Lab 2: Amy Crypto Machine

Submission Due Dates:

Demo: 2022/09/27 17:20
Source Code: 2022/09/27 18:30
Report: 2022/10/02 23:59

Objective

1. Getting familiar with sequential Verilog behavior modeling.
2. Practicing finite-state machines (FSMs) in Verilog.
3. Practicing memory data structure in Verilog
4. Practicing for-loop syntax in Verilog

Background

Amy is an excellent NTHU CS student. But one day, an evil alien captured her and teleported her back to 1938. To survive, she uses her computer science knowledge to get a job in the army of Allies for cryptography research. Fortunately, she carries her laptop and some FPGA boards while traveling the time machine. Amy can utilize those equipment to build an unbreakable crypto machine for the Allies to defeat the enemy. Can you help Amy finish this research?

Design Description

To build a crypto machine, we need two key components. One is an encryption algorithm, and the other is an error correction code. The first part is to ensure that the enemy cannot decode the message even if they intercept it. And the latter is to ensure that the noisy environment in the air won't make the transmitted message unreadable. The wireless communication technology in the mid-20 century is not as reliable as today.

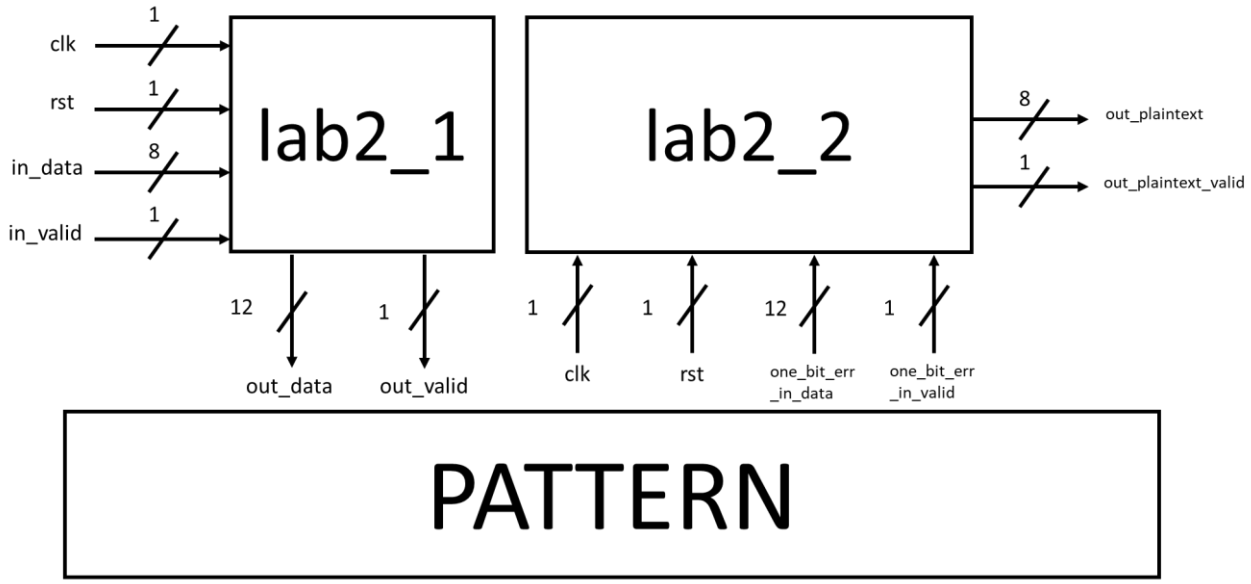


Figure1: Design overview

Hence, you must design two modules: the encoder (lab2_1) and the decoder (lab2_2). The encoder must encode the incoming message with an encryption algorithm and apply an error correction code. The output data will be sent to PATTERN module (i.e., the test stimulus) to check its correctness. The PATTERN module may also inject an error. After that, the decoder will try to fix the corrupted data and decrypt the message to plaintext.

