Writing Sequential Circuit in Verilog

Understanding Sequential Circuits

Sequential circuits are digital circuits that store and utilize previous state information to determine their next state. Unlike combinational circuits, which rely solely on current input values to produce outputs, sequential circuits depend on both the current inputs and the previously stored state in memory elements. [Find out more here]

Before implementing any sequential circuit, it is crucial to thoroughly understand its behavior. Start by analyzing its description and the provided timing diagram. Once you have a clear understanding, proceed to create a state diagram or an Algorithmic State Machine (ASM) chart to accurately represent its operation.

With the state diagram or ASM chart in hand, you can then move on to implementing the circuit in Verilog.

Implementing the Sequential Circuits

In this part, we are going to walk you through the implementation of some basic example sequential circuits.

Example #1 : 2 bits counter with RCO signal

In this example, we will implement a 2-bit counter that counts from 0 to 3 when the Trigger signal is asserted. The counter will only change its value on the rising edge of the clock signal. When the Trigger is asserted and the counter reaches a value of 3, it will reset to 0 and assert the RCO (Ripple Carry Output) signal immediately. If the Reset signal is asserted, the counter value will be reset to 0 on the next rising edge of the clock signal. This module has the following ports :

- **Input :**
  - Trigger : Trigger signal to increase counter value.
  - Reset : Reset signal.
  - Clk : Clock signal.
- **Output :**

- DataOut [1:0] : The Counter Value.
- RCO : The Ripple Carry Out signal. (use in case you want to expand this module into 4-bit counter)

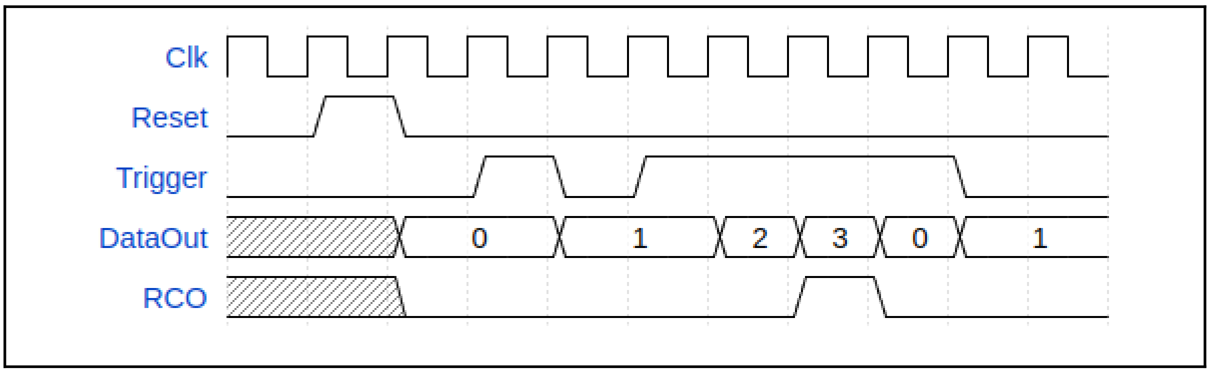


Figure 1 : 2-bits counter with RCO signal Timing Diagram

After reading through the above description we are going to create the state diagram to represent this Module Behavior.

