Lab Member

Table Number : <Insert your table number here>

Name	Student Number
Worralop Srichainont	6632200221
Chavapon Suwanna	6630055221
Akkharachai Yongsuwankul	6630357821
Arin Ariyata	6631360821

Objectives

1. Understand Debouncing.
2. Understand Time-Division Multiplexing.
3. Understand BCD Counter.

Debouncing The Switch Input

Given that a switch is a mechanical device, a mechanical contact between the two surfaces would cause a bounce (the same idea when dropping a ball to the floor. The ball may bounce a few times before stopping at the floor). The switch also causes a bounce when it is being pressed. From an oscilloscope, the bounce is observed as Figure 1. If a program does not carefully avoid the bounce, we may easily observe multiple clicks from only one physical click.

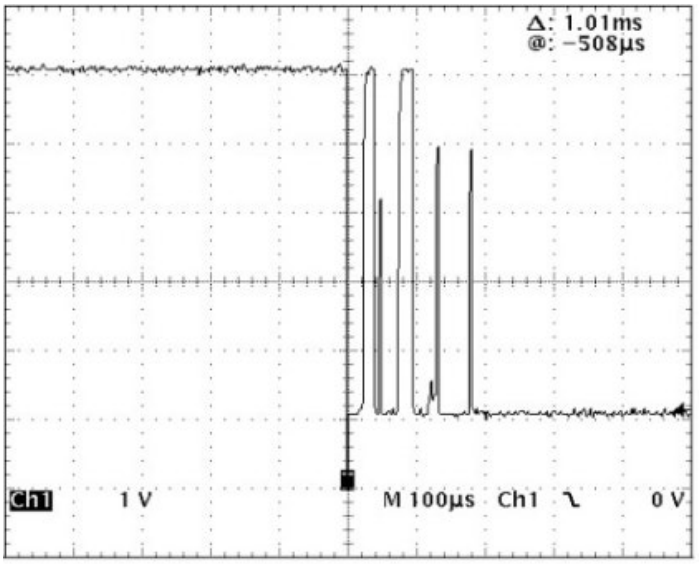


Figure 1: Bouncing Button

A way to avoid reading multiple clicks from a single click is called **debouncing**. Depending on the size and the mechanical properties of the switch, the bounce may last several milliseconds. There are several ways to avoid bounce. Some methods require extra hardware. However, the naïve software solution is to delay a few milliseconds before reading the next input.

Binary-coded Decimal Counter

A Binary-Coded Decimal (BCD) counter is a type of digital counter where each decimal digit (0-9) is represented by its binary equivalent in a 4-bit format. For example, the decimal number 5 would be represented as 0101 in BCD. BCD counters are commonly used in digital systems where the output needs to display decimal numbers, such as in digital clocks or calculators. A BCD counter typically counts in decimal rather than binary, making it easier to display values directly on devices like 7-segment displays. In a BCD counter, each digit is incremented separately, and when it reaches 9, it resets to 0 and carries over to the next higher digit.

Decimal Number	Hexadecimal Number	8-digit BCD
0	0	0000 0000
1	1	0000 0001
2	2	0000 0010
3	3	0000 0011
4	4	0000 0100
5	5	0000 0101
6	6	0000 0110
7	7	0000 0111
8	8	0000 1000
9	9	0000 1001
10	A	0001 0000
11	B	0001 0001
12	C	0001 0010
13	D	0001 0011

Time-Division Multiplexing

To drive a seven-segment display (whether it is common anode or common cathode), each digit would require 9 wires (a to g, dot, common ground or common VCC). With several digits of a seven-segment display, the number of wires would intuitively multiplied. For example, 4 digits of seven-segment displays may require up to 33 wires (a to g and dot for each digit with a shared common wire). It is not practical to have so many wires. To share (reduce) physical wires, Time-Division Multiplexor is introduced.

With time-division multiplexing, we can share 8 wires (a to g and dot) among the displays. Only a digit will be active at a time. If the segments turn on and off at the appropriate rate (i.e. 15 frames per second or more), the observer would see it as if all segments are on at the same time.--hence the term time-division multiplexing. This way, four digits of seven-segment displays can be connected with only 12 wires (8 from a to g and dot + 4 for activating digits).

Lab Exercise

In this lab exercise, you are tasked with developing a BCD counter that counts from 0 to 9999 and displays the value on a 4-digit 7-segment display. The system uses five buttons, each assigned a specific function as listed below:

- **BTNL**: Increases the BCD counter by 1.
- **BTNR**: Increases the BCD counter by 10.
- **BTNU**: Increases the BCD counter by 100.
- **BTND**: Increases the BCD counter by 1000.
- **BTNC**: Resets the counter.

If the counter value exceeds 9999, it will wrap around by taking the modulo 10000 of the result.

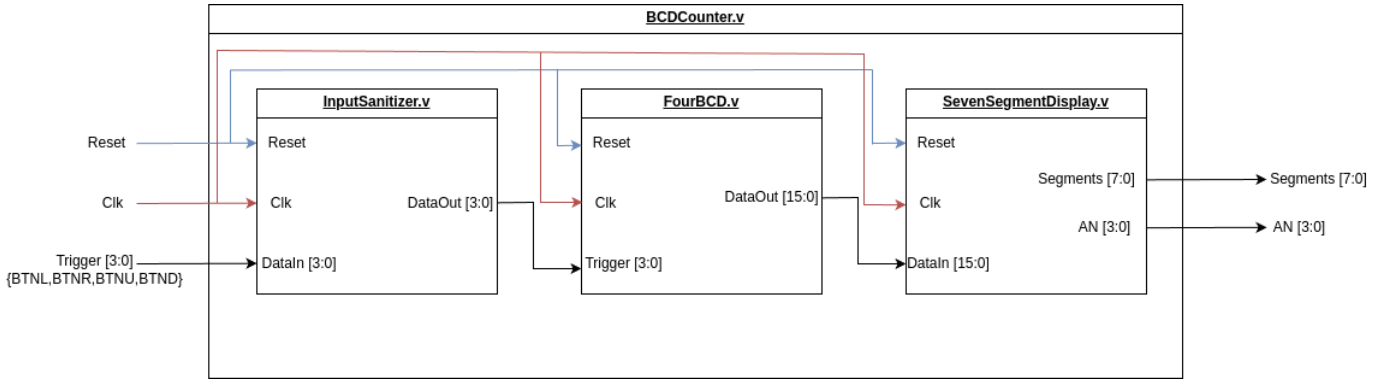


Figure 2 : System Overview

Figure 2 illustrates the system overview to be implemented in this lab exercise. The system consists of four modules, each with a specific role as described below:

1. **BCDCounter**: This is the top-level module responsible for integrating and connecting all submodules.
2. **InputSanitizer**: Handles debouncing of the on-board switches and passes the cleaned signal to the single pulser.
3. **FourBCD**: Implements the BCD counter that counts values from 0 to 9999.
4. **SevenSegmentDisplay**: Displays the current value of the BCD counter on the 7-segment display.