Action Items

1 lab2_1.v (60%)

Write a Verilog module that can apply the following encryption algorithm and error correction code. This module should have a **finite state machine** with five steps:

- initialization
- Get the input data
- Apply the encryption algorithm
- Apply the error correction code
- Generate the output data

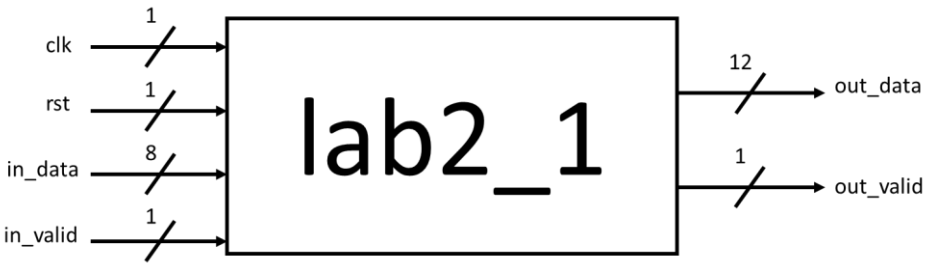


Figure 2: lab2_1 block diagram

a. Encryption algorithm

Firstly, you are required to implement a 7-bit counter, and when it counts to 127, the next count should be 0. The counter will be used for the encryption by the following equation:

$$e_i = p_i + c_i,$$

where the e stands for an encrypted message; p is the input data (in_data); c is the current counter value, and i is the index of current data.

Example:

If we get the following data {25, 100, 36, 47, 3} with the index of {1, 2, 3, 4, 5} as the input sequence, the encrypted data will be in the following table.

Index	in_data (p)	Counter Value (c)	Encrypted Data (e)
1	25	0	25 + 0 = 25
2	100	1	100 + 1 = 101
3	36	2	36 + 2 = 38
4	47	3	47 + 3 = 50
5	3	4	3 + 4 = 7

b. Error correct code

Data transmission in real life will suffer from data corruption. To solve this issue, Richard W. Hamming invented an error detection and correction algorithm in 1950 named Hamming codes. This algorithm inserts some redundant bits during data transmission to detect and correct the error bit.

This algorithm uses redundant bits for parity check. We can do either even parity or odd parity. For even parity, we will set the redundant bit to zero if there is an even number of the ones (1's) in the data. If the ones add up to an odd number, the redundant bit is set to one.

For example, if we want to transmit an 8-bit data, **1**0100001, which has three ones in it, we will add a parity bit (the 9th bit) as one to form a 9-bit data, **1** 10100001, so that there are four ones (i.e., an even number of ones). For another example, if the data to be transmitted is **111**0000**1**, we set the parity bit as zero, and the result data will be **0** 11100001.

Hence, the receiver knows the received data are correct if the number of ones is even. The received data is corrupted when there is an error (or a bit-flip; 0 → 1 or 1 → 0) to make the number of ones an odd number. The parity check with one parity bit can detect one bit-flip. Hamming code adds multiple parity bits into the transmitted message, which not only detects one error but also corrects it. In this

lab, we use **even parity** to construct the error correction code.

For more information you can check Wikipedia:

Hamming code: https://en.wikipedia.org/wiki/Hamming_code

Parity check: https://en.wikipedia.org/wiki/Parity_bit

The ordinary Hamming code can detect two errors in transmitted data but cannot correct them. To simplify our design, we assume at most **one single error** in one transmitted data. As the following table, the Hamming code adds 4 redundant bits to the 8-bit data, where $r1, r2, r3, r4$ are the redundant bits and $d1, d2, d3, d4, d5, d6, d7, d8$ are the data bits. So, there are 12 bits in the transmitted data. Remember that we adopt the **even parity** scheme.

Bit	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
Notation	d8	d7	d6	d5	r4	d4	d3	d2	r3	d1	r2	r1

The following example demonstrates the encoding of Hamming code:

First, an input data $\{101\}_{10}$ can be represented as $\{01100101\}_2$.