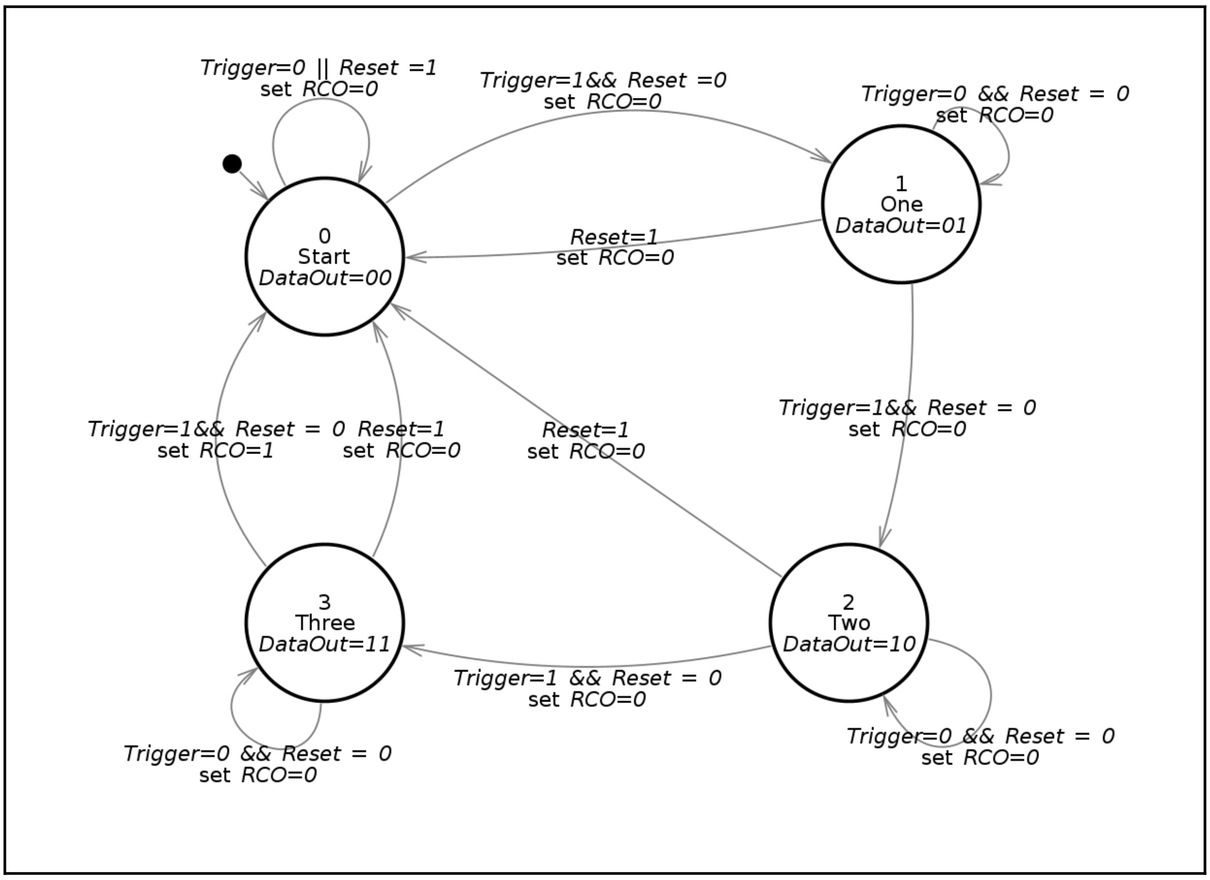
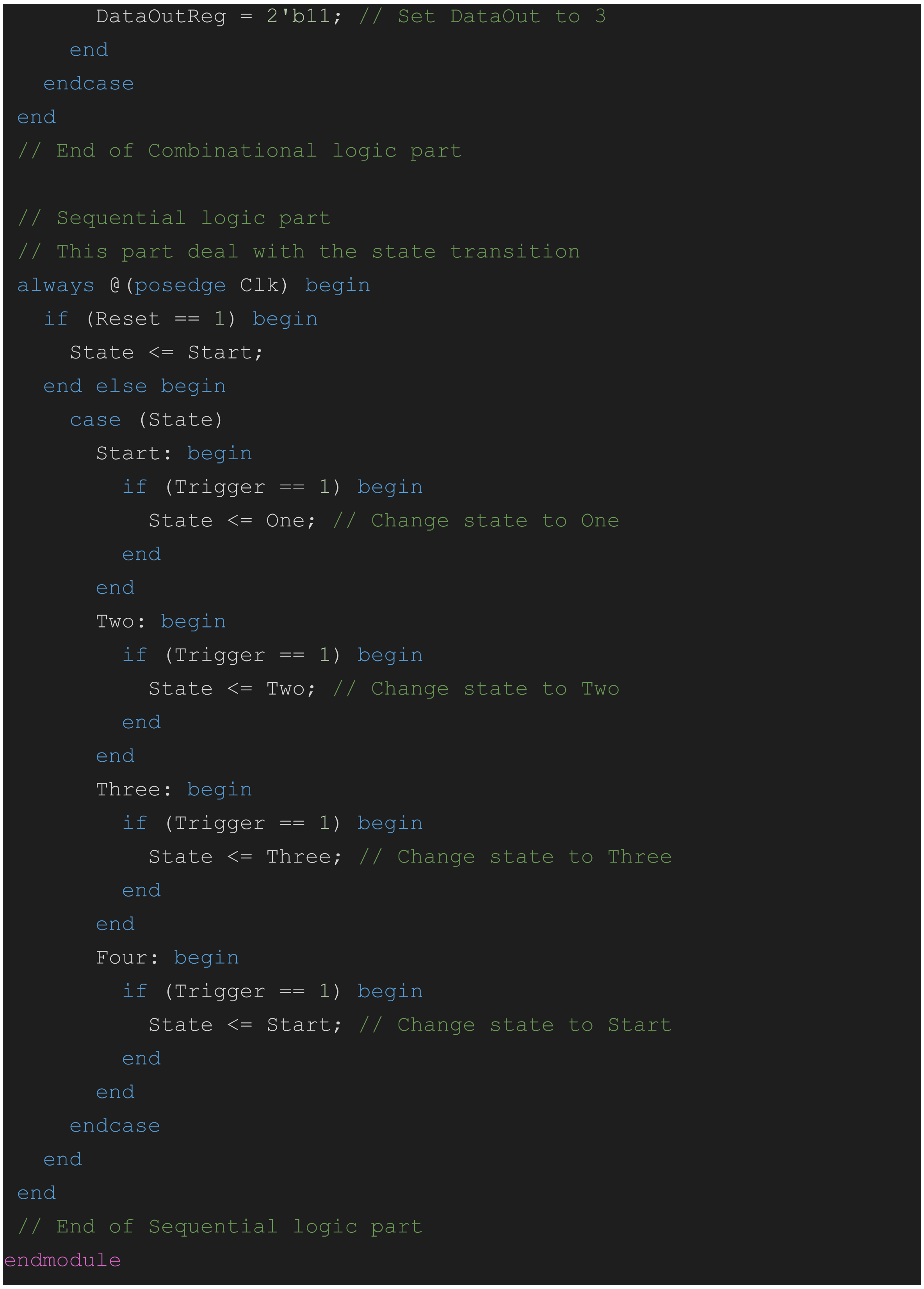


Figure 2 : The State Diagram of the 2-bits counter with RCO Signal

Now, let's proceed to implement this in Verilog. There are several ways to design sequential circuits in Verilog, and the following sections will highlight some common implementation styles.

```
`timescale 1ns / 1ps

module Counter (
   input  wire       Trigger,
   input  wire       Reset,
   input  wire       Clk,
   output wire [1:0] DataOut,
   output wire       RCO
);
 // declare the state
 localparam Start = 2'b00;
 localparam Two   = 2'b01;
 localparam Three = 2'b10;
 localparam Four  = 2'b11;
 // end of state declaration

 // declare the state register
 reg [1:0] State = 2'b00;
 // declare the internal wire/reg
 reg [1:0] DataOutReg = 0;

 // Combinational logic part
 assign DataOut = DataOutReg;
 assign RCO = (State == Three && Trigger == 1'b1 && Reset == 1’b0) ?
1'b1 : 1'b0;
 // This part deal with the output data
 always @(*) begin
   case (State)
     Start: begin
       DataOutReg = 2'b00; // Set DataOut to 0
     end
     Two: begin
       DataOutReg = 2'b01; // Set DataOut to 1
     end
     Three: begin
       DataOutReg = 2'b10; // Set DataOut to 2
     end
     Four: begin
```



```
       DataOutReg = 2'b11; // Set DataOut to 3
     end
   endcase
 end
 // End of Combinational logic part

 // Sequential logic part
 // This part deal with the state transition
 always @(posedge Clk) begin
   if (Reset == 1) begin
     State <= Start;
   end else begin
     case (State)
       Start: begin
         if (Trigger == 1) begin
           State <= One; // Change state to One
         end
       end
       Two: begin
         if (Trigger == 1) begin
           State <= Two; // Change state to Two
         end
       end
       Three: begin
         if (Trigger == 1) begin
           State <= Three; // Change state to Three
         end
       end
       Four: begin
         if (Trigger == 1) begin
           State <= Start; // Change state to Start
         end
       end
     endcase
   end
 end
 // End of Sequential logic part
endmodule
```

