CDS206_Lab3-Safe FSM encoding

Introduction

In lecture 4, we studied the fundamentals of information redundancy.

In a nutshell, we discussed

1. **Parity Bits:** In a parity check, an extra bit (parity bit) is added to the data to make the total number of 1-bits either always even (even parity) or always odd (odd parity). This allows the detection of single-bit errors. However, it cannot correct the mistakes or detect multiple-bit errors.
2. **Checksums:** A checksum is a value calculated from a data set and transmitted (or stored) along with it. Upon receipt, the same calculation is performed again, and if the result does not match the transmitted (stored) checksum, it indicates that the data has been corrupted.
3. **Cyclic Redundancy Check (CRC):** CRC is a more robust method for error detection, which involves treating the data as a large binary number and dividing it by a set value. The remainder of this division is then sent with the data. Upon receipt, the calculation is repeated; if the rest does not match, it indicates an error. CRC is good at detecting common types of errors in communication channels, such as burst errors.
4. **Error Correcting Codes (ECC):** ECC like Hamming code, Reed-Solomon, and others not only detect errors but also correct them. They add extra bits (more than the parity bit) and perform more complex calculations, allowing them to detect which bit caused the error and correct it. This is especially useful in situations where re-transmission of data is costly or impractical.

So, in the context of error codes, redundancy is purposefully added information that allows a system to check and correct the data it receives or stores, ensuring the data remains accurate and reliable.

FSM basics

Lets start with some basics about FSMs.

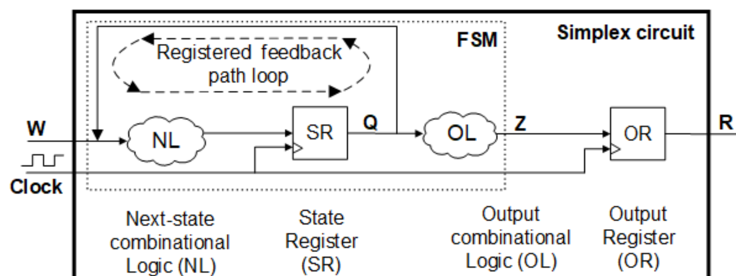


Fig. 2: Finite State Machine (FSM)

A finite state machine (FSM) is a model of computation or behaviour composed of a finite number of states, transitions between those states, and actions. It's called finite because it can be in a limited number of states. FSMs are often used in sequential logic circuits.

A finite state machine is usually implemented in hardware with flip-flops (see SR) and logic gates (see NL and PL). Each state of the FSM is represented by a unique binary value stored in the SR. Transitions between states are driven by the current state and the input signals, processed through logic gates.

There are two types of finite-state machines in the hardware context: Moore and Mealy. In a Moore machine, the current state determines the outputs. In a Mealy machine, the outputs are determined by the current state and the current inputs.

Here is a high-level overview of how a finite-state machine can be implemented in hardware:

1. **State Register (SR):** First, we need a way to store the current state of the machine. This is usually done with flip-flops, which are memory elements that can store a binary value. The number of states in the FSM determines the number of flip-flops required. For example, if the FSM has 8 states, we would need 3 flip-flops (since 3 bits can represent 2^3=4 unique values).
2. **Next state Logic (NL):** Next, we need to implement the transition logic that determines what the next state should be based on the current state and the inputs. This is typically done with a combination of AND, OR, and NOT gates (i.e., basic digital logic gates).
3. **Output Logic (OL):** Finally, we need to implement the output logic that determines what the output should be for each state. In a Moore machine, the output only depends on the current state, so this is relatively straightforward. In a Mealy machine, the output depends on both the current state and the inputs, which requires a bit more complex logic.

At each clock cycle, the current state and inputs are fed into the next state logic (NL), which determines the next state. The flip-flops then update to this next state. The current state (and possibly inputs in the case of a Mealy FSM) also go through the output logic to produce the outputs.

Exercise 1

In this lab, we will implement the following eight-state (s0 - s1) binary Finite State Machine (FSM) and perform fault injection on the application level, i.e., by flipping bits in the register that stores the state of the FSM.