Bit	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
Notation	d8	d7	d6	d5	r4	d4	d3	d2	r3	d1	r2	r1
data	0	1	1	0	?	0	1	0	?	1	?	?

To apply the Hamming code of single error correction, 4 redundant (parity) bits are added as the following equations:

$$\mathrm{r1} = \mathrm{B3} \oplus \mathrm{B5} \oplus \mathrm{B7} \oplus \mathrm{B9} \oplus \mathrm{B11} = 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 0$$
$$\mathrm{r2} = \mathrm{B3} \oplus \mathrm{B6} \oplus \mathrm{B7} \oplus \mathrm{B10} \oplus \mathrm{B11} = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 0$$
$$\mathrm{r3} = \mathrm{B5} \oplus \mathrm{B6} \oplus \mathrm{B7} \oplus \mathrm{B12} = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$
$$\mathrm{r4} = \mathrm{B9} \oplus \mathrm{B10} \oplus \mathrm{B11} \oplus \mathrm{B12} = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$

where r is the redundancy, d is the data, and $\oplus$ denotes the exclusive-or operation. Therefore, the 12-bit result is listed as follows:

Bit	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
Notation	d8	d7	d6	d5	r4	d4	d3	d2	r3	d1	r2	r1
data	0	1	1	0	0	0	1	0	1	1	0	0

c. IO list:

Input Signals	Bit Width	Definition
clk	1	Clock signal
rst	1	Reset signal
in_data	8	Input data

in_valid	1	Input data valid: 1 (high) for valid

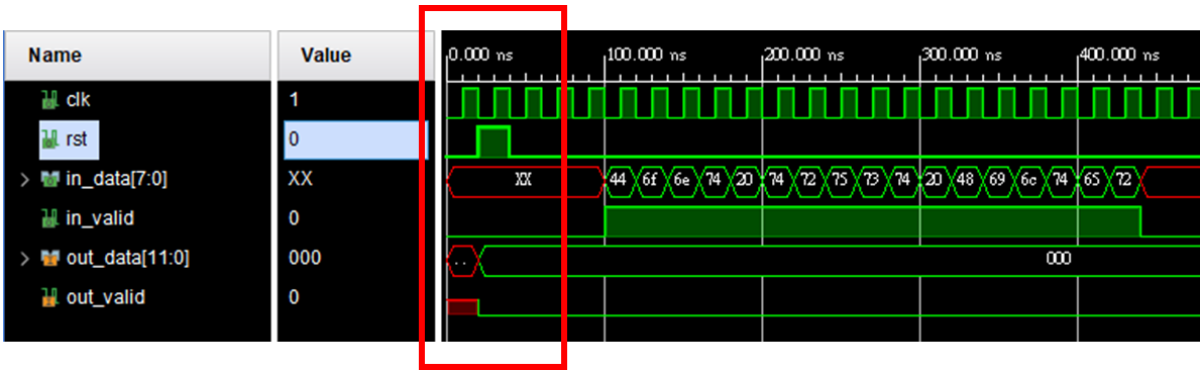
Output Signals	Bit Width	Definition
out_data	12	Encrypted data with error correct code
out_valid	1	Output data valid: 1 (high) for valid

d. Specification
   1. Reset
      - The reset signal (rst) is the active-high synchronous reset.
      - The design will be reset only at the beginning of the simulation. All output signals should be reset to 0 after the reset!
   2. Input data
      - The in_data will be valid only when the in_valid is high.
   3. Output data
      - The out_valid should be high when the out_data is ready.
   4. Simulation time
      - Please follow Lab 1 tutorial to change the runtime to 100,000 ns.
   5. Simulation
      - Please refer to Appendix A for the simulation guide.

e. PATTERN testing guide
   1. Reset
      - The PATTERN module will check whether your output signals are set to 0 after reset.
   2. Input data
      - Each test case will have a random number of the in_data ranging from 1~255.
      - After finishing one test case, the next in_data will come with a 2~10 cycle delay.
   3. Output data
      - The PATTERN module will check if you have the same number of the out_data as the in_data you got.
      - The PATTERN module will check the correctness of the out_data.
      - The out_valid and in_valid cannot be high at the same time.