Figure 3 : 2-bits counter with RCO signal implementation#1

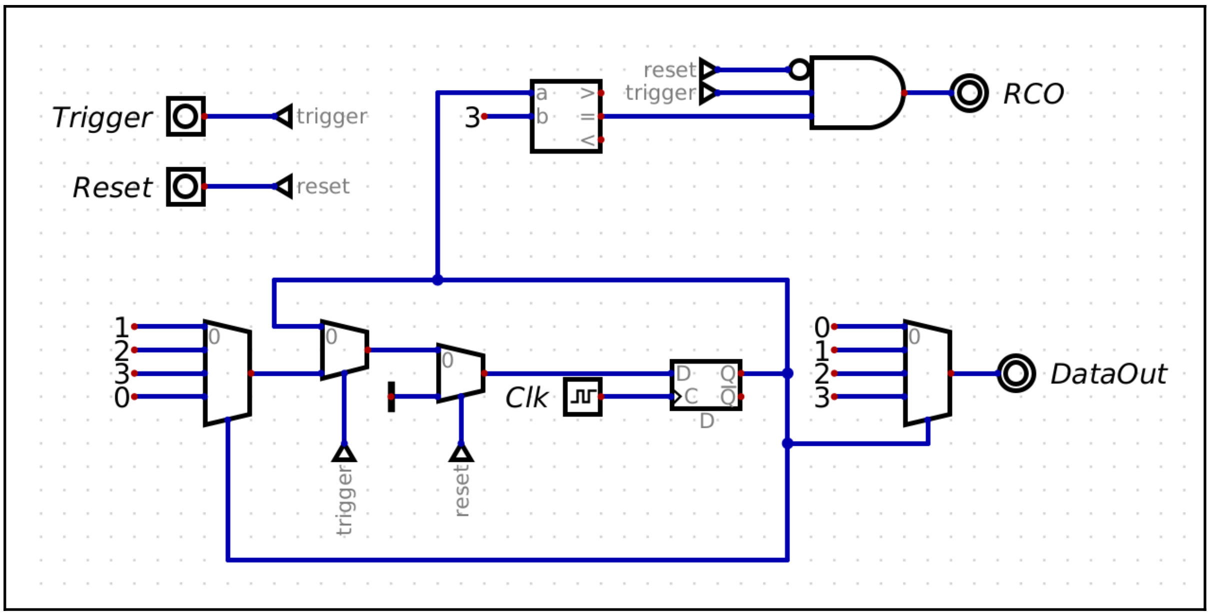
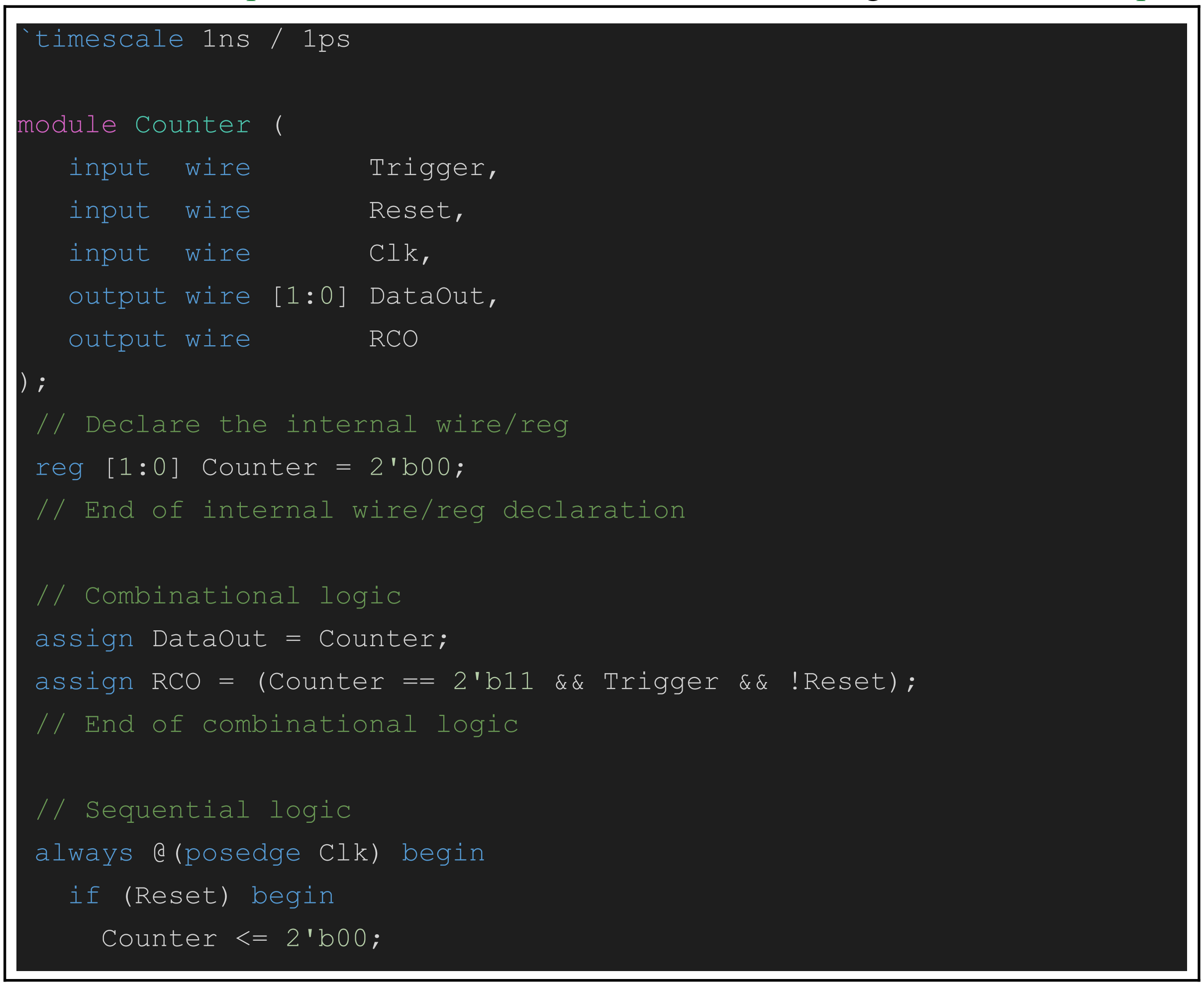


Figure 4 : 2-bits counter with RCO signal implementation#1 visualization

The code in Figure 3 illustrates the coding style where the states of a finite state machine (FSM) are explicitly declared using localparam. We use the always @(posedge Clk) block to describe the state transition. And then we can describe the Output of the module as the combinational logic of State and Input.



```
`timescale 1ns / 1ps

module Counter (
   input  wire       Trigger,
   input  wire       Reset,
   input  wire       Clk,
   output wire [1:0] DataOut,
   output wire       RCO
);
 // Declare the internal wire/reg
 reg [1:0] Counter = 2'b00;
 // End of internal wire/reg declaration

 // Combinational logic
 assign DataOut = Counter;
 assign RCO = (Counter == 2'b11 && Trigger && !Reset);
 // End of combinational logic

 // Sequential logic
 always @(posedge Clk) begin
   if (Reset) begin
     Counter <= 2'b00;
```

```
    end else begin
      if (Trigger) begin
        Counter <= Counter + 1;
      end
    end
  end
  // End of sequential logic
endmodule
```