Before you begin each part, please make sure that you activate your Python virtual environment which contain cocotb via **conda** by using

```
conda activate <cocotb-env-name>
```

Also, make sure that the project structure for Lab 2 isn't modified from the original source to avoid issues with Makefile for each testbench.

Part 1 : Input Sanitizer

In this part of the lab, you will complete the Input Sanitizer Module. This module eliminates noise and generates a clean single pulse for each button press, ensuring accurate increments to the BCD Counter value without bounce.

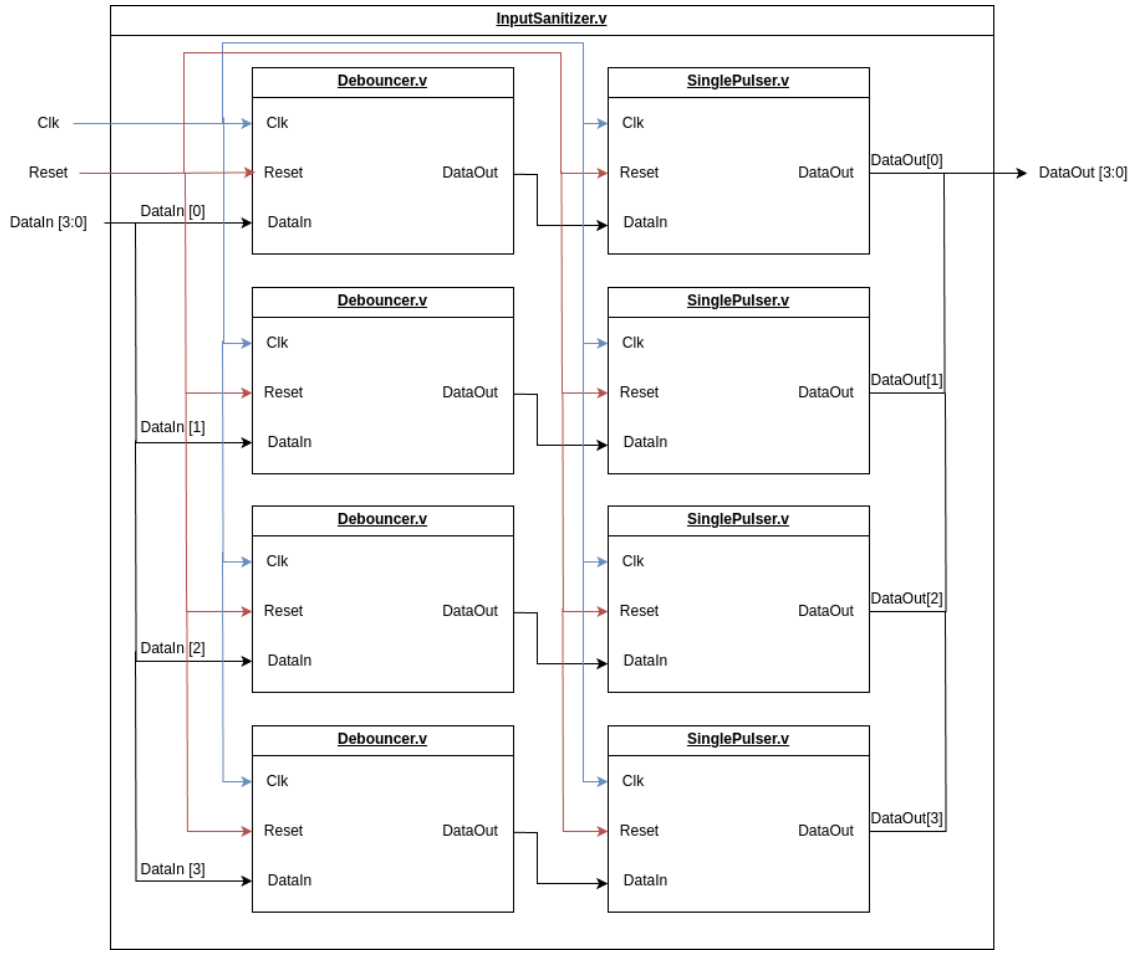


Figure 3 : InputSanitizer Module

Figure 3 illustrates the implementation of the InputSanitizer Module.

This task is divided into three subparts: **Debouncer**, **Single Pulser**, and **System Integration and Validation.**

Part 1.1 : Debouncer

In this section, you will implement the **Debouncer Module**, 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.

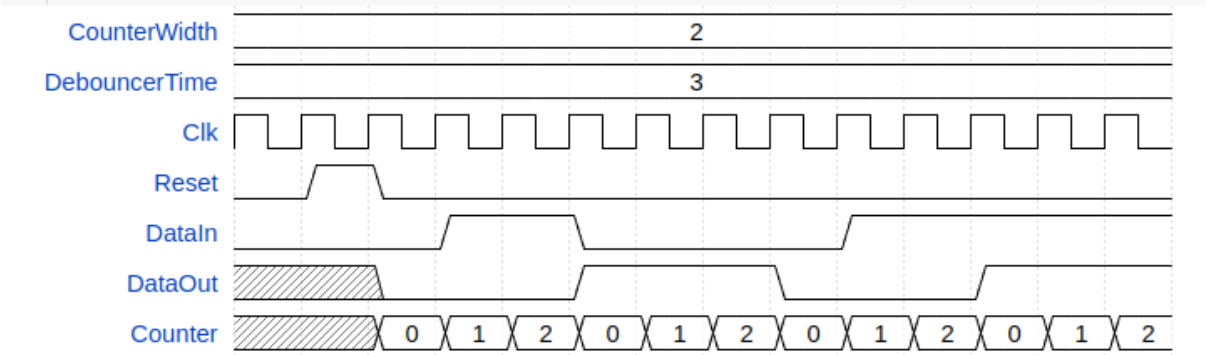


Figure 4: Debouncer Timing Diagram (when DebouncerTime is 3)

Figure 4 illustrates the timing diagram of the Debouncer Module. This diagram demonstrates how the DataIn signal is sampled at regular intervals and stabilized at the DataOut port.

Instruction

1. Create a new project and import all the necessary files from the following source: https://github.com/2110363-HW-SYN-LAB/lab/tree/main/Lab2 or MCV and read Lab 2-3 Guide document.
2. Complete the Debouncer Module to work as described above.

Hint: You may want to read how to write a parameterized module on HW Lab Brief page 14.

Insert your modified Debouncer Module here.