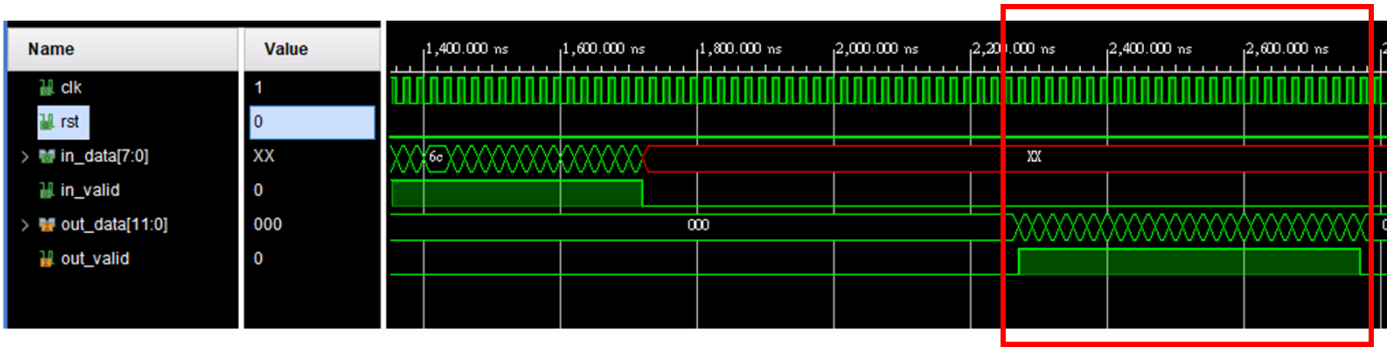
f. Waveform examples

Example for the reset (Reset is synchronous to clock edges.)

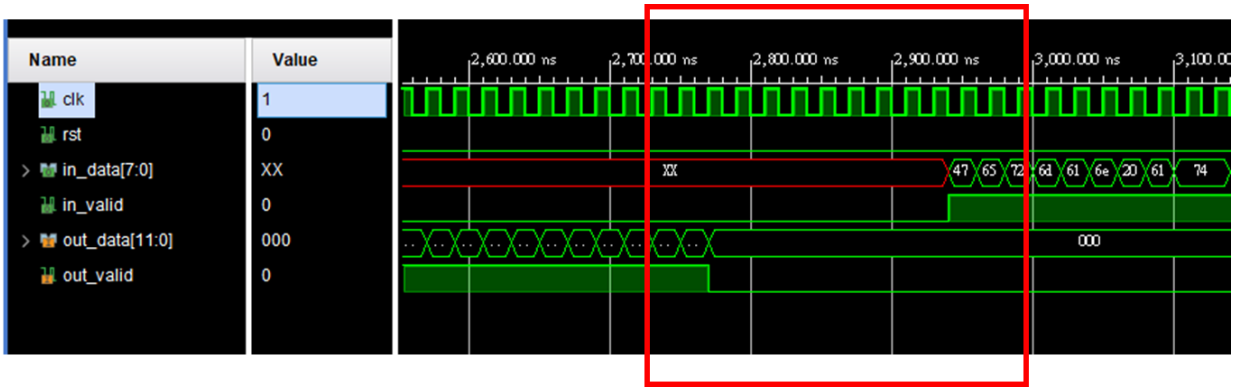


Example for the input data (Each test case has a different number of in_data, ranging from 1~255.)

Example for the output data (The number of output data should be the same as the number of input data.)



Example for the next input data (The next input will come after 2 to 10 cycles in the stimulus.)



