3    input logic        clk_i,
4    input logic [2:0]  data0_i,
5    input logic [2:0]  data1_i,
6    input logic [1:0]  sel_i   ,
7
8    // outputs
9    output logic [2:0] result_o
10 );
11
12   design_1 dut (
13     .data0_i (data0_i ),
14     .data1_i (data1_i ),
15     .sel_i   (sel_i   ),
16     .result_o(result_o)
17   );
18
19   always @(posedge clk_i) begin : proc_assertions
20     if (sel_i == 2'b00)
21       assert (result_o == '0);
22     if (sel_i == 2'b01)
23       assert (result_o == (data0_i & data1_i));
24     if (sel_i == 2'b10)
25       assert (result_o == (data0_i | data1_i));
26     if (sel_i == 2'b11)
27       assert (result_o == (data0_i ^ data1_i));
28   end
29
30 endmodule : top
```

Second, according to Figure 3, the design under test, `design_1`, is instantiated in `top.sv`, line 12 — 17. Then, line 19 — 28 set some assertions or required contraints (F3), which can easily be inferred from Figure 2. `assert` keyword with the expression inside will signal the EDA to stop when the expression failed and report it on the monitor or screen. As this design is simple and has four cases in total, this example uses `if` to set up the "check point." You may notice, even though the design is de facto a combinational logic, you have to set a clock because `assert` needs it as a reference time to check the expression. F2 and F4 are already configured in `tb_top.cpp`, and `driver.cpp` will set the inputs of the DUT.

```
#define MAX_SIM 20

void set_random(Vtop *dut, vluint64_t sim_unit) {
  dut->data0_i = rand()%8;
  dut->data1_i = rand()%8;
  dut->sel_i   = rand()%4;
}
```

`MAX_SIM` is the number of inputs generated. Basically, illustrated in the code above, the randomized values are set by following a syntax of `dut->` and the inputs with `rand()` function. This function in C++ generates a 32-bit number, but because an input varies in width, the modulo operator will restrict the value, which `rand()` assigns to the input.

> **Question**
>
> If `data0_i` varies from 3 to 7, how to set it in `driver.cpp`?

### 3.3.2 Simulation

In directory `tb`, run `make sim` to simulate.

```
~/gi/c/l/ex01/tb $ make sim
<> BUILD ---------------------
...
<> SIMULATING ----------------
[0] %Error: top.sv:27: Assertion failed in TOP.top.proc_assertions: 'assert' failed.
%Error: top.sv:27: Verilog $stop
Aborting...
make: *** [makefile:58: sim] Aborted (core dumped)
~/gi/c/l/ex01/tb $
```

The error states that the assertion at line 27 failed. In `top.sv`, that line indicates the design failed to handle the case `2'b11`, the `xor` operator. To investigate the waveform for debugging, run `make wave`. Figure 5 shows the result `3'b101` instead of `3'b010`.

A little bit further, `design_1.sv` in `src` directory containing the source code has a bug in line 18. The operator is intentionally altered, which is `xnor` instead. Fix it and run simulation again.

```
~/gi/c/l/ex01/tb $ make sim
<> BUILD ---------------------
...
<> SIMULATING ----------------
~/gi/c/l/ex01/tb $
```

FABULOUS!! The design is successfully debugged!

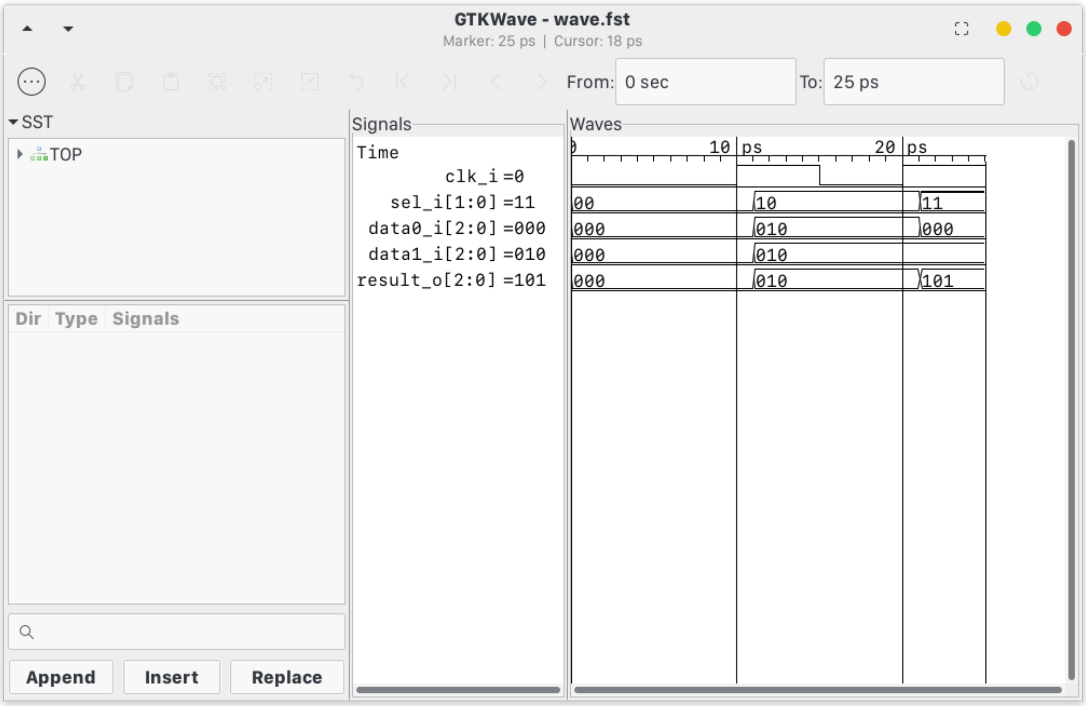


Figure 5: Waveform of the first simulation

## 3.4 Implementation

### 3.4.1 Setup

To implement the design in FPGA, `wrapper.sv` is to instantiate the design and connect FPGA pins to it. In this example, DE2 is used, so you could look up the pin names in this link. You may check DE2 User Manual in `quartus` directory. `design_1.sv` uses switches for inputs and LEDs for outputs. The example `wrapper.sv` is already written and placed into `quartus` directory.