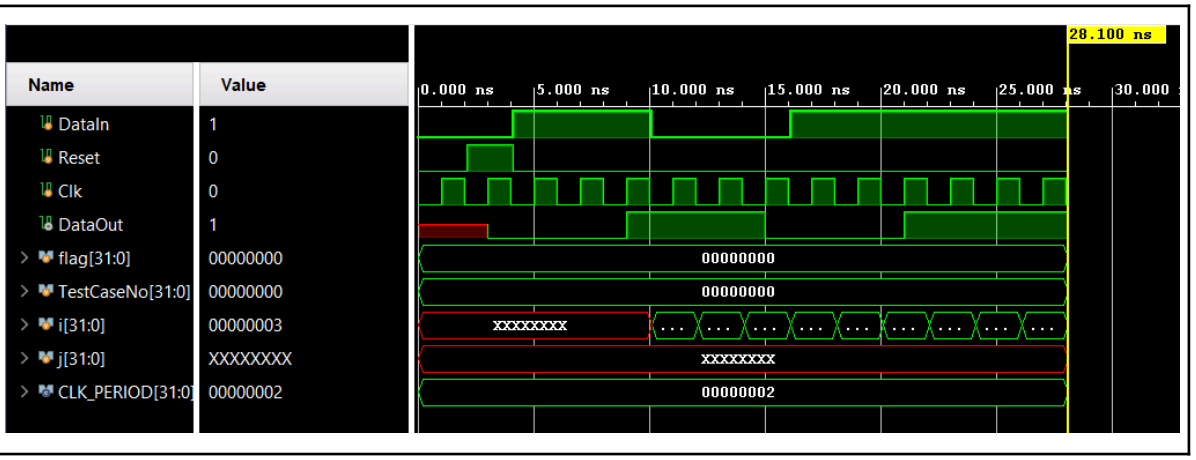
```
module Debouncer #(
    parameter CounterWidth = 1,
    parameter DebounceTime = 1
) (
    input  wire DataIn,
    input  wire Clk,
    input  wire Reset,
    output wire DataOut
);
    reg [CounterWidth-1:0] Counter;

    // Add your code here
    reg rDataOut;
    assign DataOut = rDataOut;

    always @(posedge Clk) begin
        if (Reset) begin
            rDataOut <= 0;
            Counter  <= 0;
        end
        else begin
            if(Counter == DebounceTime - 1) begin
                Counter <= 0;
                rDataOut <= DataIn;
            end
        else begin
            Counter <= Counter + 1;
        end
        end
    end

// End of your code
endmodule
```

3. Run the Testbench to verify your Debouncer Module (either Xilinx or CocoTB).

Insert your Testbench result here.

```
source DebouncerTB.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found.
#   }
# }
# run 1000ns
All test cases pass
```

Insert your Testbench waveform here.



Part 1.2 : Single Pulser

In this section, you will implement the SinglePulser Module, which generates a single pulse of a specified width in response to an input signal, ensuring one output pulse per input transition regardless of the input's duration. The module includes the following ports:

- **Inputs:**
  - Reset: Resets the module and initializes the output to 0.
  - Clk: Clock signal for synchronization.
  - DataIn: Input signal.
- **Output:**
  - DataOut: Output signal.

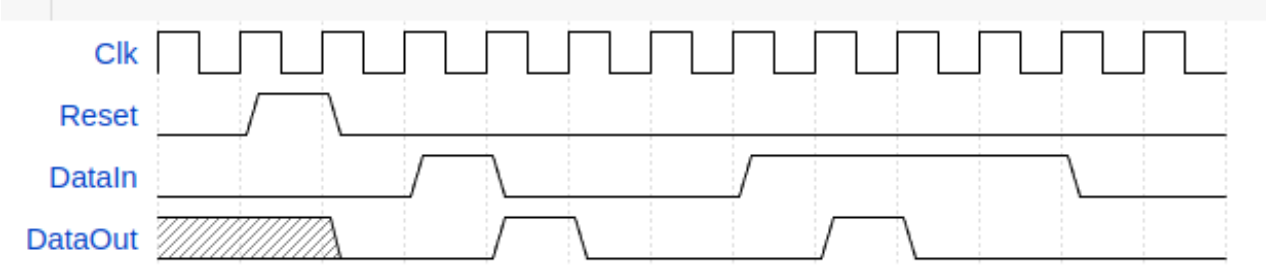
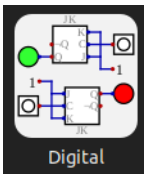


Figure 5: Single Pulser Timing Diagram

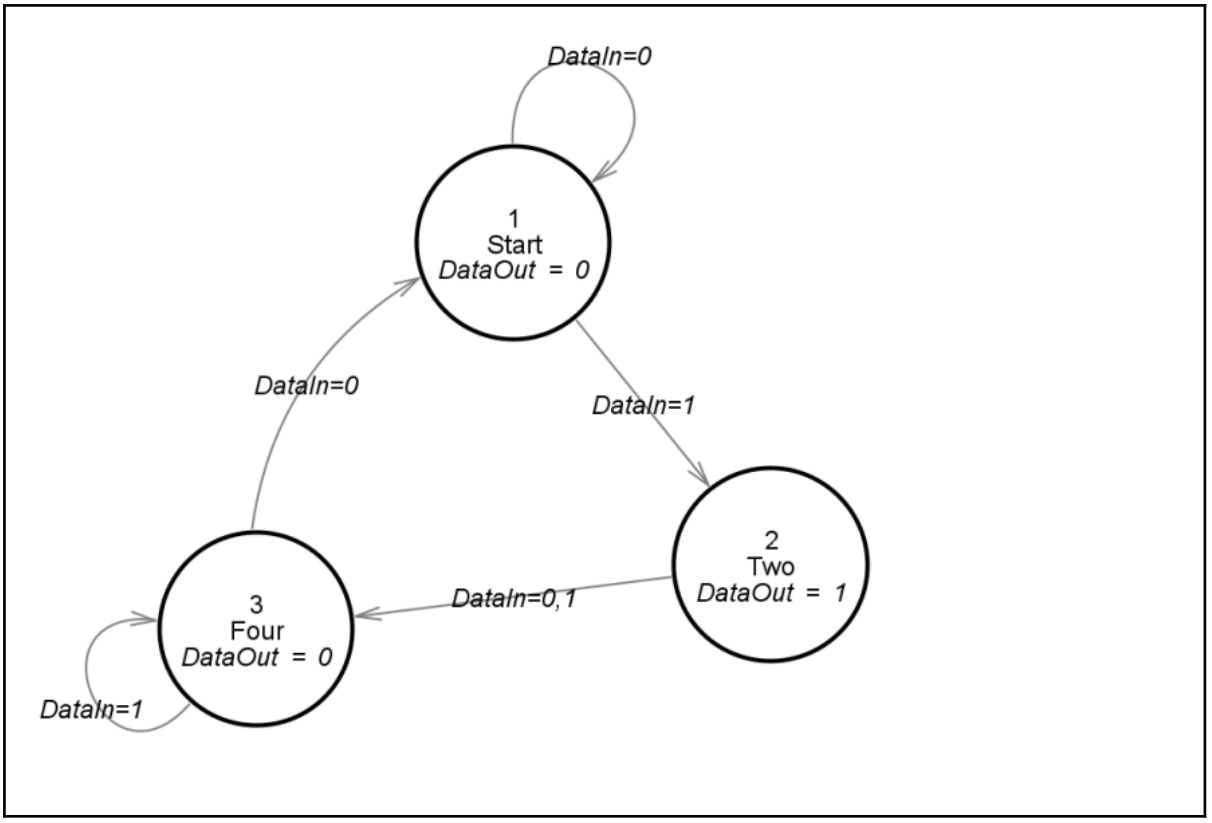
Figure 5 illustrates the behavior of the SinglePulser module. When Reset is asserted, the module resets its internal state, and DataOut is set to 0 in the next clock cycle. If DataIn is asserted, the module generates a single pulse on DataOut, setting it to 1 for one clock cycle. Afterward, DataOut returns to 0 and remains at 0 until DataIn is deasserted.