If you pass the test patterns, you can see the similar message on the Tcl Console.

```
lab2_1  Pass Pattern NO.          0
lab2_1  Pass Pattern NO.          1
lab2_1  Pass Pattern NO.          2
```

If you pass all the test patterns, you can see this message on the Tcl Console.

```
================  lab2_1  Your colleagues used the information defeating the enemy.  ==============================
```

If you have errors in your design, you can see this similar message.

```
--------------------------------------------------------------------------------------------------------------------
                                                    FAIL!
                                            Simulation cycle limit reach
--------------------------------------------------------------------------------------------------------------------

------------------------------------------------------------
                    PATTERN NO.   0 Word index NO.       0
          answer should be :        1193 , your answer is :         68
------------------------------------------------------------
```

g. You must use the following template for your design:

```
module Encoder (
    input clk,
    input rst,
    input [7:0] in_data,
    input in_valid,
    output reg [11:0] out_data,
    output reg out_valid
);
  // Output signals can be reg or wire
  // add your design here

endmodule
```

2 lab2_2.v (40%)

In Part 2, you are required to write a **decoder** module that can apply the following decryption algorithm and correct the potential error. And we will combine the **encoder** from **lab2_1** to complete the system so that Amy can deliver the solution to the frontline to save the world! This module should have a **finite state machine** with five steps:

- Initialization
- Get the input data
- Fix the potential error in the input data
- Apply the decryption algorithm
- Generate the output data

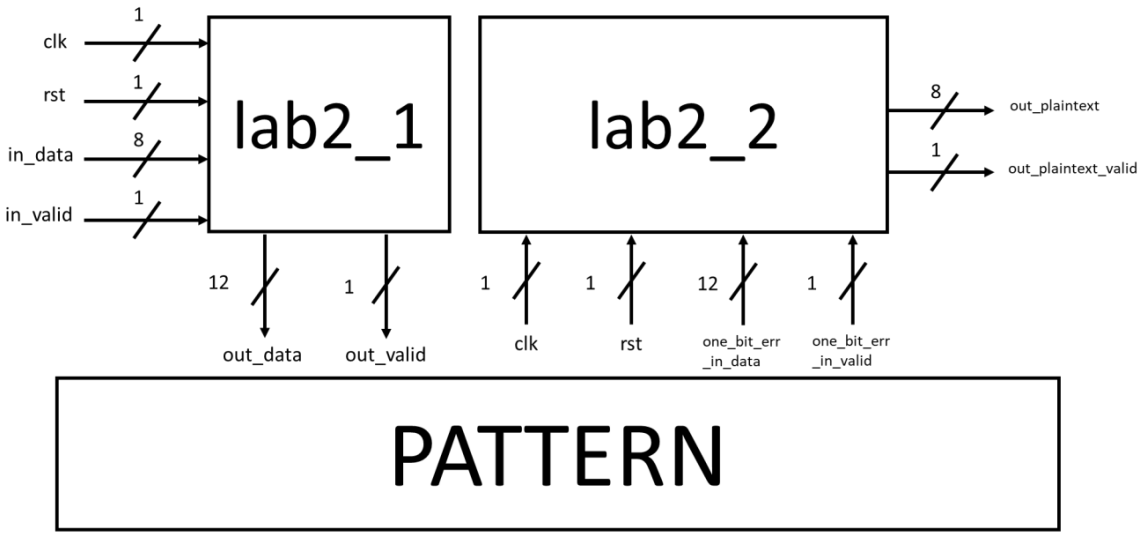


Figure 3: lab2_2 block diagram

a. Error correct algorithm

We can define four correction bits based on the previous algorithm to detect the potential error bit.

$$c1 = B1 \oplus B3 \oplus B5 \oplus B7 \oplus B9 \oplus B11$$
$$c2 = B2 \oplus B3 \oplus B6 \oplus B7 \oplus B10 \oplus B11$$
$$c3 = B4 \oplus B5 \oplus B6 \oplus B7 \oplus B12$$
$$c4 = B8 \oplus B9 \oplus B10 \oplus B11 \oplus B12$$

Since we use the **even parity** scheme, the value of each correct bit **should be 0**. If B3 happens to bit-flip during transmission, c1 and c2 will be 1 while c3 and c4 remain 0. Hence, we can identify that B3 has an error, and we can flip its value to correct the error. We list the correction table with all potential combinations for your reference to simplify this lab.