Figure 5 : 2-bits counter with RCO signal implementation#2

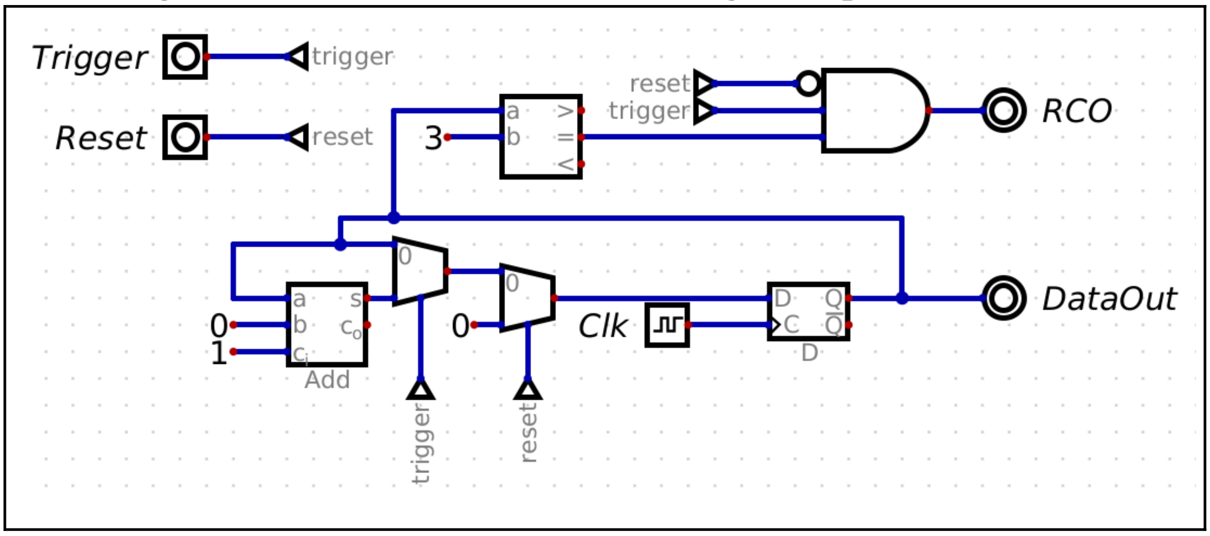


Figure 6 : 2-bits counter with RCO signal implementation#2 visualization

The code in Figure 5 illustrates a coding style where the state is implicitly declared and managed within the register. In this design, the counter value itself serves as both the state and the variable.

Ultimately, both implementations achieve the same goal: a digital circuit that behaves as a 2-bit counter with an RCO signal. You can choose any style that suits your preference, but always ensure the circuit is described with accurate behavior.

Writing Testbenches

Verifying and testing are critical in Verilog (and hardware design in general) because hardware designs, once implemented in silicon, cannot be easily modified. Mistakes in design can lead to costly errors, defective products, or catastrophic failures. This is why we must write testbenches.

Recap: What is testbench?

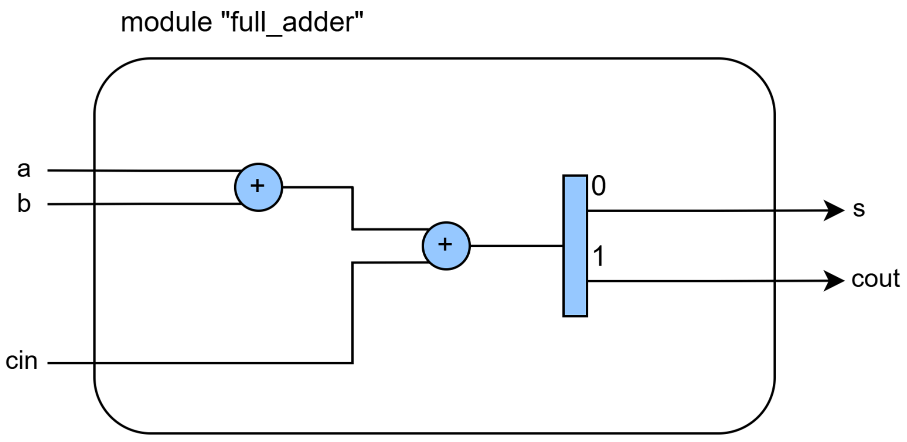
Testbench is a thing/controller which is used for testing our designs. It will simulate the scenario and check if our designs work well or not.

(If you are similar with software development, it is similar to unit testing like JUnit in Prog Meth)

Testbenches for Combinational Logic

For testing combinational logic circuits, you should test all possible input combinations. For example, if you have 4 1-bit inputs, your testbench must have $2^4 = 16$ combinations.

Let's see the example of 1-bit full adder. Our design has 3 input ports (a, b, and cin) and 2 output ports (s and cout). Therefore, our testbench must have $2^3 = 8$ combinations of inputs.

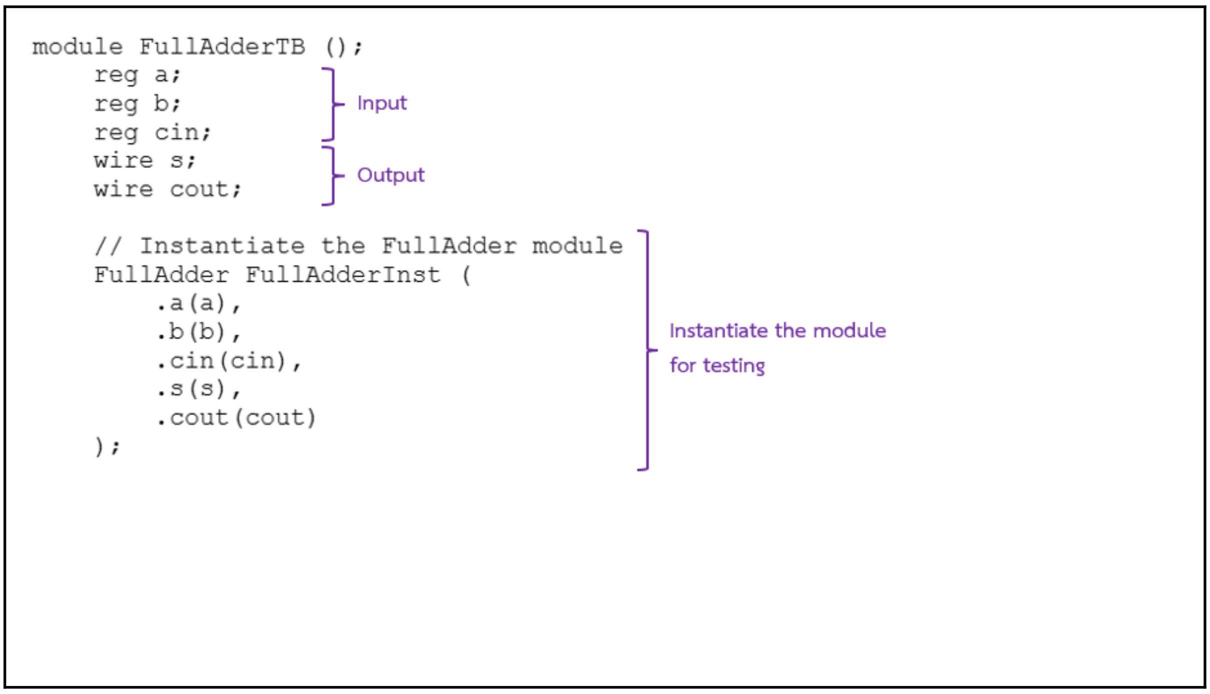


Our testbench must include these cases:

Inputs			Outputs	
a	b	cin	s	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Testbench on Vivado Simulator (Xilinx)

We will create a Verilog file for testbench. The template of the testbench is as follows:



```
module FullAdderTB ();
    reg a;
    reg b;
    reg cin;
    wire s;
    wire cout;

    // Instantiate the FullAdder module
    FullAdder FullAdderInst (
        .a(a),
        .b(b),
        .cin(cin),
        .s(s),
        .cout(cout)
    );
    // instantiate variable
    integer flag = 0;
    integer TestCaseNo = 0;
    integer i;
    integer j;

    // task to check the output
    task check_output;
        input integer TestCaseNo;
        input reg expected_s;  // Expected output
        input reg expected_cout;  // Expected cout
        begin
            if (s !== expected_s || cout !== expected_cout)
            begin
                $error("ERROR: TestCaseNo %0d ", TestCaseNo);
                flag = 1;
            end
        end
    endtask
```

Function to check if each test Case is passed or not

```
     // test cases
     initial begin
         // this is our testbench
         // Test case 1
         a = 0;
         b = 0;
         cin = 0;
         #1;
         check_output(TestCaseNo, 0, 0);

         // Test case 2
         a = 0;
         b = 0;
         cin = 1;
         #1;
         check_output(TestCaseNo, 1, 0);

         // Test case 3
         a = 0;
         b = 1;
         cin = 0;
         #1;
         check_output(TestCaseNo, 1, 0);

         // Test case 4
         a = 0;
         b = 1;
         cin = 1;
         #1;
         check_output(TestCaseNo, 0, 1);

         // 4 more cases...
```

Our test cases
(We must write here)

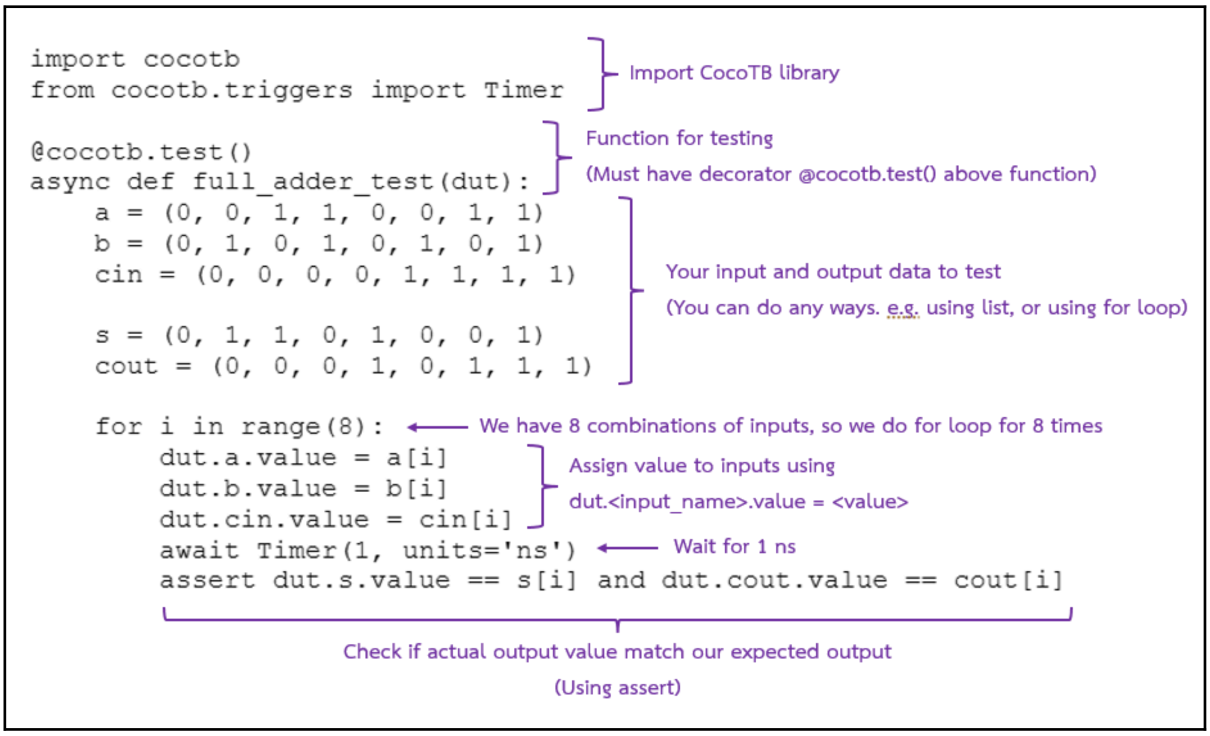
```
         // We will check if the test passed or failed
         if (flag == 0) begin
             $display("All test cases passed!");
         end else begin
             $display("Some test cases failed!");
         end
     end

endmodule
```

If all test cases are passed, then our test is passed

Testbench on CocoTB

As we know, writing testbench using Verilog is a bit painful. So, we will use Python for our testbench. We will use CocoTB and Pytest libraries to simulate scenarios of our design.

To write a testbench, we must create a Python (.py) file. The format of the code inside this file is as follows:



```
import cocotb
from cocotb.triggers import Timer

@cocotb.test()
async def full_adder_test(dut):
    a = (0, 0, 1, 1, 0, 0, 1, 1)
    b = (0, 1, 0, 1, 0, 1, 0, 1)
    cin = (0, 0, 0, 0, 1, 1, 1, 1)

    s = (0, 1, 1, 0, 1, 0, 0, 1)
    cout = (0, 0, 0, 1, 0, 1, 1, 1)

    for i in range(8):
        dut.a.value = a[i]
        dut.b.value = b[i]
        dut.cin.value = cin[i]
        await Timer(1, units='ns')
        assert dut.s.value == s[i] and dut.cout.value == cout[i]
```

Import CocoTB library
Function for testing
(Must have decorator @cocotb.test() above function)
Your input and output data to test
(You can do any ways. e.g. using list, or using for loop)
We have 8 combinations of inputs, so we do for loop for 8 times
Assign value to inputs using
dut.<input_name>.value = <value>
Wait for 1 ns
Check if actual output value match our expected output
(Using assert)

- We will assign the value of inputs by using dut.<input_name>.value = <value>
- We will retrieve the value of outputs after assigning value of input by using dut.<output_name>.value
- We will use Timer which will fire after the specified simulation time period has elapsed

To run the CocoTB testbench, you must activate your Conda environment first.

```
conda activate <cocotb_conda_env_name>
```

Then go to directory which include this Python file, then run the command:

```
make
```

Testbenches for Sequential Logic

For testing sequential logic circuits, you should design a timing diagram of the circuit first, then write the testbench depending on the timing diagram.

