Project Overview

Encryption Algorithm Used

Your project uses a **substitution-based encryption algorithm**. Specifically, it uses a custom substitution table (5x5 matrix) where each letter is mapped to a unique position. The encryption involves combining the positions of plaintext characters with those of a secret key using row-column arithmetic. Special characters are handled separately by modifying their ASCII values based on the key positions.

- **Encryption Process:**
  1. **Substitution Table Initialization**: A 5x5 table is populated with characters.
  2. **Key Setup**: A fixed key `["N", "E", "D", "E", "L", "C", "U"]` is used.
  3. **Character Handling**:
     - For letters, their positions are identified in the table and combined with the key's position values.
     - For special characters, their ASCII values are directly modified by adding key position values.
- **Decryption Process:**
  1. **Key and Table Initialization**: The same key and substitution table are used.
  2. **Position Extraction**: The encrypted values are split back into their original row-column positions by subtracting the key position values.
  3. **Character Restoration**:
     - For letters, positions are mapped back to characters.
     - For special characters, ASCII values are adjusted to restore the original characters.

Test Benches Overview

Your project includes multiple test benches to validate different scenarios in the encryption-decryption pipeline:

1. **`tb_enc_dec_repetitive_letters.sv`**:
   Tests how repetitive letters (like "AABBCC...") are encrypted and decrypted to ensure consistent handling.
2. **`tb_word_enc_dec.sv`**:
   Encrypts and decrypts a word (like "HELLOW"), checking if the row-column mapping is correctly applied.
3. **`special_characters_tb.sv`**:
   Ensures that special characters (`@`, `#`, `$`, etc.) are properly encrypted and restored during decryption.

4. **`tb_enc_dec_name.sv`**:
   Encrypts and decrypts a name ("NEDELCU"), validating if key-based encryption performs correctly on names.
5. **`tb_single_letter.sv`**:
   Tests encryption-decryption for a single letter, ensuring edge cases like minimal input work correctly.
6. **`mixed_case_tb.sv`**:
   Verifies that mixed case inputs ("HeLLoW") are handled correctly, converting to uppercase for encryption and restoring the original case after decryption.
7. **`tb_enc_dec_long_message.sv`**:
   Tests a long message ("HELLOTHISISATESTMESSAGE") to validate the algorithm's scalability and performance for larger texts.

In **SystemVerilog**, the choice between **wires** and **registers** (or `logic`) depends on how the variables are used within the module, specifically how they interact with **combinational** and **sequential** logic. Here's a breakdown of why you're using them in your project:

---

1. Wires

Purpose:
`wire` is used to connect different modules or parts of your design where **continuous assignment** is expected. It represents a **physical connection** and reflects the real-time state of the connected components.

When to Use:

- When the signal is driven by **another module** (like outputs of the encryptor/decryptor).
- When using **continuous assignments** (`assign` keyword).
- When the signal is **read-only** in the current module but driven elsewhere.

In Your Project:
You use `wire` for variables like **`encrypted_text`** and **`decrypted_text`** because:

- These values are **outputs** from your `encryptor` and `decryptor` modules.
- They are **driven continuously** by the logic in those modules.

Example:

```
systemverilog
CopyEdit
wire [7:0] encrypted_text [0:MSG_LEN-1];  // Output from encryptor
wire [7:0] decrypted_text [0:MSG_LEN-1];  // Output from decryptor
```

---

2. Registers (`reg` or `logic`)

Purpose:
`reg` or `logic` is used for variables that **store** data, meaning they hold their value until explicitly updated. They are typically associated with **sequential logic** (driven by clocks) or variables assigned in **procedural blocks** (`always`, `initial`).

When to Use:

- When the signal is assigned inside **procedural blocks** like `initial` or `always`.
- When **temporary storage** is needed for intermediate calculations.
- When working with **testbenches** for simulation (since you initialize and change values over time).

In Your Project:
You use `reg` or `logic` for variables like **`plain_text`** because:

- They are initialized and modified inside **procedural blocks** (`initial`).
- They act as **inputs** to the encryption module but hold specific data like a test message.

