CS666 Assignment 1

Vissa Surya Sai Sandeep
22111084
vsssandeep22@iitk.ac.in

Abhinandan Singh Baghel
231110002
asbaghel23@iitk.ac.in

Himanshu Karnatak
231110017
himanshuk23@iitk.ac.in

HRUSHIKESH NAKKA
231110018
hnakka23@iitk.ac.in

Q 1. Design a Verilog module for the 1-bit full adder that should have three inputs: A, B, and Cin (carry-in), and two outputs: Sum and Cout (carry-out). Now, using instanciation of the 1-bit full adder module, design a 8-bit full adder that should have three inputs: A (8-bit input), B (8-bit input),
and Cin (1-bit carry-in), and two outputs: Sum (8-bit output) and Cout (1-bit carry-out).

Ans.

Behavioral Design Approach:

1. **Problem Understanding**: Understand the problem statement and the requirements of the 8-bit full adder.
2. **Conceptualize the Logic**: Visualize how the 8-bit full adder works conceptually, focusing on the inputs, outputs, and the logic required for addition.
3. **High-Level Abstraction**: Create a behavioral Verilog module that describes the desired functionality without going into implementation details.
4. **Use Procedural Constructs**: Utilize procedural constructs like `always` blocks to define how inputs are processed over time.
5. **Test with Testbenches**: Write testbenches to validate the correctness of your behavioral design through simulation.

Structural Design Approach:

1. **Problem Understanding**: Ensure you thoroughly understand the problem requirements and the interactions between the components.

2. **Component Breakdown**: Break down the 8-bit full adder into smaller components, considering both inputs and outputs for each component.
3. **Hierarchical Design**: Create separate Verilog modules for each component, describing their inputs, outputs, and internal logic.
4. **Module Instantiation**: In the main module, instantiate the lower-level modules and connect their inputs and outputs.
5. **Interconnections**: Define how the components are interconnected to achieve the desired functionality.
6. **Reuse of Components**: Take advantage of modularity to reuse components you've designed or may design in the future.
7. **Test and Verification**: Develop testbenches to validate the structural design, ensuring that the components work as expected.

Continuous Assignment Approach:

1. **Problem Understanding**: Gain a clear understanding of the problem and the logic involved in the 8-bit full adder.
2. **Combinational Logic**: Recognize that the full adder's logic is combinational, where outputs are solely determined by inputs.
3. **Continuous Assignment**: Use continuous assignment statements (`assign`) to describe the logic for each output signal.
4. **Equations for Logic**: Write equations that describe the logic required for the sum bits and the carry-out.
5. **Propagate Signals**: Define how input signals propagate through the logic equations to produce the required outputs.
6. **Validation and Simulation**: Create a testbench to simulate the behavior of the continuous assignments, ensuring correctness.

By approaching the problem step by step using these three design paradigms, you'll be able to effectively design, implement, and verify the 8-bit full adder. Keep in mind that each approach has its strengths and weaknesses, and the choice depends on the complexity of the design, modularity requirements, and the design team's preferences.

Q 2. Write a Verilog module for 4-bit multiplication? The module should have two 4-bit inputs, A and B,
and produce an 8-bit product, P. The computational workflow is shown in Figure 1. Firstly, generate
a partial product matrix with dimensions of 4x8. Then, compress the matrix to a 2x8 matrix using a
1-bit full adder module that was designed earlier. Finally, generate the final result using an 8-bit adder module that was also designed earlier.

Ans .

1. **Problem Understanding and Analysis**:
   - Understand the problem statement: Create a 4-bit multiplier using partial products, compression, and addition.
   - Identify key components: Inputs A and B, partial product generation, compression, and final addition.

2. **Partial Product Generation**:
   - Create a 4x8 matrix to hold partial products (16 bits in total).
   - Use nested loops to iterate through both input bits and generate products.
   - Store partial products in the matrix using AND operation.

3. **Compression using 1-bit Full Adder**:
   - Implement a 1-bit full adder module (if not already designed) for sum and carry calculation.
   - Compress the 4x8 matrix to a 2x8 matrix using the 1-bit full adder:
     - Iterate through each column, adding two rows at a time using the full adder.
     - Carry from the previous addition should be propagated to the next column's addition.