Let's see an example of a debouncer, which stabilizes the input signal by eliminating noise caused by signal bouncing. The module has the following ports:

- **Inputs**:
  - Reset: Resets the module and initializes the output to 0.
  - Clk: Clock signal for synchronization.
  - DataIn: Input signal that needs to be debounced.
- **Output**:
  - DataOut: Debounced output signal.
- **Parameter**:
  - **DebounceTime**: Sampling rate for DataIn Signal.
  - **CounterWidth**: The width of the counter register.

The **Debouncer Module** operates by sampling the DataIn signal every **DebounceTime** clock cycle. For example, if the **DebounceTime** parameter is set to 10, the module samples the DataIn input once every 10 clock cycles. To achieve this, you must configure the **CounterWidth** parameter to ensure the counter can count up to **DebounceTime**.

Additionally, when the Reset signal is asserted, the module will reset its internal state, and the DataOut value will be set to 0 at the next clock cycle.

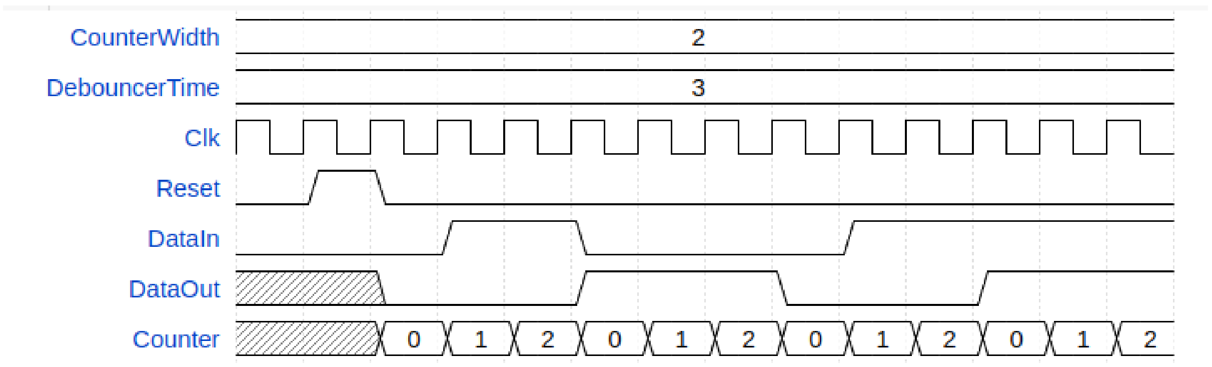
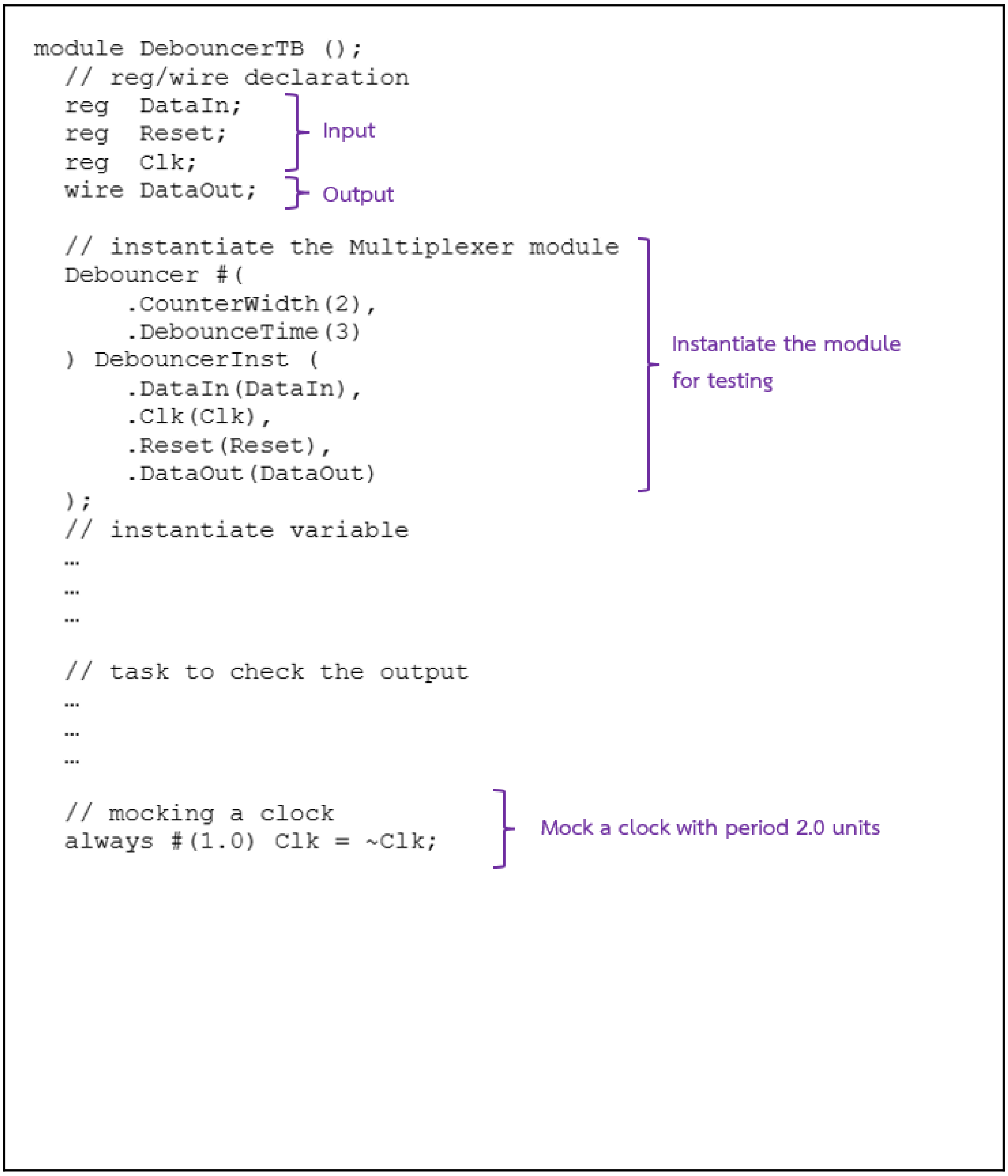


Figure 7: Debouncer Timing Diagram (when DebouncerTime is 3)

Our testbench must follow the above timing diagram, in the other meaning, you must assign inputs and check the outputs according to the timing diagram.

