Assignment 1

Imagine you are an electronic engineer working at ToyStory (Pvt) Ltd. The senior design engineers are developing a mini calculator for children under the age of 5 years. The calculator should be simple and only perform the multiplication of two numbers (each below 10) and display the result in decimal form.

Instead of using a microcontroller, the senior engineers have decided to design a custom integrated circuit (IC) for the calculator. To confirm the functionality before fabricating the IC, they require a preliminary FPGA implementation. Your task is to design and simulate the FPGA module for this mini calculator.

This assignment is divided into four sections, each containing specific exercises to complete.

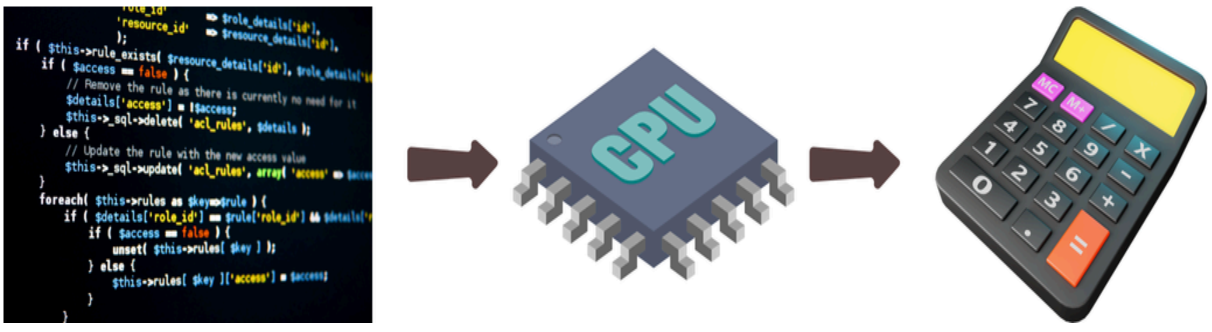
You only need to complete the first three sections to get full marks.
Section 4 is optional.

Submission Guidelines

- Submit a zip file containing all .sv files, testbenches and screenshots
- Ensure proper file organization:

Throughout design only consider numbers less than 10 will be inputs to the module

---



```
                'resource_id'       => $role_details['id'],
                                    => $resource_details['id'],
                );
if ( $this->rule_exists( $resource_details['id'], $role_details['i
    if ( $access == false ) {
        // Remove the rule as there is currently no need for it
        $details['access'] = !$access;
        $this->_sql->delete( 'acl_rules', $details );
    } else {
        // Update the rule with the new access value
        $this->_sql->update( 'acl_rules', array( 'access' => $acce
    }
    foreach( $this->rules as $key=>$rule ) {
        if ( $details['role_id'] == $rule['role_id'] && $details['r
            if ( $access == false ) {
                unset( $this->rules[ $key ] );
            } else {
                $this->rules[ $key ]['access'] = $access;
            }
        }
```

Section 1: 8-Bit Adder Design

Complete the section by using slides

1. Implement 1-Bit Full Adder in System Verilog

- Write an SV module for the 1-bit full adder using the data flow modeling approach.
- Use simple logic expressions (Example: *assign* out = A ^ B).
- The module should have 3 inputs (A, B, Cin) and 2 outputs (Sum, Cout) as figure 01.
- Save the file using the module name (e.g. one_bit_adder.sv).
- Save the file and module in the same name.

2. Testbench for 1-Bit Adder

- Write a testbench to verify the functionality of your one-bit full adder module.
- The testbench should check all possible input combinations.
- Use *`include "one_bit_adder.sv"* to reference the one_bit_adder module in the beginning .
- Simulate the design and capture a screenshot of the waveform.
- Save the testbench file in the same directory as one_bit_adder_tb.sv.

3. Implementing a 8-Bit unsigned Adder

- Using the 1-bit full adder, design a 8-bit ripple carry adder.
- Instantiate eight one_bit_adder modules within a new 8-bit adder module.
- The module should include (figure 02):
  - Inputs: Two 8-bit numbers (A[7:0], B[7:0]) and a Carry_in (Cin).
  - Outputs: One 8-bit sum (Sum[7:0]) and Carry_out (Cout).
- Save the module file in the same directory as eight_bit_adder.sv.