```
module wrapper (
  // inputs
  input  logic [7:0] SW,
  // outputs
  output logic [2:0] LEDR
);

  design_1 dut (
    .data0_i (  SW[2:0]),
    .data1_i (  SW[5:3]),
    .sel_i   (  SW[7:6]),
    .result_o(LEDR[2:0])
  );

endmodule : wrapper
```

**Question**

Explain how FPGA's pins connect to the design.

It's time to use Quartus.

1. Create Quartus Project: `File` → `New Project Wizard`
   The working directory is `quartus` in the example project.
   Name the new project `wrapper` to match the `wrapper.sv` module.
2. Select all source codes, `design_1.sv` and `wrapper.sv`.
   Make sure in the **Type** column, all files are **SystemVerilog HDL**.
3. For DE2, the device is `EP2C35F672C6`.
4. Import pin assignment of DE2. The file is `de2_pin.qsf` placed in `quartus` directory already.
5. `Ctrl L` or `Processing` → `Start Compilation`.
   Mkae sure there are no errors or critical warnings.
6. Connect DE2 with your computer via **Blaster**.
7. Check the connection: `Tools` → `Programmers`
8. If the device is not connected, select `Hardware Setup`
9. Click `Start` to start the implementation.

# 4 Sequential Logic/FSM Modeling

| **Problem** |
| --- |
| Design a counter to count each time a button is pressed then display the number using 7-segment LED. |

## 4.1 Analysis

The design requires a button and a 7-segment LED. It, absolutely, needs a clock and an active low reset. Based on the requirements:

- A module to receive the signal from a button and its output will be high in only one cycle if the button is pressed.
- A counter to count, let's say, from 0 to 9 (maximum).
- A module to decode a binary number to data that drives a 7-segment LED.

The design is shown in Figure 6

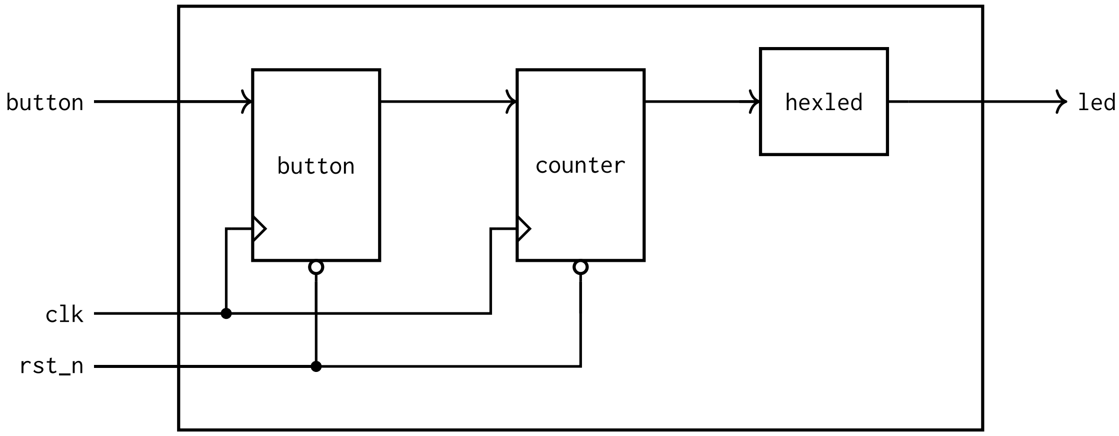


Figure 6: Block diagram of the counter

The button module is designed as an FSM with three states: IDLE, PRESS, and HOLD.

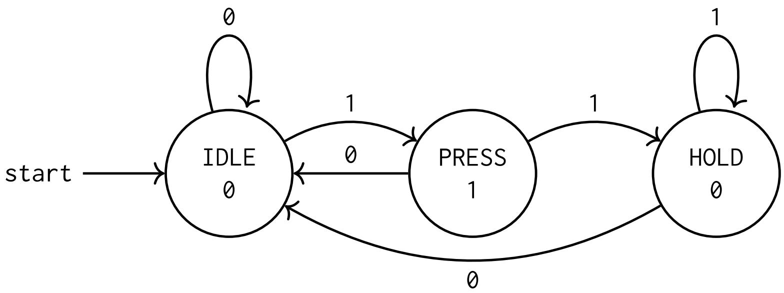


Figure 7: Button FSM

## 4.2 Design

This project is `ex02`.

## 4.3 Verification

Before diving into linting and simulating, here are some notices regarding writing assertions:

1. This "more complex" example needs to assert the data between module `button` (output) and `counter` (input), so the assertions have to be placed in `design_2.sv`. The "clock requirement" of using assertions is satisfied.
2. `$past` keyword is to sample the data in the previous cycle, and thus line 2–6 and line 14 are needed to sample the write data. Also, When using `$past`, remember to AND the `pastvld`.
3. Line 1 and 15 are to make sure the assertions are checked when running simulation with Verilator but ignored when running Quartus.