Example:

```
systemverilog
CopyEdit
logic [7:0] plain_text [0:MSG_LEN-1];  // Input to encryptor, initialized
in testbench
```

---

3. Key Differences in Your Testbenches

Variable Type	Example	Why Used?
`wire`	`encrypted_text`, `decrypted_text`	Outputs from modules, continuously driven by the encryption/decryption logic.
`reg/logic`	`plain_text`	Inputs to modules, initialized and modified inside procedural blocks for testing.

Why Does This Matter?

- **Simulation vs Synthesis:**
    - In **simulation** (like in your testbenches), `reg/logic` helps to define behavior over time (initialize, modify, check outputs).
    - In **synthesis** (hardware realization), `wire` represents actual physical connections, while `reg/logic` might become flip-flops or latches based on how they are used.
- **Avoiding Errors:**
  Using the wrong type can cause simulation errors or mismatches in expected behavior:
    - Declaring an output as `reg` when it should be `wire` might cause synthesis issues.
    - Forgetting to declare storage variables as `reg/logic` in procedural blocks will lead to simulation errors.

In your **mixed case testbench (`mixed_case_tb.sv`)**, you track whether each character in the plaintext is **uppercase** or **lowercase** using a **case mapping array**. This array helps restore the original casing after decryption.

How the Case Mapping Works

1. **Track the Case Before Encryption:**
   - Before encryption, you loop through each character of the plaintext.
   - You check if the character is uppercase (`'A'` to `'Z'`) or lowercase (`'a'` to `'z'`).
   - You store this information in a `case_map` array:
     - `1` if the character is **uppercase**.
     - `0` if the character is **lowercase**.
2. **Convert All Characters to Uppercase for Encryption:**
   - The encryption algorithm handles only uppercase letters, so you **convert lowercase letters to uppercase** before encryption.
3. **Restore the Original Case After Decryption:**
   - After decryption, you refer back to the `case_map` array.
   - If `case_map[i] == 0`, the original character was lowercase, so you convert it back by adding **32** to the ASCII value (since the ASCII difference between uppercase and lowercase letters is 32).

Step-by-Step Example

Let's say your original plaintext is **"HeLLoW"**:

Index	**Character**	**Upper/Lower?**	**Stored in `case_map`**	**Converted for Encryption**
0	H	Uppercase	1	H
1	e	Lowercase	0	E
2	L	Uppercase	1	L
3	l	Lowercase	0	L
4	o	Lowercase	0	O
5	W	Uppercase	1	W

After **decryption**, you use the `case_map` to restore:

Decrypted Character	**Case Info (`case_map`)**	**Final Decrypted Character**
H	1	H
E	0	e
L	1	L
L	0	l
O	0	o
W	1	W

Final output after decryption: **"HeLloW"**.

The directive `timescale 1ns/1ps` in SystemVerilog (and Verilog) sets the **time unit** and **time precision** for your simulation. It tells the simulator how to interpret delays and time-related events.

Breaking Down `timescale 1ns/1ps`:

1. **`1ns` (Time Unit):**
    - This specifies the **base time unit** for the simulation.
    - All time delays (like `#10`, `#5`, etc.) will be interpreted in **nanoseconds (ns)**.
    - For example:
        - `#10` means a **10-nanosecond delay**.
        - `#5` means a **5-nanosecond delay**.
2. **`1ps` (Time Precision):**
    - This defines the **resolution** or **precision** of time calculations in the simulation.
    - It indicates that the smallest measurable time increment is **1 picosecond (ps)**.
    - The simulator can track events with **1 picosecond accuracy**, even if the time unit is nanoseconds.

Why Use `1ns/1ps`?

1. **Accuracy in Simulations:**
    - It allows for **fine-grained control** of time in simulations, especially when modeling high-speed circuits where even tiny timing differences matter.
2. **Consistency:**
    - Using nanoseconds as the time unit is standard for many digital designs, making it easy to understand and consistent across different modules.
3. **Sub-Nanosecond Events:**
    - The **1 picosecond precision** ensures that you can handle scenarios where timing differences smaller than a nanosecond are important.

What Happens If You Change It?