$\{c1, c2, c3, c4\}$	Error bit
1100	B3
1010	B5
0110	B6
1110	B7
1001	B9
0101	B10
1100	B11
0011	B12
1000	B1
0100	B2
0010	B4
0001	B8
Other	No error

After we finish the correction, we can remove the redundant bits and continue with the decryption to recover the plaintext.

The following example demonstrates the decoding of Hamming code:
From lab2_1 we know the output data is 011000101100.

Bit	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
Notation	d8	d7	d6	d5	r4	d4	d3	d2	r3	d1	r2	r1
data	0	1	1	0	0	0	1	0	1	1	0	0

If there is no error during transmission the correct bits will be the following.

$c1 = B1 \oplus B3 \oplus B5 \oplus B7 \oplus B9 \oplus B11 = 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 0$

$c2 = B2 \oplus B3 \oplus B6 \oplus B7 \oplus B10 \oplus B11 = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus = 0$

$c3 = B4 \oplus B5 \oplus B6 \oplus B7 \oplus B12 = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = 0$

$c4 = B8 \oplus B9 \oplus B10 \oplus B11 \oplus B12 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 0$

Since { c1, c2, c3, c4} = 4'b0000, we know there is no error during transmission.

If B12 flips during transmission, the input to lab2_2 module will be

Bit	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
Notation	d8	d7	d6	d5	r4	d4	d3	d2	r3	d1	r2	r1
data	1	1	1	0	0	0	1	0	1	1	0	0

The correct bit will become the following.

$c1 = B1 \oplus B3 \oplus B5 \oplus B7 \oplus B9 \oplus B11 = 0 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 0$

$c2 = B2 \oplus B3 \oplus B6 \oplus B7 \oplus B10 \oplus B11 = 0 \oplus 1 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus = 0$

$c3 = B4 \oplus B5 \oplus B6 \oplus B7 \oplus B12 = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 = 1$

$c4 = B8 \oplus B9 \oplus B10 \oplus B11 \oplus B12 = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1$

Then, we got the correct bits { c1, c2, c3, c4} = 4'b0011. From the correction table, we know the error bit is B12. So we need to correct it by flipping its value.

Bit	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1
Notation	d8	d7	d6	d5	r4	d4	d3	d2	r3	d1	r2	r1
data	0	1	1	0	0	0	1	0	1	1	0	0

Therefore, the corrected data is 01100101, even if there is an error during transmission.

b. Decryption algorithm

Decrypting the message is simply the reverse of encryption. We need to subtract the counter value:

$$p_i = e_i - c_i,$$

where the e stands for the encrypted message; the p is the plaintext; the c is the current counter data, and the i is the index of current data.

c. IO list

Input Signals	Bit Width	Definition
clk	1	Clock signal
rst	1	Reset signal
one_bit_err_in_data	12	Input data signal which may have one bit error
one_bit_err_in_valid	1	High for input data valid

Output Signals	Bit Width	Definition
out_plaintext	8	Plaintext data
out_plaintext _valid	1	High for plaintext data valid

d. Specification

1. Reset
   - The reset signal (rst) is the active-high synchronous reset.
   - The design will be reset only at the beginning of the simulation. All output signals should be reset to 0 after the reset!
2. Input data
   - The one_bit_err_in_data will be valid only when the one_bit_err_in_valid is high.
3. Output data
   - The out_plaintext_valid should be high when the out_plaintext_data is ready.
4. Simulation time
   - Please follow Lab 1 tutorial to change the runtime to 100,000 ns.
5. Simulation
   - Please refer to Appendix A for the simulation guide.

e. PATTERN testing guide

1. Reset
   - The PATTERN module will check whether your output signals are set to 0 after reset.
2. Input data
   - Each test case will have a random number of the one_bit_err_in_data ranging from 1~255.
   - After finishing one test case, the next one_bit_err_in_data will come with a 2~10 cycle delay.
3. Output data
   - The PATTERN module will check if you have the same number of the out_plaintext_data as the one_bit_err_in_data you got.