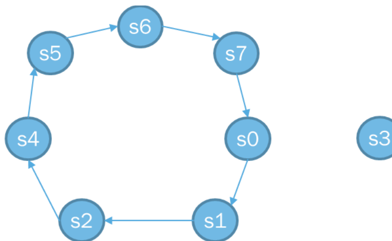
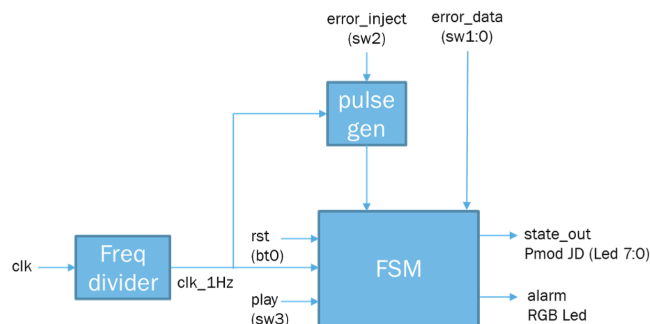


Figure 2: Binary FSM

Download the source file from http://gunet2.cs.unipi.gr/modules/document/file.php/CDS126/Lab exercises/lab3/003 - labs_srcs.zip and create a Vivado project for your ZyboZ7 board.

The block diagram of the design is the following:



FSM: The FSM block (FSM.vhd) implements the binary FSM shown in Figure 2. The FSM takes the following inputs

```
1  entity FSM is
2  generic (N: integer := 3);
3   port (
4      rst          : in std_logic; -- Reset to S0
```

```
5      clk          : in std_logic; -- Clock input
6      play         : in std_logic; -- Starts the FSM
7      state_out    : out std_logic_vector(N-1 downto 0); -- The output of the FSM SR
8
9      error_inject : in std_logic; -- Inject a fault
10     error_data   : in std_logic_vector(1 downto 0); -- Specify which bit flip in the SR
11     alarm        : out std_logic -- The FSM is in a non valid state
12 );
13 end FSM;
```

Clock divider: Takes as input 125MHz and generates a 1Hz clock (1 second). We used it to observe through the LEDS how the FSM transitions from one state to another.

Pulse Generator: Provides a pulse on the rising edge of Zybo's switch → SW2

Implementation and test

1. Read carefully the comments in the `top_level.vhd`
2. make N = 3 in line 54 of the `top_level.vhd`
3. Generate the bitstream and program the FPGA
4. Connect the PMOD leds to the JD connector
5. Turn on SW3 (play) and see how the state of the FSM transitions from one state to another (S0 → S1 → S2 → S4 → S5 → S6 → S7 → S0) or (000 → 001 → 010 → 100 → 101 → 110 → 111 → 000)
6. Turn off SW3 (play). The FSM should stop changing states.
7. Choose with switches SW1 and SW0 which bit to flip in the SR. For example, if SR=000 and you want to make it 011 →Config SW1 = 1 and SW = 0
8. Turn on SW2 (inject) to inject the fault. You will see that the SR LEDs have changed to 100.
9. Turn on SW3 (play) and see how the states transition. If the FSM was at state 000 and you injected a fault that put it at state 100, the next state should be 101.
10. Now reset the FSM by pressing the button BT0
11. Follow steps 4 to 8, but inject a fault that will put the FSM to state S3 (011).
12. What do you observe? (open a notepad and write your observation)
13. Modify the FSM.vhdl file and change line 65 to `next_state <= s0 when current_state = s3 else`
14. Execute steps 4-8.
15. What do you observe? (open a notepad and write your observation)

Exercise 2

In your sources, you will find an FSM_Hamming3.vhd FSM.

Instantiate it, implement and perform fault injection as for the previous binary FSM

1. make N = 6 in line 54 of the top_level.vhd
2. Modify line 77 →   entity work.FSM_Hamming3(Behavioral)
3. Check
4. Generate the bitstream and program the FPGA
5. Perform fault injection
6. What do you observe? (open a notepad and write your observation)