Instruction

1. Draw a State Diagram for the SinglePulser module. Hint : you may use a "Digital" program to draw State Diagrams.



Insert your State Diagram here .



2. Complete the SinglePulser Module to work as the state machine you have created.

Insert your modified SinglePulser Module here.

```
module SinglePulser (
    input  wire DataIn,
    input  wire Clk,
    input  wire Reset,
    output wire DataOut
);
    // Add your code here
    localparam Start = 2'b00;
    localparam Two = 2'b01;
    localparam Three = 2'b10;
    reg [1:0] State = 2'b00;
    assign DataOut = (State == Two);
    always @(posedge Clk or posedge Reset) begin
        if (Reset) begin
            end
        else begin
            case (State)
                Start: begin
                    if (DataIn == 1) begin
                    State <= Two; // Change state to One
```

```
                    end
                end
                Two: begin
                    State <= Three; // Change state to Two
                end
                Three: begin
                    if(DataIn == 0) begin
                        State <= Start; // Change state to Start
                    end
                end
            endcase
        end
    end
// End of your code
endmodule
```

3. Create a TestBench (SinglePulserTB) for the SinglePulser Module. Your testbench should include all the possible state transitions (either Xilinx or CocoTB).

Hint#1: You may want to read how to write a testbench of sequential logic circuits in the Lab 2 Guide.

Hint#2: You can implement the testbench by referring to the timing diagram provided in Figure 5.

Insert your Testbench here.

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

  // instantiate the Multiplexer module
  SinglePulser SinglePulserInst (
      .DataIn(DataIn),
      .Clk(Clk),
      .Reset(Reset),
      .DataOut(DataOut)
  );
  // 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_DataOut;  // Expected output
    begin
      if (DataOut !== expected_DataOut) begin
          $error("ERROR: TestCaseNo %0d | Time = %0t | DataOut = %b
(Expected: %b)", TestCaseNo, $time, DataOut,
              expected_DataOut);
        flag = 1;
      end
    end
  endtask

  localparam CLK_PERIOD = 2;
  always #(CLK_PERIOD / 2.0) Clk = ~Clk;

  // test cases
  initial begin
    DataIn = 0;
    Reset = 0;
    Clk = 0;

    // Insert test cases here
    #(CLK_PERIOD + 0.1);
    Reset = 1;
    #(CLK_PERIOD + 0.1);
    Reset = 0;
    check_output(0, 0); // Initial must be 0

    DataIn = 1;
    check_output(1, 0); // Wait 1 clock
    #(CLK_PERIOD + 0.1);
    check_output(2, 1); // Change value for 1 clk
    #(CLK_PERIOD + 0.1);
    check_output(3, 0); // Not changing
    #(CLK_PERIOD + 0.1);
    check_output(4, 0); // Not changing
```

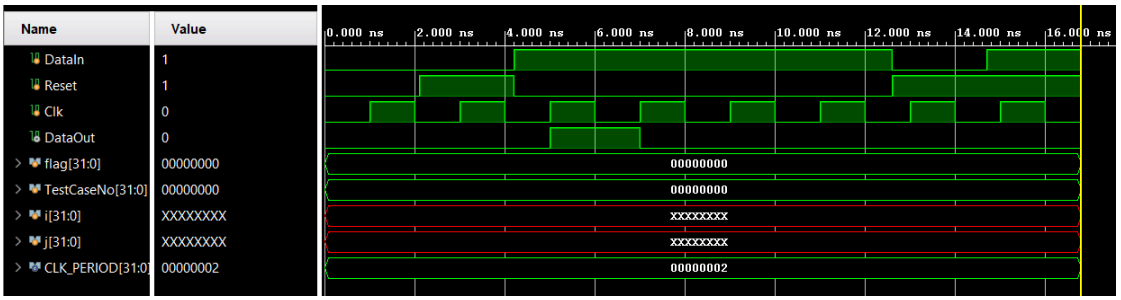
```
    #(CLK_PERIOD + 0.1);
    Reset = 1;
    DataIn = 0;
    #(CLK_PERIOD + 0.1);
    DataIn = 1;
    check_output(5, 0); // Wait 1 clk
    #(CLK_PERIOD + 0.1);
    check_output(5, 0); // Not changing (reset = 1)

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

4. Run the TestBench to verify your SinglePulser Module (either Xilinx or CocoTB).

Insert your Testbench result here.

```
source SinglePulserTB.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found.
#   }
# }
# run 1000ns
All test cases pass
```

Insert your Testbench waveform here.



Part 1.3 : System Integration and Validation

In this section, you will implement the InputSanitizer module as shown in Figure 3. Each input signal will first pass through the Debouncer module, followed by the SinglePulser module. The InputSanitizer module includes the following ports:

- **Inputs**:
  - Reset: Resets the module and initializes the output to 0.
  - Clk: Clock signal for synchronization.
  - DataIn [3:0]: Input signal that will be sanitized.
- **Output**:
  - DataOut [3:0]: Sanitized output signal.
- **Parameter**:
  - **DebounceTime**: Defines the sampling rate for the **DataIn** signal in the Debouncer module.
  - **CounterWidth**: Specifies the width of the counter register in the Debouncer module.