- The PATTERN module will check the correctness of the out_plaintext_data.
- The out_plaintext_valid and one_bit_err_in_valid cannot be high at the same time.

f. You must use the following template for your design:

```
module Decoder (
    input clk,
    input rst,
    input [11:0] one_bit_err_in_data,
    input one_bit_err_in_valid,
    output reg [7:0] out_plaintext,
    output reg out_plaintext_valid
);
  // Output signals can be reg or wire
  // add your design here

endmodule
```

3 Questions and Discussion

Please answer the following questions in your report.

A. In this lab, our reset signal is a synchronous reset. What if it is an asynchronous reset? And how to modify your design to implement an asynchronous reset?

B. If you are not allowed to use for-loop to initialize your memory, what should you do instead?

4 Guidelines for the report

Your report should include but not limit to the following items.

A. **Lab Implementation (35%)**

You may elaborate the following.

1. Block diagram of the design with explanation
2. Partial code screenshot with the explanation: you don't need to paste the entire code into the report. Just explain the kernel part.
3. Event-based FSM with the explanation:
   Note1: Use events to describe the transition of FSM but not signals logic statement.
   Note2: You don't need to do this part if there is no FSM in this lab.

B. **Questions and Discussions (50%)**

Provide your answer to the Questions and Discussions in the lab assignment.

C. **Problem Encountered (10%)**

Describe the problems you encountered, solutions you developed, and the discussion. Explaining them with code segments or diagrams is recommended.

D. **Suggestions (5%)**
Any suggestions for this course are more welcome.
(If not, you may also post a joke. A funny one, please. It is not mandatory and has nothing to do with the grading. But it would undoubtedly amuse us. ☺)

5 Hint

1. You can use the storage/memory (in this case, flip-flops) to store input/output data.
2. You can use for-loop to initialize the storage/memory
3. It is easier to perform error decoding using concatenation operators.
4. You can use practice as a template to design your lab2.