- **`timescale 1ns/1ns`:**
  The simulator only tracks events at the **nanosecond** level. Sub-nanosecond delays (like `#0.5`) would be **rounded** or ignored.
- **`timescale 1ps/1ps`:**
  The base time unit and precision are in **picoseconds**. Now, `#10` would represent **10 picoseconds** instead of nanoseconds.

Summary:

- `1ns`: Your simulation's **time unit** is nanoseconds.
- `1ps`: Your simulation's **precision** is picoseconds, allowing sub-nanosecond accuracy.

Why I Used SystemVerilog

1. **Enhanced Features Over Verilog:**
   - SystemVerilog extends Verilog with **advanced features** for both **design** and **verification**.
   - It provides better syntax for **modular code**, making it easier to write, read, and maintain complex designs like encryption and decryption algorithms.
2. **Improved Data Types and Structures:**
   - SystemVerilog offers **richer data types** like `logic`, `bit`, `byte`, and **dynamic arrays**, making it more flexible for handling text data and encryption keys.
   - The use of **multidimensional arrays** and **structures** simplifies complex operations like substitution tables and key management in my project.
3. **Superior Testbench Capabilities:**
   - SystemVerilog excels in **testbench creation** due to **object-oriented programming (OOP)** features and **randomization** capabilities.
   - This allowed me to create **robust testbenches** to verify various edge cases (e.g., repetitive letters, mixed cases, special characters) efficiently.
4. **Better Control Over Simulation:**
   - The **`logic`** data type removes ambiguity present in Verilog's `reg` and `wire`, reducing simulation errors.
   - SystemVerilog supports **concurrent assertions** and **functional coverage**, which ensures more reliable verification.
5. **Industry Standard for Modern Hardware Design:**
   - SystemVerilog is widely used in the **semiconductor industry** for both **design** and **verification** of digital systems.
   - Learning and applying SystemVerilog gives me **practical experience** aligned with current **industry practices**.
6. **Cleaner Syntax for Complex Operations:**
   - The **simplified syntax** and additional operators in SystemVerilog make it easier to write complex logic, such as the substitution cipher and ASCII manipulation for encryption and decryption.
7. **Supports Both RTL Design and Verification in One Language:**
   - Unlike older HDLs that separate design and verification, SystemVerilog allows me to use **one language for the entire project lifecycle**—from writing encryption/decryption logic to creating comprehensive testbenches.

Example Comparison:

- **In Verilog:**
  - Managing multi-dimensional arrays (like your substitution table) is cumbersome and less intuitive.
  - No native support for `logic`, which can cause confusion between `wire` and `reg`.
- **In SystemVerilog:**
  - Multi-dimensional arrays and dynamic arrays are straightforward.
  - `logic` simplifies variable declaration and makes code more readable.

Final Answer (Concise Version):

"I chose SystemVerilog because it extends Verilog with advanced features for both design and verification. Its rich data types, superior testbench capabilities, and clean syntax made it easier to implement and test complex encryption and decryption logic. Additionally, it's the industry standard for modern hardware design, giving me valuable experience aligned with current professional practices."

Potential Questions & Answers for Presentation

1. **Q: What kind of encryption are you using?**
   A: The project uses a substitution-based encryption algorithm inspired by classical cipher techniques. It combines the positions of characters from a substitution table with key-derived positions to create an encrypted output.
2. **Q: How do you handle special characters?**
   A: Special characters are not found in the substitution table. Instead, their ASCII values are modified by adding a numeric representation of the key's position, ensuring they're securely encrypted and can be accurately restored during decryption.
3. **Q: How does the decryption process reverse the encryption?**
   A: The decryption subtracts the key's position values from the encrypted text, retrieves the original row-column positions, and maps them back to the corresponding characters from the substitution table.
4. **Q: How do you ensure the encryption is consistent across different text cases?**
   A: Mixed case inputs are converted to uppercase during encryption, and the original casing is restored during decryption using a case map.
5. **Q: What happens if the message length exceeds the key length?**
   A: The key is reused cyclically using modulo operations (`i % SEC_LEN`), ensuring every character in the message is encrypted with a key character.
6. **Q: Are there any limitations to this encryption method?**
   A: While effective for basic text encryption, this method doesn't offer high-level security like modern cryptographic algorithms (AES, RSA). It's primarily educational and demonstrates fundamental encryption techniques.