Instruction

1. Modify the InputSanitizer module to connect all the submodules according to figure 3. Hint : you can use generate.

Insert your modified InputSanitizer Module here.

```
module InputSanitizer #(
    parameter CounterWidth = 1,
    parameter DebounceTime = 1
) (
    input  wire [3:0] DataIn,
    input  wire       Clk,
    input  wire       Reset,
    output wire [3:0] DataOut
);
    // Add your code here
    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1) begin
            wire tmp;
            Debouncer #(
                .CounterWidth(CounterWidth),
                .DebounceTime(DebounceTime)
```

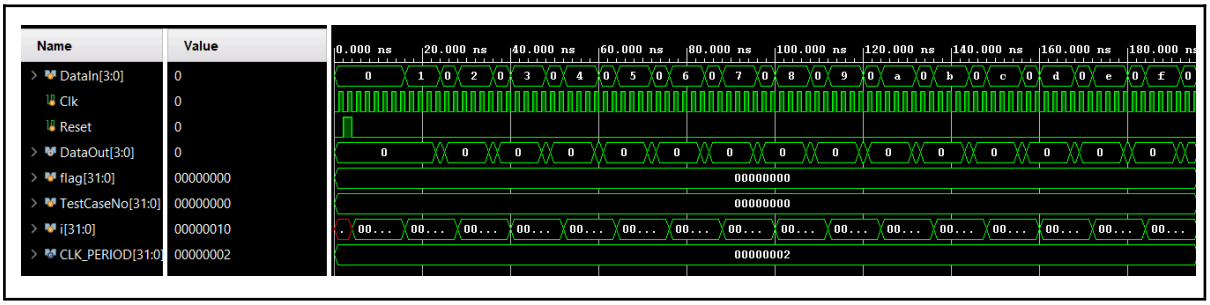
```
            ) DebouncerInst (
                .DataIn (DataIn[i]),
                .Clk    (Clk),
                .Reset  (Reset),
                .DataOut(tmp)
            );

            SinglePulser SinglePulserInst (
                .DataIn(tmp),
                .Clk(Clk),
                .Reset(Reset),
                .DataOut(DataOut[i])
            );
        end
    endgenerate
// End of your code
endmodule
```

2. Run the TestBench (InputSanitizerTB) to verify your InputSanitizer Module (either Xilinx or CocoTB).

Insert your Testbench result here.

```
source InputSanitizerTB.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found.
#   }
# }
# run 1000ns
All test cases pass
```

Insert your Testbench waveform here.



3. Call TA to inspect your work.

Part 2 : Four Digit BCD

In this part of the lab, you will complete the FourBCD Module. This module functions as a 4-digit BCD counter, constructed by combining four 1-digit BCD counters. Figure 6 illustrates the implementation of the FourBCD Module.

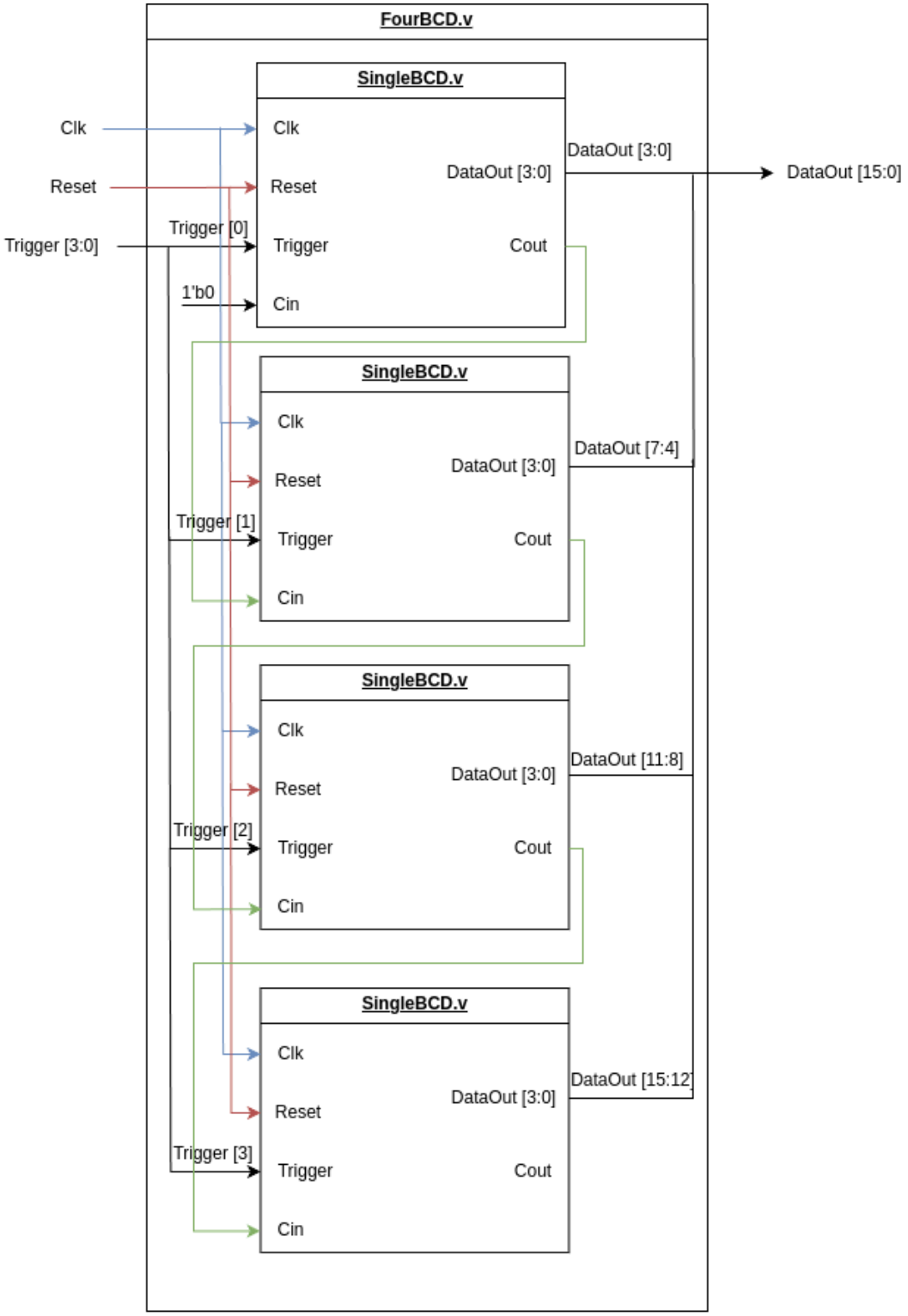


Figure 6 : FourBCD Module

This task is divided into two subparts: **Single BCD Counter**, and **System Integration and Validation**.

Part 2.1 : Single BCD Counter

In this section, you will implement the **SingleBCD Module**. This module acts as a single digit BCD counter that counts from 0 to 9. The module has the following ports:

- **Inputs**:
  - Reset: Resets the module and initializes the output to 0.
  - Clk: Clock signal for synchronization.
  - Trigger: Trigger signal that increases the counter value.
  - Cin: Carry in signal from other BCD counter Module.
- **Output**:
  - DataOut: The Value of this BCD counter.
  - Cout: The Carry out signal to other BCD counter Modules.