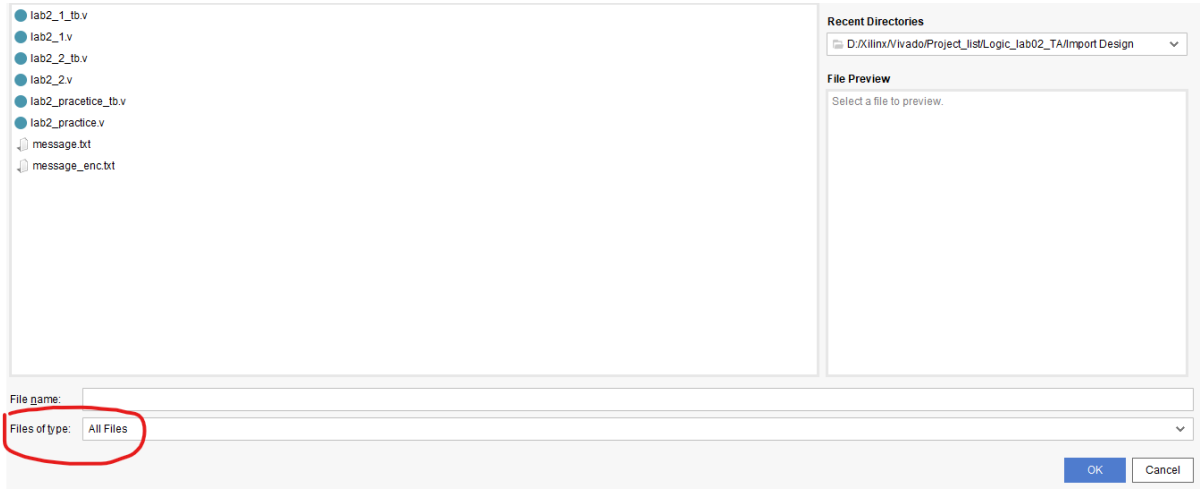
Attention

- ✓ DO NOT copy-and-paste code segments from the PDF materials. It may also paste invisible non-ASCII characters, leading to hard-to-debug syntax errors.
- ✓ You should hand in two source files, including **lab2_1.v, and lab2_2.v**. Upload each source file individually. DO NOT hand in any compressed ZIP files, which will be considered an incorrect format.
- ✓ You should also hand in your report as **lab2_report_StudentID.pdf** (i.e., lab2_report_108456789.pdf).
- ✓ You should be able to answer questions of this lab from TA during the demo.
- ✓ You may also add a **$monitor** in the testbench to show all the inputs and outputs during the simulation.
- ✓ Before the simulation, you may need to change the runtime to 100,000ns (or a large enough period) in “Simulation Settings”.

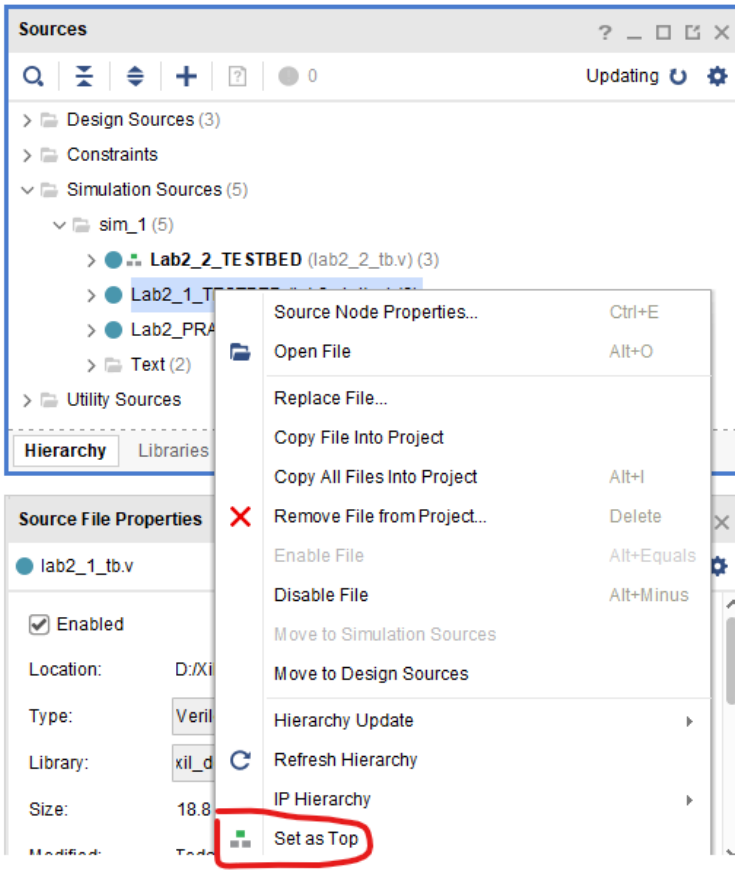
Appendix A

How to add simulation files and change the top module

1. Add **lab2_1.v** and **lab2_2.v** to design sources
2. Add **lab2_1_tb.v, lab2_2_tb.v, message.txt, message_enc.txt** to simulation sources
   A. If you can't find the message.txt/message_enc.txt
      i. Remember to change the file type to All Files
      ii.



3. If you want to do lab2_1, you can right click the lab2_1_TESTBED and left-clicks **Set as Top.**



4. The simulation sources will change (refer to the following figure). Then, you can continue the simulation of lab2_1.