```
1  `ifdef VERILATOR
2    /*verilator lint_off UNUSED*/
3    logic pastvld;
4    always @(posedge clk_i) begin : proc_setup_past
5      pastvld <= 1'b1;
6    end
7
8    always @(posedge clk_i) begin : proc_assertions
9      if (pastvld && $past(inc))
10       assert(!inc);
11
12     assert(counter <= 4'h9);
13   end
14   /*verilator lint_on UNUSED*/
15 `endif
```

**Question**

Explain the two assertions in `design_2.sv`.

Here are some notices regarding `driver.cpp`:

1. This design has the maximum number of 9, so `MAX_SIM` must be set to a large number.

2. Because `driver.cpp` is a C++ file, assigned values could be manipulated usefully. Such as, the reset should be low for several cycles from the beginning, so using `sim_unit`, illustrated in Figure 8, as in line 4, it will certainly be low for 4 cycles from the beginning. Yet, it is just a part of the story.

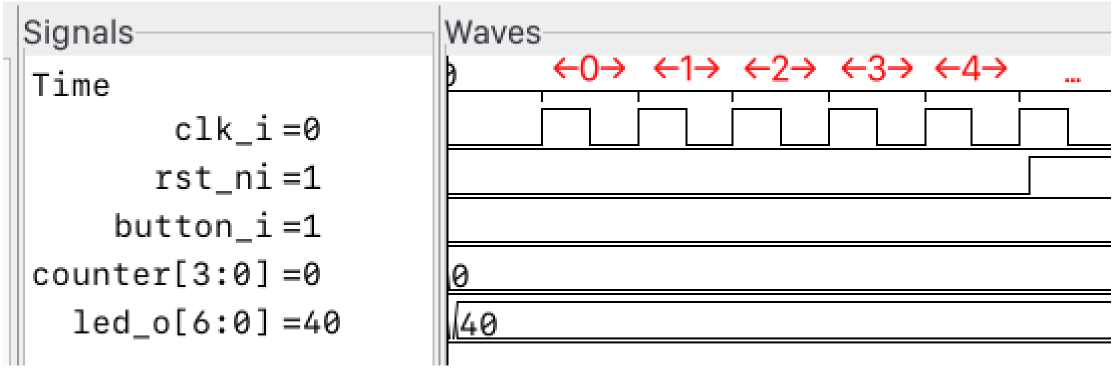


Figure 8: `sim_unit`

3. The reset should be 0 at some points, to prove that the counter will reset to 0 and count again, but its probability should be low to show the case that the counter only counts to 9 and stop. The expression `rand()%30 != 0` has $1/30 = 3.3\%$ chance of being FALSE or 0. It is sufficient to create a reset to verify this design.

```
1 #define MAX_SIM 200
2
3 void set_random(Vtop *dut, vluint64_t sim_unit) {
4   dut->rst_ni   = (sim_unit > 4) && (rand()%30 != 0);
5   dut->button_i = (rand()%8 >= 2);
6 }
```

**Question**

Compute the probability $\Pr(\texttt{button\_i} = 1)$.
Explain the reason for choosing that value.

**Question**

Lint and debug this design.

## 4.4 Implementation

**Question**

Implement this design into DE2.

# 5 Exercise

**Question**

Design a traffic light that has 3 colors: RED, AMBER, and GREEN. This traffic light cycles through:

- GREEN in 6 seconds
- AMBER in 2 seconds
- RED in 7 seconds
- RED +AMBER in 2 seconds

It also has a button for pedestrians, when one pushes it, the traffic light itself will turn RED.
Try to use assertions to verify the design.