The SingleBCD module counts from 0 to 9. On the positive clock edge, the counter value increases in the next clock cycle based on the Trigger and Cin signals: by 1 if either Trigger or Cin is asserted, or by 2 if both are asserted. If the resulting counter value exceeds 9, it wraps around to value % 10 (modulo 10), and the Cout signal is asserted immediately to carry over to the next BCD counter digit in the subsequent clock cycle.

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.

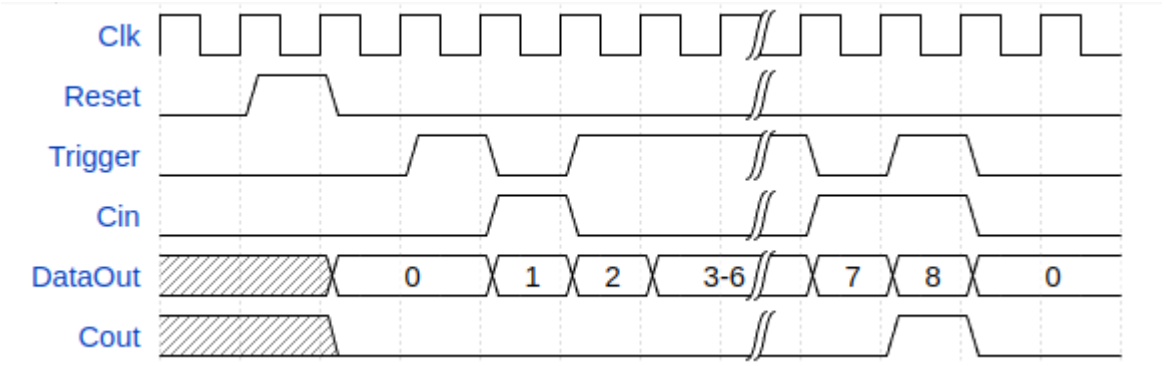


Figure 7 : SingleBCD Module Timing Diagram

Figure 7 illustrates the timing diagram of the SingleBCD Module.

Instruction

1. Complete the SingleBCD Module.

Insert your modified SingleBCD Module here.

```
module SingleBCD (
    input wire Trigger, // Trigger signal to increment counter
    input wire Clk, // Clock signal
    input wire Reset, // Reset signal
    input wire Cin, // Carry in from previous counter
    output wire [3:0] DataOut, // 4-bit BCD value
    output wire Cout // Carry out to next counter
);

    // Internal signal to calculate next value
    reg [3:0] Counter = 4'b0000;
    assign DataOut = Counter;
    assign Cout = (Counter + Trigger + Cin) >= 10;
    always @(posedge Clk) begin
        if (Reset) begin
            // Reset the counter to 0
            Counter <= 4'b0000;
        end
        else if (Cin | Trigger) begin
            Counter <= (Counter + Cin + Trigger) % 10;
        end
    end

endmodule
```

2. Create a TestBench (SingleBCDTB) for the SingleBCD Module. Your Testbench should include all the possible state transitions (either Xilinx or CocoTB).

Hint : You can implement the testbench by referring to the timing diagram provided in Figure 5.

Insert your Testbench here.

```
module SingleBCDTB ();
  // reg/wire declaration
  reg         Trigger;
  reg         Reset;
  reg         Clk;
```

```
  reg         Cin;
  wire [3:0] DataOut;
  wire        Cout;

  // instantiate the Multiplexer module
  SingleBCD SingleBCDInst (
      .Trigger(Trigger),
      .Clk(Clk),
      .Reset(Reset),
      .Cin(Cin),
      .DataOut(DataOut),
      .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 [3:0] expected_DataOut;  // Expected output
    input reg expected_Cout;           // Expected output
    begin
        if (DataOut !== expected_DataOut || Cout !== expected_Cout)
begin
          $error("ERROR: TestCaseNo %0d | Time = %0t | DataOut = %b
(Expected: %b) | Cout = %b (Expected: %b)",
                 TestCaseNo, $time, DataOut, expected_DataOut, Cout,
expected_Cout);
        flag = 1;
      end
    end
  endtask

  localparam CLK_PERIOD = 2;
  always #(CLK_PERIOD / 2.0) Clk = ~Clk;

  // test cases
  initial begin
```

```
    // Initialize signals
    Reset = 0;
    Clk = 0;
    Trigger = 0;
    Cin = 0;

    // Reset the system
    Reset = 1;
    #(CLK_PERIOD);
    Reset = 0;

    // Test cases
    #(CLK_PERIOD);
    check_output(0, 0, 0);
    Trigger = 1;
    check_output(1, 0, 0);
    #(CLK_PERIOD);
    check_output(2, 1, 0);
    Trigger = 0;
    Cin = 1;
    check_output(3, 1, 0);
    #(CLK_PERIOD);
    check_output(4, 2, 0);
    Cin = 0;
    Trigger = 1;
    check_output(5, 2, 0);
    #(CLK_PERIOD);
    check_output(6, 3, 0);
    #(CLK_PERIOD);
    check_output(7, 4, 0);
    #(CLK_PERIOD);
    #(CLK_PERIOD);
    #(CLK_PERIOD);
    check_output(8, 7, 0);
    Trigger = 0;
    Cin = 1;
    check_output(9, 7, 0);
    #(CLK_PERIOD);

    // Check the flag for test case results
    if (flag == 0) begin
      $display("All test cases pass");
```

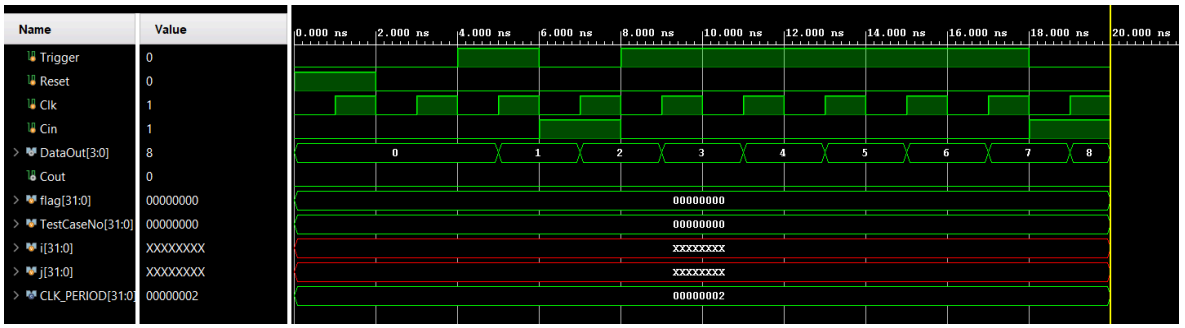
```
    end else begin
      $display("Some test cases fail");
    end
    $finish;
  end
endmodule
```

3. Run the TestBench to verify your SingleBCD Module (either Xilinx or CocoTB).

Insert your Testbench result here.

```
source SingleBCDTB.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found.
#   }
# }
# run 1000ns
All test cases pass
```

Insert your Testbench waveform here.