4. Testbench for 8-Bit unsigned Adder

- Implement a testbench to validate the functionality of the 8-bit adder.
- The testbench should use 10 different test cases, ensuring coverage of cases where carry-out occurs.
- Capture a screenshot of the output waveform.
- Save the testbench file in the same directory as eight_bit_adder_tb.sv.

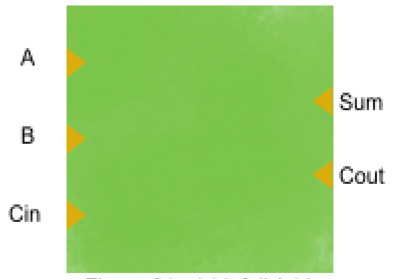


Figure 01 : 1-bit full Adder

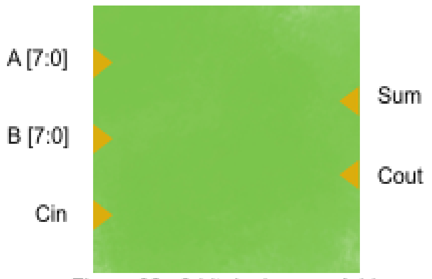


Figure 02 : 8-bit ripple carry Adder

Section 2: 4-Bit Multiplier Design

1. Understanding Multiplication Mechanism

- Perform **multiplication** of two **4-bit binary numbers** on a paper and analyze the pattern.
- Identify how shifting and addition can be used to implement multiplication.

2. Implementing a 4-bit unsigned Multiplier

- Design a **4-bit multiplier** by chaining **four 8-bit adders** (from section 1) together.
  - Hint: Change 4-bit input number into 8-bit number by zero padding where necessary. Use logical AND operation
- The module should include (Figure 03):
  - **Inputs**: Two **4-bit numbers** (A[3:0], B[3:0]).
  - **Output**: One **8-bit product** (product[7:0]).
- Save the module file in the same directory as four_bit_unsinged_multiplier.sv.

3. Testbench for 4-Bit Multiplier

- Implement a **testbench** to validate the functionality of the 4-bit multiplier.
- The testbench should use **10 different test cases**.
- Capture a **screenshot of the output waveform**.
- Save the testbench file in the same directory as four_bit_multiplier_tb.sv.

4. Implementing a 4-bit Signed Multiplier

- Change the design from (2) to perform signed multiplication
- Save the module file in the same directory as four_bit_signed_multiplier.sv.

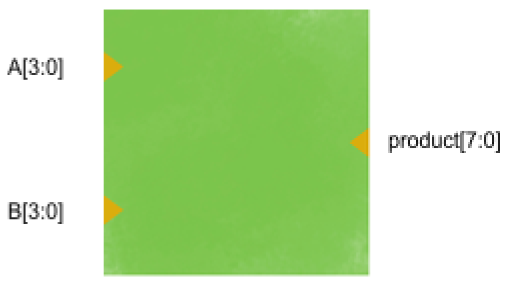


Figure 03 : 4-bit unsigned multiplier

Section 3: 7-Segment Display Decoder

Fill the missing parts of the code in Annex 01

The calculator will display results on two seven-segment displays. A decoder module is needed to convert a 4-bit binary input into 7-bit segment outputs for the display (Figure 5).

1. 7-Segment Display Decoder Design

- Design a System Verilog module that (figure 04):
  - Accepts a 4-bit input (representing numbers 0-9).
  - Outputs a 7-bit signal corresponding to the segment activation of a common cathode 7-segment display.
- The module should correctly map each decimal digit (0-9) to its corresponding 7-segment encoding.
- Save the module file in the same directory as seven_segment_decoder.sv.

2. Testbench for 7-Segment Decoder

- Write a testbench to verify the functionality of your 7-segment decoder.
- The testbench should check all numbers from 0 to 9.
- Capture a screenshot of the output waveform.
- Save the testbench file in the same directory as seven_segment_decoder_tb.sv.