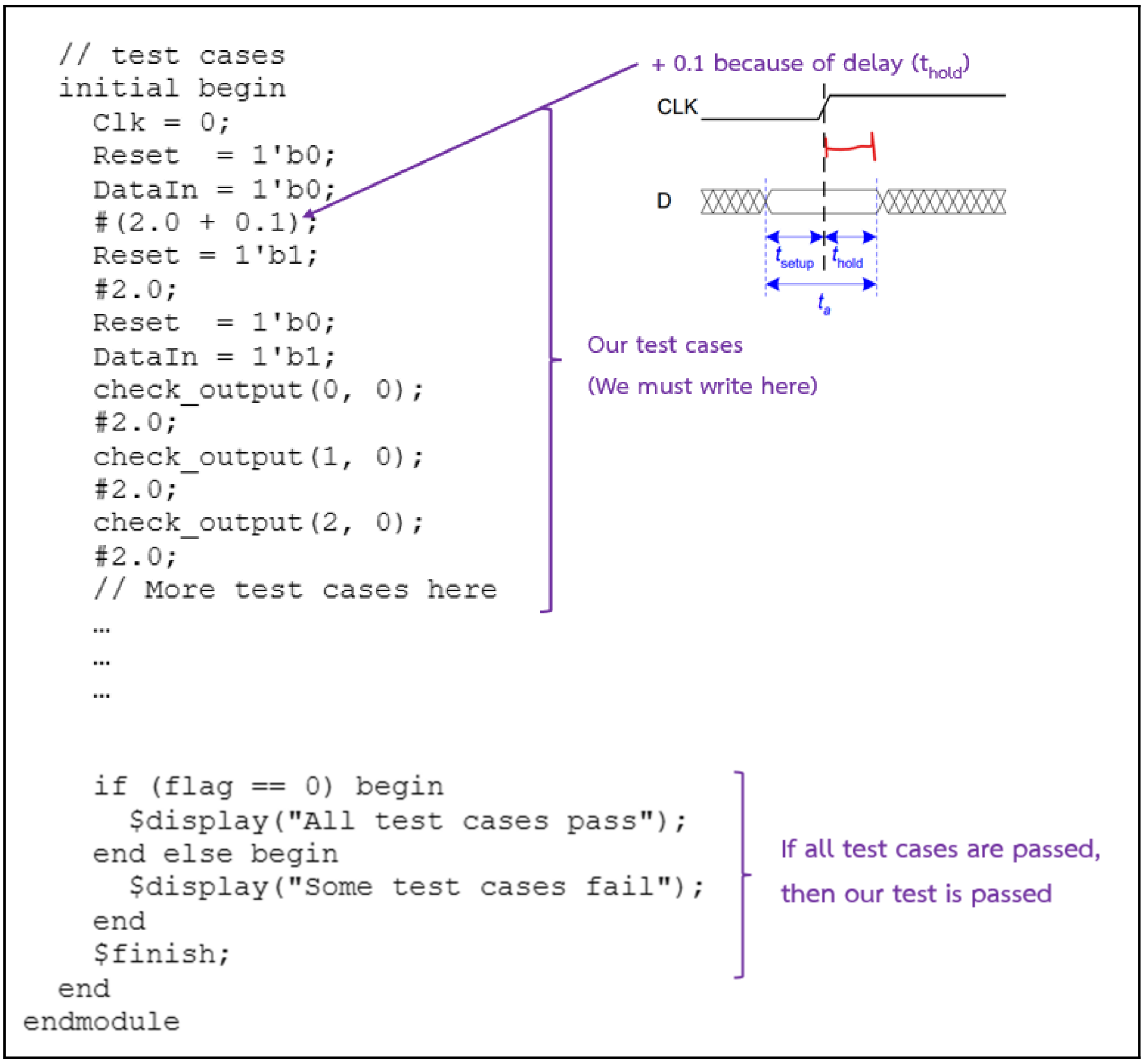
Testbench on Vivado Simulator (Xilinx)

In Verilog, the testbench of a sequential logic circuit is almost the same as a combinational logic circuit, except we must mock a clock.



```
module DebouncerTB ();
  // reg/wire declaration
  reg  DataIn;
  reg  Reset;
  reg  Clk;
  wire DataOut;

  // instantiate the Multiplexer module
  Debouncer #(
      .CounterWidth(2),
      .DebounceTime(3)
  ) DebouncerInst (
      .DataIn(DataIn),
      .Clk(Clk),
      .Reset(Reset),
      .DataOut(DataOut)
  );
  // instantiate variable
  …
  …
  …

  // task to check the output
  …
  …
  …

  // mocking a clock
  always #(1.0) Clk = ~Clk;
```

Input
Output
Instantiate the module for testing
Mock a clock with period 2.0 units



```
  // test cases
  initial begin
    Clk = 0;
    Reset  = 1'b0;
    DataIn = 1'b0;
    #(2.0 + 0.1);
    Reset = 1'b1;
    #2.0;
    Reset  = 1'b0;
    DataIn = 1'b1;
    check_output(0, 0);
    #2.0;
    check_output(1, 0);
    #2.0;
    check_output(2, 0);
    #2.0;
    // More test cases here
    …
    …
    …


    if (flag == 0) begin
      $display("All test cases pass");
    end else begin
      $display("Some test cases fail");
    end
    $finish;
  end
endmodule
```

Testbench on CocoTB

Testbench of a sequential logic circuit is almost the same as a combinational logic circuit, but we must create a clock.

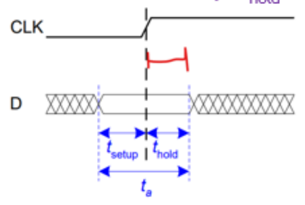
```
import cocotb
from cocotb.triggers import Timer
from cocotb.clock import Clock

@cocotb.test()
async def DebouncerTB(dut):
    # create the clock
    cocotb.start_soon(Clock(dut.Clk, 1, units="ns").start())

    # reset
    dut.Reset.value = 0
    dut.DataIn.value = 0
    await Timer(1.2, units="ns")
    dut.Reset.value = 1
    await Timer(1, units="ns")
    dut.Reset.value = 0
    dut.DataIn.value = 1
    assert dut.DataOut.value == 0
    assert dut.DebouncerInst.Counter == 0
    await Timer(1, units="ns")
    assert dut.DataOut.value == 0
    assert dut.DebouncerInst.Counter == 1
    await Timer(1, units="ns")
    assert dut.DataOut.value == 0
    assert dut.DebouncerInst.Counter == 2
    await Timer(1, units="ns")
    …
    …
    …
```

Import CocoTB library

Function for testing
(Must have decorator @cocotb.test() above function)

Mock a clock
(Mock on input Clk)

+ 0.2 because of delay (t_{hold})



Our testbench
(Same as combinational logic)

- We can create and start a clock with 50% duty cycle by using:

```
cocotb.start_soon(Clock(dut.<clock_signal>, <period>,
units=<unit>).start())
```

Note:
To run the CocoTB testbench, you must activate your Conda environment first.

```
conda activate <cocotb_conda_env_name>
```