Part 2.2 : System Integration and Validation

In this section, you will implement the FourBCD Module, as shown in Figure 6. This module combines four SingleBCD Modules, wired together to create a 4-digit BCD counter. The FourBCD Module includes the following ports:

- **Inputs**:
  - Reset: Resets the module and initializes the output to 0.
  - Clk: Clock signal for synchronization.
  - Trigger [3:0]: Input signal to increase counter value in each digit.
- **Output**:
  - DataOut [15:0]: Counter value for each digit. (DataOut[i*4+3:i*4] for i is the digit i)

Instruction

1. Modify the FourBCD module to connect all the submodules according to figure 6.

Insert your modified FourBCD Module here.

```
module FourBCD (
    input  wire [ 3:0] Trigger,
    input  wire        Clk,
    input  wire        Reset,
    output wire [15:0] DataOut
);
    // Add your code here
    wire [4:0] carry;
    assign carry[0] = 0;
    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1) begin
            SingleBCD SingleBCDInst (
                .Trigger(Trigger[i]),
                .Clk    (Clk),
                .Reset  (Reset),
                .Cin    (carry[i]),
                .DataOut(DataOut[i*4+3:i*4]),
                .Cout   (carry[i+1])
            );
        end
    endgenerate
```

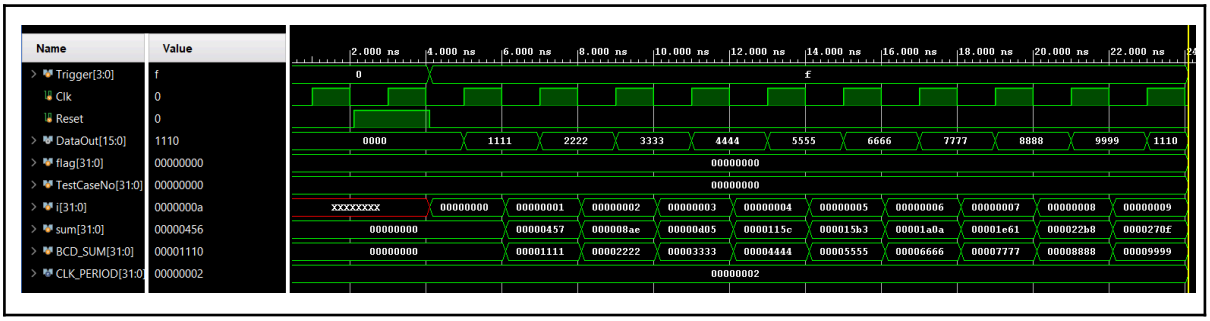
```
// End of your code
endmodule
```

2. Run the TestBench (FourBCDTB) to verify your FourBCD Module (either Xilinx or CocoTB).

Insert your Testbench result here.

```
source FourBCDTB.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found.
#   }
# }
# run 1000ns
All test cases pass
```

Insert your Testbench waveform here.



3. Call TA to inspect your work.

The last two part will be on Lab 3 (next week)

Part 3 : 7-Segments Display

In this part of the lab, you will complete the SevenSegmentDisplay Module. The purpose of this module is to display the value from the BCD counter on the 7-segment display.

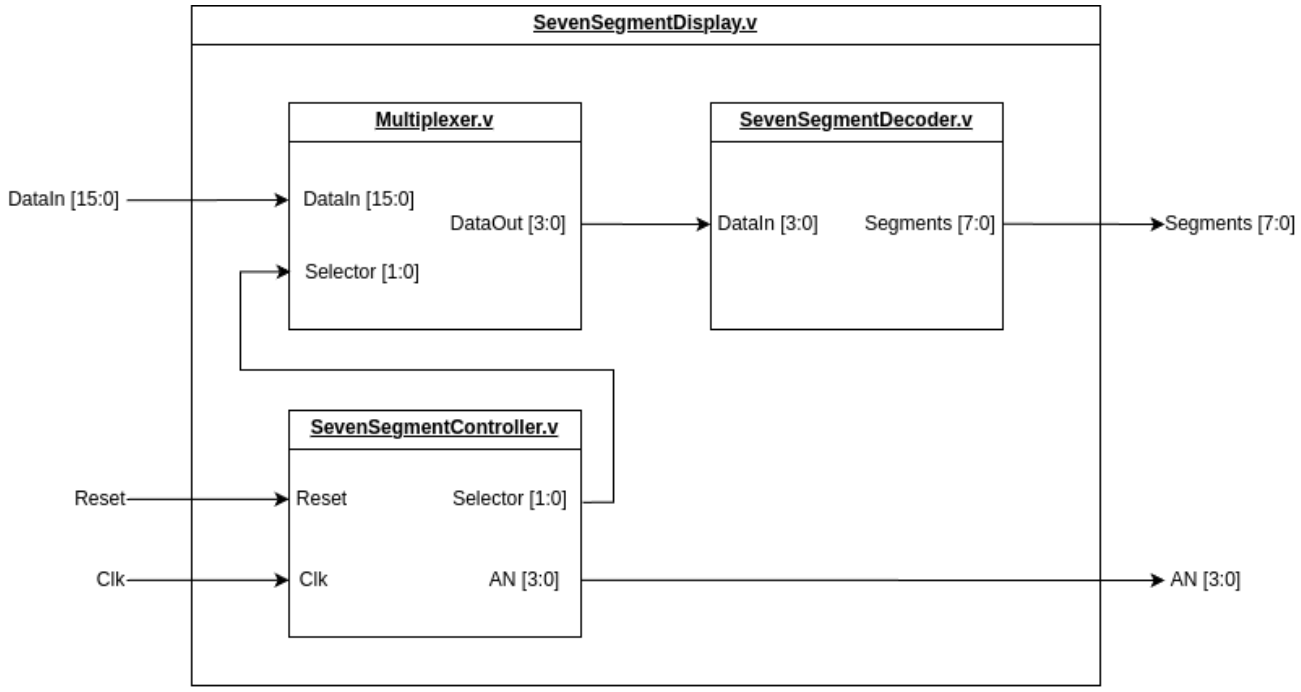


Figure 8 : SevenSegmentDisplay Module

Figure 8 illustrates the implementation of the FourBCD Module. The SevenSegmentDecoder, which you implemented in the first lab, is already provided for you. This task is divided into three subparts: Multiplexer, SevenSegmentController, and System Integration and Validation.

Part 3.1 : Multiplexer

In this section, you will implement the Multiplexer Module, which functions as a 4:1 multiplexer. The module has the following ports

- **Inputs**:
  - Selector [1:0]: Selects which data to output.
  - DataIn [15:0]: Input data consisting of four 4-bit values.
- **Output**:
  - DataOut [3:0]: Output Data.

Table 1 represents the truth table for the 4:1 Multiplexer. You are required to implement this module using combinational logic.