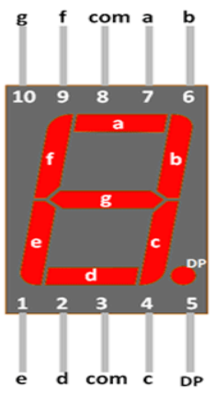


Figure 05 : Pinout of seven segment display

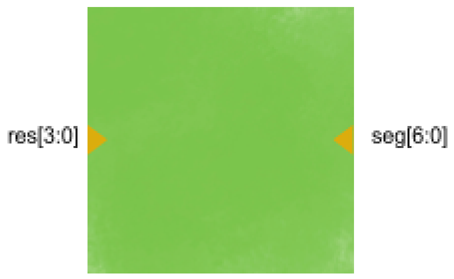


Figure 04 : seven segment decoder

Section 4: Optional - Integrating the Modules

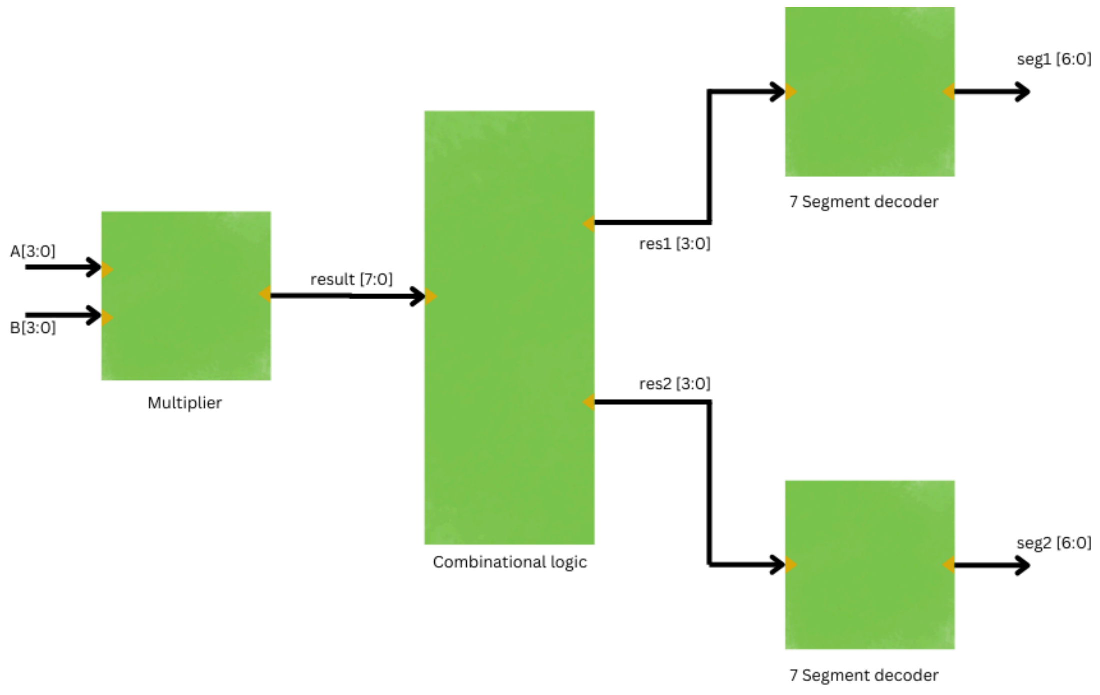
As an additional task, integrate the 4-bit adder and 7-segment decoder into a single top module.

1. Integration of multiplier and Display Decoder

- Create a top module that combines the 4_bit_adder and two 7_seg_decoder modules.
- The module should:
  - Accept two 4-bit numbers as inputs (A[3:0], B[3:0]) along with a Carry_in.
  - Compute the 4-bit sum and Carry_out.
  - Convert the sum into two 7-segment display outputs (combinational logic).
    - Hint: - You can use modulus division (%) to separate values.
- Save the module file in the same directory as top_module.sv.

2. Testbench for Integrated Design

- Write a testbench to validate the overall design.
- Use 10 test cases to confirm correct addition and display conversion.
- Capture a screenshot of the output waveform.
- Save the testbench file in the same directory as top_module.sv.

