**Final Report: Combinational 8-bit Checksum Generator**

**Project – Digital Logic Design**

**Objective**

To design and implement a **combinational circuit** that calculates the **one’s complement checksum** of two 8-bit binary inputs. The design must handle **carry-out wraparound** and use only **basic logic gates**. The checksum generator will act as a simplified version of checksum logic used in networking protocols like UDP and TCP.

**Understanding the Problem**

In one's complement addition:

* Two binary numbers are added.
* If the result overflows (i.e., there’s a carry-out from the most significant bit), the carry-out is wrapped around and added back to the least significant bit.
* The result is then inverted bit-by-bit to get the final checksum.

This technique helps in detecting transmission errors. At the receiver end, if all the data and the checksum sum up to a word of all 1s (for 8-bit: 11111111), the data is assumed to be correct.

**Design Overview**

**Step 1: 8-bit Adder**

The first stage uses an 8-bit adder to add two 8-bit inputs, A and B. This gives:

* An 8-bit output sum.
* A 1-bit carry-out signal.

**Step 2: Carry Wraparound Logic**

If the carry-out is 1, we add 00000001 to the 8-bit sum.  
If the carry-out is 0, the sum remains unchanged.

To do this:

* A **MUX** is used to choose between two values:
  + 00000000 if carry = 0
  + 00000001 if carry = 1
* This output is added to the original sum using a second 8-bit adder.

This ensures the circuit performs proper carry wraparound logic as per one's complement arithmetic.

**Step 3: One’s Complement**

The final sum (after carry wraparound) is passed through **NOT gates** to invert each bit. The result is the **final checksum**.

**Simulation Results**

Multiple test cases were used to verify correctness. In each case, when the inputs and their computed checksum were summed together, the result was 11111111, which confirms correct one's complement behavior.

**Sample Test Cases**

| **Input A (Hex)** | **Input B (Hex)** | **Raw Sum** | **Carry** | **Final Sum (Wraparound)** | **Checksum** |
| --- | --- | --- | --- | --- | --- |
| 0xAB | 0xCD | 0x178 | 1 | 0x79 | 0x86 |
| 0xFF | 0x01 | 0x100 | 1 | 0x01 | 0xFE |
| 0x00 | 0x00 | 0x00 | 0 | 0x00 | 0xFF |
| 0xFF | 0xFF | 0x1FE | 1 | 0xFE + 1 = 0xFF | 0x00 |

The circuit behaved correctly across all tested combinations.

**Design Choices**

* **MUX-based carry logic**: Instead of using splitters, the design uses a MUX and constant inputs to choose between adding 00000000 or 00000001, depending on the carry-out.
* **Two-stage adder**: One adder performs the main sum, and the second handles the optional increment based on carry.
* **Inversion for checksum**: NOT gates are used individually for each bit to get the one's complement result.

This structure keeps the design simple, readable, and purely combinational.

**Challenges Faced**

* Initially, the circuit gave incorrect results when carry-out was 0, due to incorrect conditional addition. This was resolved using MUX logic.
* Ensuring all inputs and outputs were exactly 8-bit wide was important to avoid bus-width mismatch errors.
* Understanding the difference between binary addition and one's complement checksum logic required a few test runs and corrections.

**Optimizations and Potential Improvements**

**Implemented:**

* Efficient carry wraparound logic without needing splitters.
* Proper use of MUX to minimize components.

**Possible Enhancements:**

* Extend the design to add more than two words (e.g., full 16-bit checksum logic).
* Add error simulation (e.g., corrupting one bit and seeing if the checksum fails).
* Implement a checksum verification circuit to mimic a receiver.

**Conclusion**

This project successfully demonstrated the design and implementation of a combinational checksum circuit using basic logic gates. It involved applying theoretical concepts like binary addition, carry handling, and one’s complement arithmetic into practical logic gate implementations. The final circuit passed all validation tests and reflects the way real-world protocols like UDP and TCP ensure data integrity.

**Screenshot on different Test Cases:**