Selector [1:0]	DataOut [3:0]
0	DataIn [3:0]
1	DataIn [7:4]
2	DataIn [11:8]
3	DataIn [15:12]

Table 1: 4:1 Multiplexer Truth Table

Instruction

1. Complete the Multiplexer Module.

Insert your modified Multiplexer Module here.

Part 3.2 : SevenSegmentController

In this section, you will implement the SevenSegmentController Module, which manages the data to be displayed on the 7-segment display. The module includes the following ports:

- **Inputs**:
  - Reset: Resets the module and initializes the output to 0.
  - Clk: Clock signal for synchronization.
- **Output**:
  - AN [3:0]: 7-Segments Anode Controller.
  - Selector [1:0]: Selector signal.
- **Parameter**:
  - **ControllerClockCycle**: Specifies the number of clock cycles allocated for displaying each digit in the SevenSegmentController module.
  - **ControllerCounterWidth**: Specifies the width of the counter register in the SevenSegmentController module.

The SevenSegmentController module manages the timing for displaying each digit on the 7-segment display. According to the Basys3 Reference Manual, each digit should be refreshed once every 1 to 16 milliseconds, corresponding to a refresh rate of 60 Hz to 1 kHz. The AN signal controls which digit is lit, while the Selector signal determines the appropriate data to be displayed.

When the Reset signal is asserted, the module resets its internal state, setting the AN[3:0] value to 4'b1111 and the Selector[1:0] value to 0 in the next clock cycle.

Figure 9 illustrates the timing diagram of the SevenSegmentController module.

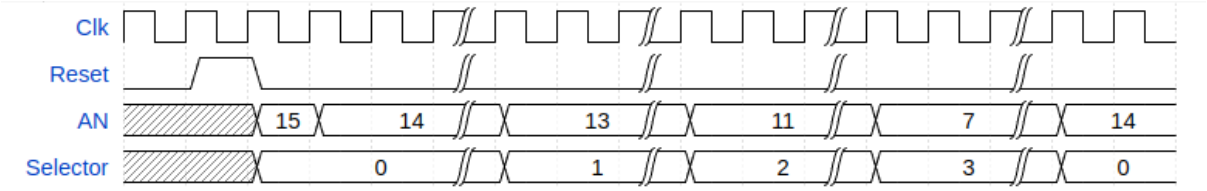


Figure 9 : SevenSegmentController Module Timing Diagram

Instruction

1. Complete the SevenSegmentController Module.

Insert your modified SevenSegmentController Module here.

2. Create a TestBench (SevenSegmentControllerTB) for the SevenSegmentController Module. Your Testbench should include all the possible state transitions (either Xilinx or CocoTB).

Hint: You can implement the testbench by referring to the timing diagram provided in Figure 5.

Insert your Testbench here.

3. Run the TestBench to verify your SevenSegmentController Module (either Xilinx or CocoTB).

Insert your Testbench result here.

Insert your Testbench waveform here.

Part 3.3 : System Integration and Validation

In this section, you will implement the SevenSegmentDisplay Module, as shown in Figure 8. The SevenSegmentDisplay Module includes the following ports:

- **Inputs**:
  - Reset: Resets the module and initializes the output to 0.
  - Clk: Clock signal for synchronization.
  - DataIn [15:0]: Input data from FourBCD Module
- **Output**:
  - Segments [7:0]: Signal that is used to control cathode of the 7-Segments Display.
  - AN [3:0]: Signal that is used to control anode of the 7-Segments Display.
- **Parameter**:
  - **ControllerClockCycle**: Specifies the number of clock cycles allocated for displaying each digit in the SevenSegmentController module.
  - **ControllerCounterWidth**: Specifies the width of the counter register in the SevenSegmentController module.

Instruction

1. Modify the SevenSegmentDisplay module to connect all the submodules according to figure 8.

Insert your modified SevenSegmentDisplay Module here.

2. Run the TestBench (SevenSegmentDisplayTB) to verify your SevenSegmentDisplay Module (either Xilinx or CocoTB).

Insert your Testbench result here.

Insert your Testbench waveform here.

3. Call TA to inspect your work.

Part 4 : Top Module and Hardware Programming

In this section, you will complete the **BCDCounter Module**, create a constraint file for the **BCDCounter System**, and generate a bitstream to program your Basys3 board.

This task is divided into two subparts: BCDCounter, and Hardware Programming.

Part 4.1 : BCDCounter

In this section, you will implement the **BCDCounter Module**. This module is the top module for this project's system. The module has the following ports:

- **Inputs**:
  - Reset: Resets the module and initializes the output to 0.
  - Clk: Clock signal for synchronization.
  - Trigger [3:0]: Trigger signal that increases the counter value.
- **Output**:
  - Segments [7:0]: Signal that is used to control cathode of the 7-Segments Display.
  - AN [3:0]: Signal that is used to control anode of the 7-Segments Display.
- **Parameter**:
  - **DebounceTime**: Defines the sampling rate for the **DataIn** signal in the Debouncer module.
  - **CounterWidth**: Specifies the width of the counter register in the Debouncer module.
  - **ControllerClockCycle**: specifies the number of clock cycles allocated for displaying each digit in the SevenSegmentController module.
  - **ControllerCounterWidth**: Specifies the width of the counter register in the SevenSegmentController module.

Instruction

1. Modify the BCDCounter module to connect all the submodules according to figure 2. Also you need to set the appropriate Parameter value.

Insert your modified BCDCounter Module here.

Part 4.2 : Hardware Programming

In this section, you will create a constraint file, synthesis, implementation, generate bitstream, and then upload it to the Basys 3 board. The port mappings from BCDCounter Module to Basys3 board are as follows:

BCDCounter ports	Basys3 board
Reset	BTNC
Clk	Basys3 100Mhz Clk
Trigger [3:0]	{BTND,BTNU,BTNR,BTNL}
AN [3:0]	{AN3, AN2, AN1, AN0}
Segments [7:0]	{CA, CB, CC, CD, CE, CF, CG, DP}

Table 2 : System Port Mapping

Instruction

1. Complete this mapping table below to map your system's inputs and outputs onto the **Basys3** pins.

Hint: Look up the Basys3 reference manual between page 6 and 15.

System.v input/output	Basys3 Pins
Reset	
Clk	
Trigger[0]	
Trigger[1]	
Trigger[2]	
Trigger[3]	
AN[0]	
AN[1]	
AN[2]	
AN[3]	

Segments[0]	
Segments[1]	
Segments[2]	
Segments[3]	
Segments[4]	
Segments[5]	
Segments[6]	
Segments[7]	

2. Create the constraint file.

Submit your constraint file here.

3. Run the synthesis.
4. Run the implementation, generate the bitstream, and program the **Basys3** device.

Take a picture of your Board working after programming.

5. Call TA to inspect your work.
6. Submit this sheet to the MCV.