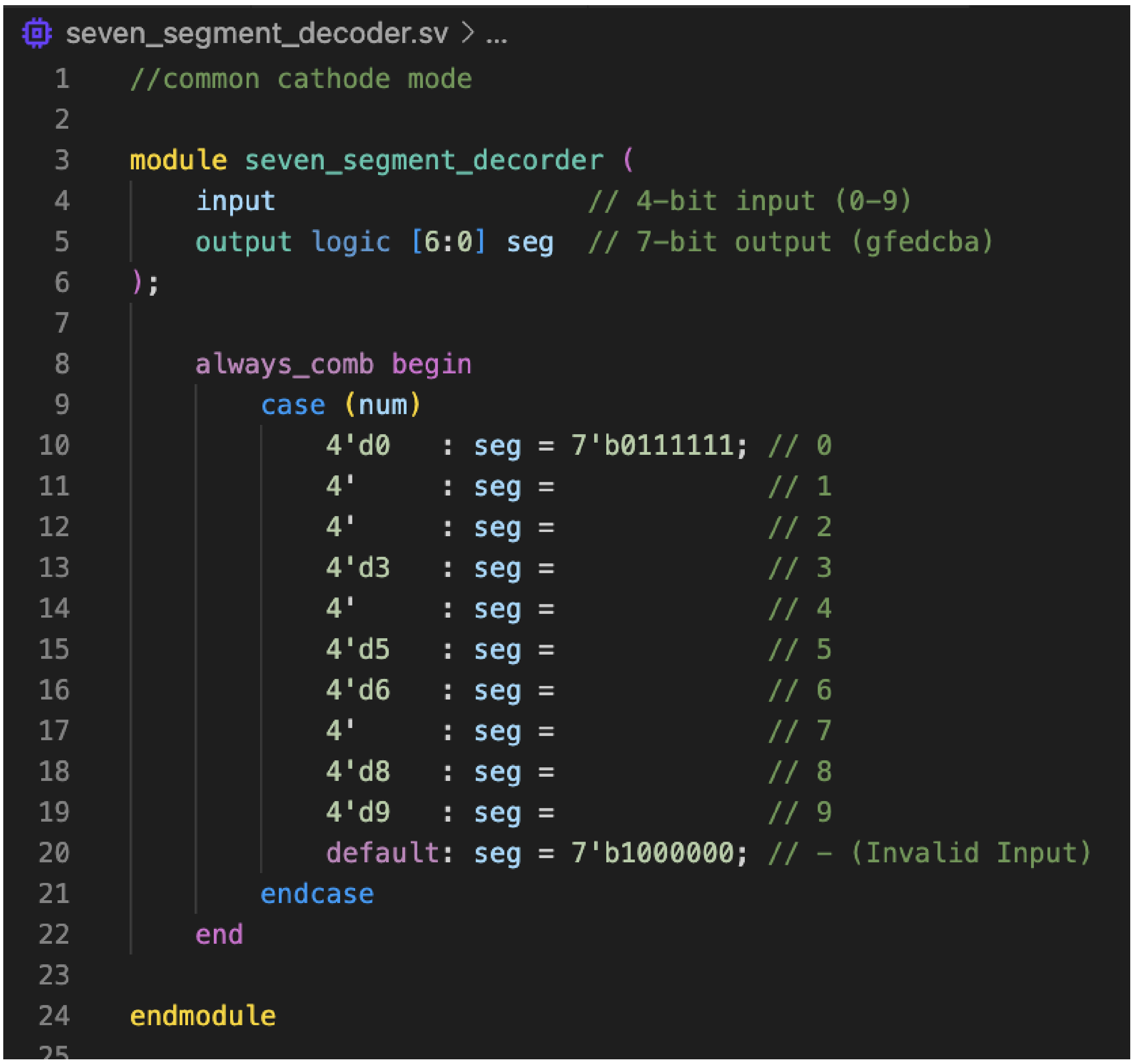


3. Truth Table and Logic Simplification (For basic theoretical knowledge)

- Write down the truth table for a 1-bit full adder with three inputs:
    - Input_A (single bit)
    - Input_B (single bit)
    - Carry_in (single bit)
- Two outputs: Sum and Carry_out.
- Using the truth table, simplify the logic equations using Boolean algebra.
- Draw a simple logic gate implementation based on the simplified equations.
- Reference materials:
    - YouTube Tutorial
    - GeeksforGeeks Full Adder Guide

Annex 01

Partially completed seven segment decoder module



```
//common cathode mode

module seven_segment_decorder (
    input                   // 4-bit input (0-9)
    output logic [6:0] seg  // 7-bit output (gfedcba)
);

    always_comb begin
        case (num)
            4'd0   : seg = 7'b0111111; // 0
            4'     : seg =             // 1
            4'     : seg =             // 2
            4'd3   : seg =             // 3
            4'     : seg =             // 4
            4'd5   : seg =             // 5
            4'd6   : seg =             // 6
            4'     : seg =             // 7
            4'd8   : seg =             // 8
            4'd9   : seg =             // 9
            default: seg = 7'b1000000; // - (Invalid Input)
        endcase
    end

endmodule
```