4. **Final Result using 8-bit Adder**:
   - Design an 8-bit adder module (if not already designed) for adding 8-bit numbers.
   - Extend the 2x8 compressed matrix to 8 bits by adding a leading zero.
   - Use the 8-bit adder to add the two compressed rows, including the carry bit.

5. **Testbench Development**:
   - Write a testbench to verify the correctness of the 4-bit multiplier module.
   - Create test cases with different inputs to cover various scenarios.
   - Simulate the design and verify the expected outputs against the simulated outputs.

6. **Simulation and Debugging**:
   - Run the simulation and observe waveforms to ensure correct behavior.
   - Debug any unexpected behavior, errors, or mismatches between expected and actual outputs.

7. **Synthesis**:
   - Use synthesis tools to convert the behavioral description to gate-level representation.
   - Optimize the design for area, power, and timing using synthesis options.

8. **Methodology Documentation**:
   - Create a methodology document that outlines your approach.
   - Explain the behavioral, structural, and continuous assignments design methods.
   - Include code snippets, explanations, and tips for each step of the design process.

9. **Simulation using VCD Files and Waveform Viewing**:
   - During simulation, use `$dumpvars` to specify signals for waveform dumping.
   - Generate a `.vcd` file containing waveform data.
   - Use waveform viewer tools like GTKWave to load and analyze the `.vcd` file.

10. **Iterate and Improve**:
    - Iterate through the design, simulation, and synthesis steps as needed to refine your module.
    - Verify the design with additional test cases to ensure robustness.

Q.3 Write a Verilog code for a 4-bit forward counting synchronous Johnson counter with a one-bit reset
signal. The Johnson counter is a type of shift register that cycles through a sequence of 2n states. In
this case, the 4-bit counter will produce 16 states, each represented by a unique 4-bit binary pattern.
Additionally, the counter should be able to reset to its initial state when a one-bit reset signal is asserted. The input consists of two signals 1. clock; 2. reset signal; and the output is a 4-bit value
representing the state of the counter at a particular state.

Ans.

Step 1: Understanding the Problem and Specification

1. **Problem Understanding:** Understand the requirements of the Johnson counter. It's a 4-bit forward counting synchronous counter with a reset capability.

2. **Functional Specification:** Define the expected behavior of the counter. It should cycle through a sequence of 16 states, incrementing one step at each clock cycle, and reset to its initial state upon receiving a reset signal.

Step 2: Algorithmic Design

1. **Counter Sequence:** Recognize that a Johnson counter sequence alternates between a pattern and its complement. Plan to achieve this by shifting the bits and inverting the most significant bit.

Step 3: Behavioral Description

1. **High-Level Description:** Use Verilog to describe the behavior of the Johnson counter.

2. **Clock and Reset Inputs:** Declare the clock and reset inputs.

3. **Counter Register:** Define a 4-bit register to store the current state of the counter.

4. **Reset Logic:** Use an `always` block triggered by the clock and reset signals. If the reset signal is high, set the counter to its initial state. Otherwise, perform the shift operation to create the Johnson counter sequence.

Step 4: Simulation and Verification

1. **Testbench Setup:** Create a testbench module to provide the clock and reset signals and instantiate the Johnson counter module.

2. **Clock Generation:** Use an `initial` block to generate a clock signal with the desired frequency.

3. **Reset Generation:** Create a sequence of reset pulses using the `reset` signal.

4. **Monitor State Changes:** Use an `always` block to display the counter state whenever there is a clock edge.

5. **Simulation Run:** Run the simulation to observe the counter's behavior and verify that it counts forward and resets as expected.

Step 5: Refinement and Debugging

1. **Simulation Analysis:** Carefully analyze the simulation output to ensure that the counter's behavior matches the expected sequence and reset behavior.

2. **Debugging:** If there are discrepancies, review the behavioral description, clock and reset signal timings, and the reset logic.

Step 6: Documentation

1. **Methodology Document:** Create a methodology document that outlines the design approach taken, explaining the behavioral methodology's benefits and the reasons for choosing it.

2. **Code Comments:** Add comments to the Verilog code to explain the different sections, the algorithm used, and any design decisions made.

By following these steps, you can design, simulate, and verify the 4-bit forward counting synchronous Johnson counter with reset functionality using the behavioral approach. This approach focuses on describing the counter's behavior without diving into the low-level structural details.