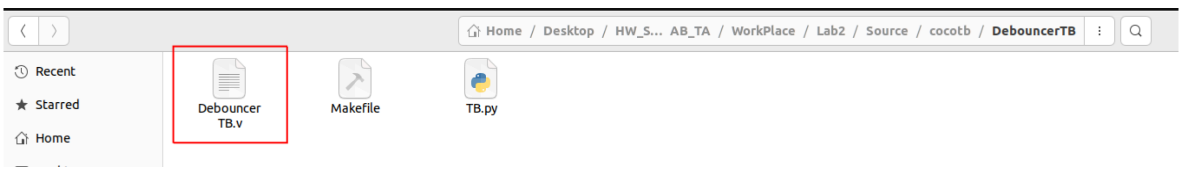
Then go to directory which include this Python file, then run the command:

```
make
```

Simulating Module with Parameter using CocoTB

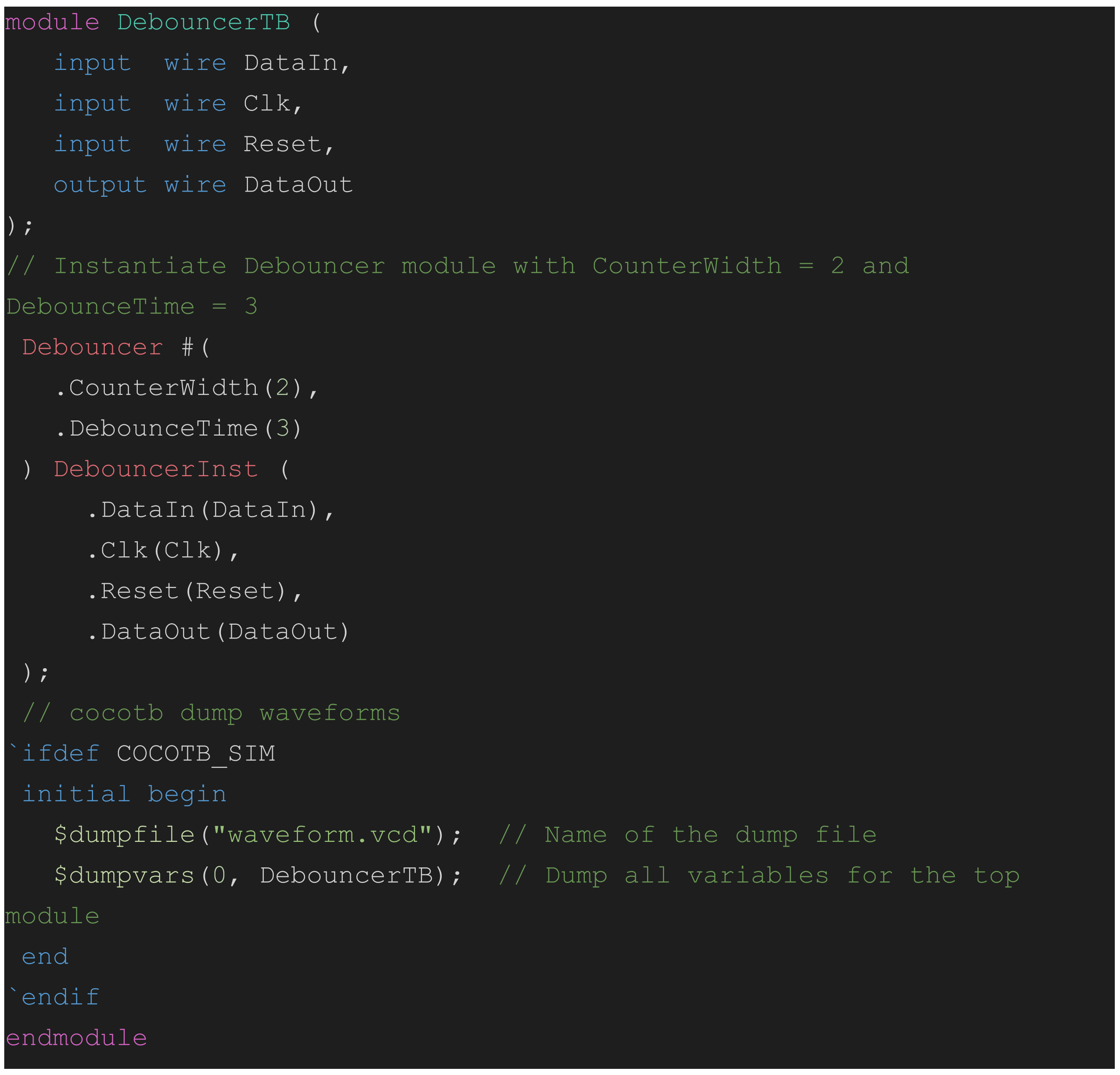
Due to certain limitations, we cannot directly assign parameters to the module we want to test in Cocotb. Instead, we need to instantiate a module with the desired parameters and run the test on that instance.

For example, to simulate the **Debouncer** module with a CounterWidth of 2 and a DebounceTime of 3 clock cycles, we create a new module, such as **DebouncerTB**, which instantiates the Debouncer with those parameters.



Home / Desktop / HW_S... AB_TA / WorkPlace / Lab2 / Source / cocotb / DebouncerTB
Recent
Starred
Home
DebouncerTB.v
Makefile
TB.py

Figure 8 : DebouncerTB Folder Structure



```
module DebouncerTB (
   input  wire DataIn,
   input  wire Clk,
   input  wire Reset,
   output wire DataOut
);
// Instantiate Debouncer module with CounterWidth = 2 and
DebounceTime = 3
 Debouncer #(
   .CounterWidth(2),
   .DebounceTime(3)
 ) DebouncerInst (
     .DataIn(DataIn),
     .Clk(Clk),
     .Reset(Reset),
     .DataOut(DataOut)
 );
 // cocotb dump waveforms
`ifdef COCOTB_SIM
 initial begin
   $dumpfile("waveform.vcd");  // Name of the dump file
   $dumpvars(0, DebouncerTB);  // Dump all variables for the top
module
 end
`endif
endmodule
```

Figure 9 : DebouncerTB.v file

In the makefile, we need to include all the necessary files in the VERILOG_SOURCES and set the TOPLEVEL variable to the name of the top-level module being tested. This part is already provided to you in this lab.

```
# Makefile

# defaults
SIM ?= icarus
TOPLEVEL_LANG ?= verilog

VERILOG_SOURCES += $(PWD)/../../src/Debouncer.v $(PWD)/DebouncerTB.v
# Debouncer = Real Debouncer module
# DebouncerTB = Module that instantiate Debouncer Module with certain
parameter
# use VHDL_SOURCES for VHDL files

# TOPLEVEL is the name of the toplevel module in your Verilog or VHDL
file
TOPLEVEL = DebouncerTB

# MODULE is the basename of the Python test file
MODULE = TB

# include cocotb's make rules to take care of the simulator setup
include $(shell cocotb-config --makefiles)/Makefile.sim
```

Figure 10 : Makefile

Now you can modify the **TB.py** file to test the **Debouncer** module configured with a CounterWidth of 2 and a DebounceTime of 3 clock cycles.