UNIVERSITI MALAYSIA PAHANG
AL-SULTAN ABDULLAH

FACULTY OF ELECTRIC AND ELECTRONICS ENGINEERING

TECHNOLOGY

BHE3233

DIGITAL SYSTEM DESIGN

SEM 2 2024/2025

PROJECT: DICE GAME CONTROLLER

DR. NURUL HAZLINA BINTI NOORDIN

STUDENT ID	**NAMA**
HC22023	IKRAM UMAR AZRA'I BIN TARMIZI
HC22004	CHUN KEAT A/L ELEK

Table of Contents

1.0 Project Background and Objective

Project Background

Project Title: Dice Game Controller

This project is focusing on designing a digital dice controller for two players who take turn to roll the dice and the highest number (score) will be nominate as winner. This system enables the players to take turns rolling a virtual dice using switches controls and to ensure liability of the dice, the number is generated using Linear Feedback Shift Register (LSFR) and displayed on a 7-Segment display. The project aims to simulate traditional dice gameplay through field-programmable gate array (FPGA).

Players interact with the dice game using simple physical controls, which are:

- A power switch to turn on and off the system
- A roll switch to simulate the dice roll and stop the counting
- A reset button to clear the current state to start a new round

Objective

As mention before the primary objective of this project is to develop a hardware-based dice game controller by using Verilog HDL to be implemented in an FPGA system. The system design is intended to simulate a rolling of a 6-sided dice as realistic as possible and showing the virtual output on 7-Segment display.

Other objectives included:

- Implement a pseudo-random number generator using Linear Feedback Shift Register (LSFR) to simulate the randomness of the dice rolling.
- Design a Finite State Machine (FSM) in order to manage the game flow such as rolling logic, timing and reset functionality.
- Test the liability of the system through simulation (ModelSim Altera) thus validate the design on actual FPGA board.
- Simulating fair and repeatable dice rolls, allowing multiple players to take turns using the same controller.

2.0 Block and Workflow Diagram

2.1 Block Diagram Description

The Dice Game Controller consists of several key blocks working in order to implement the digital dice mechanism. The system operates based on user inputs, logic control and a display for the output. The system acts as follows:

- **Power Switch**
  Act as the main controller to activate or deactivate the system. It enables the system to become responsive to the user input.

- **Roll Switch**
  Triggers the dice rolling mechanism when input high (Switch ON). It sends signal to Control Unit to enter the dice rolling state.

- **Control Unit**
  Manages the game logic, this included the state transitions from fix value to rolling and stop. It also controls the speed of the Random Number Generator through timing functions

- **Random Number Generator**
  By implementing the Linear Feedback Register (LSFR) to produce pseudo-random values from 1 to 6. This helps to simulate the dice rolling properties.

- **Stop Switch**
  Determine when the rolling animation should stop.

- **7-Segment Display**
  Displays the final dice value once the rolling of the dice is complete. The dice value will be updated in real time and fix once stopped

- **Reset Switch**
  Restart the system again to the rolling point for a new round

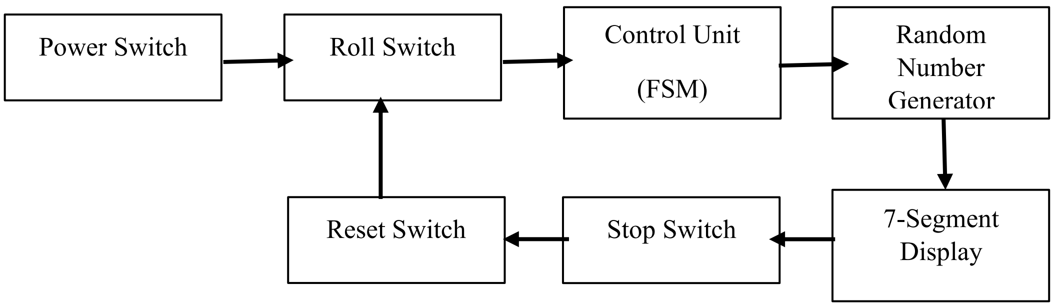


Figure 1 : Block Diagram representation

2.2 Workflow Diagram Description

The workflow diagram shows the sequence of operation from powering the FPGA to rolling and displaying the dice value. It also represents the visual implementation in the Verilog code.

The operational flow is as follows:

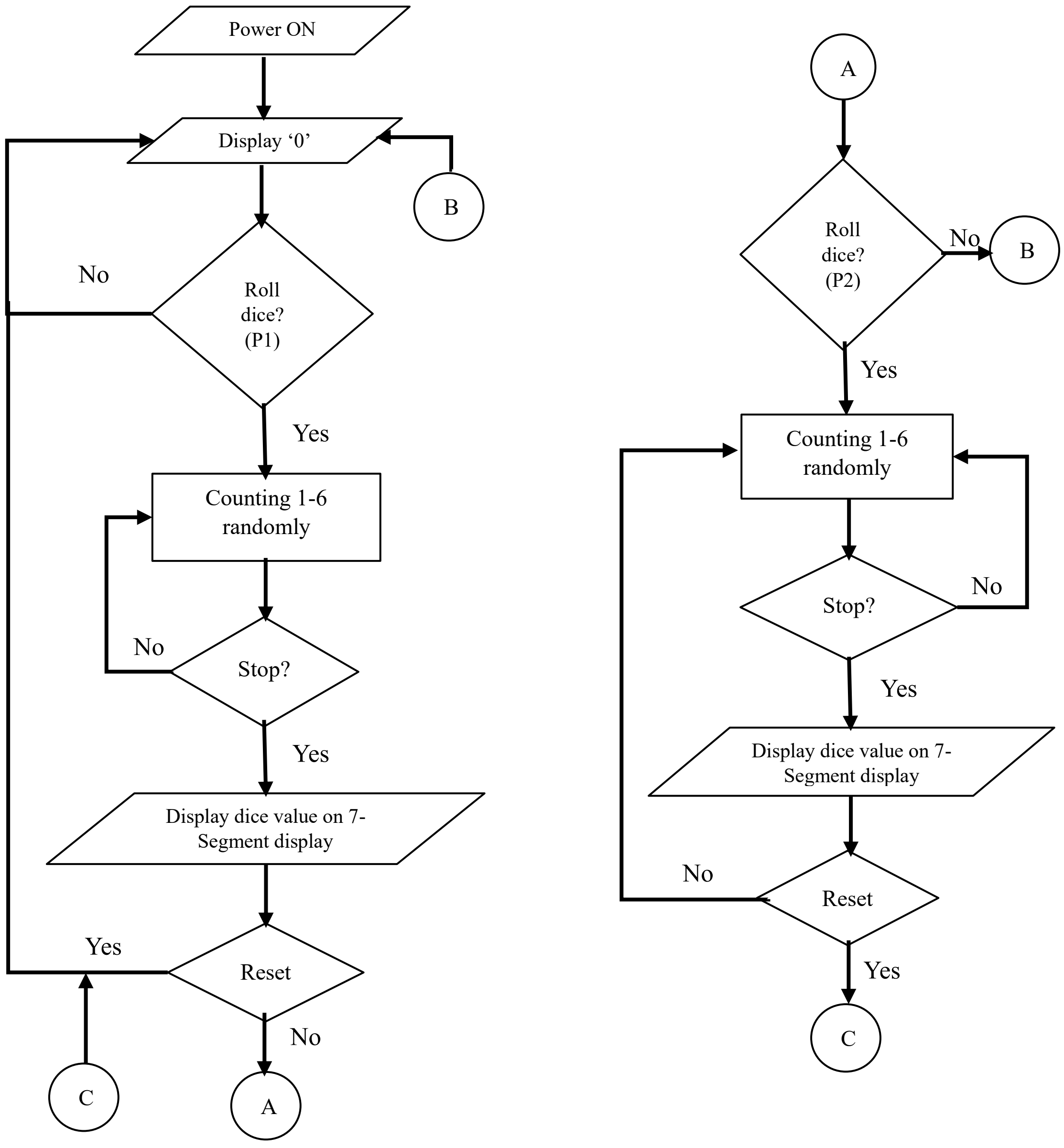


Figure 2 : Workflow Diagram representation

Note: P1 is Player 1, P2 is Player 2

3.0 Design Methodology

3.1 Finite State Machine (FSM) states

The dice game uses Moore FSM to control the logic flow of the system as its output depends only on the current state, not directly on the input. This is because the dice value is updated based on the timing edge which affect the output instead of directly by input value. No output is updated instantly when receiving input of the system. Moore is also chosen as it is simple to design and debug.

The system includes the following states, the output in the diagram represents in decimal value:

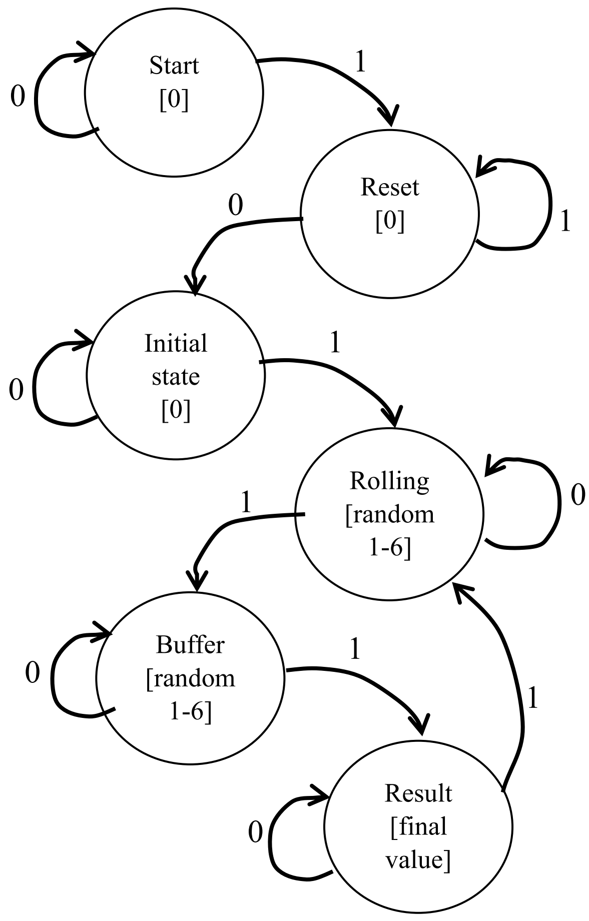
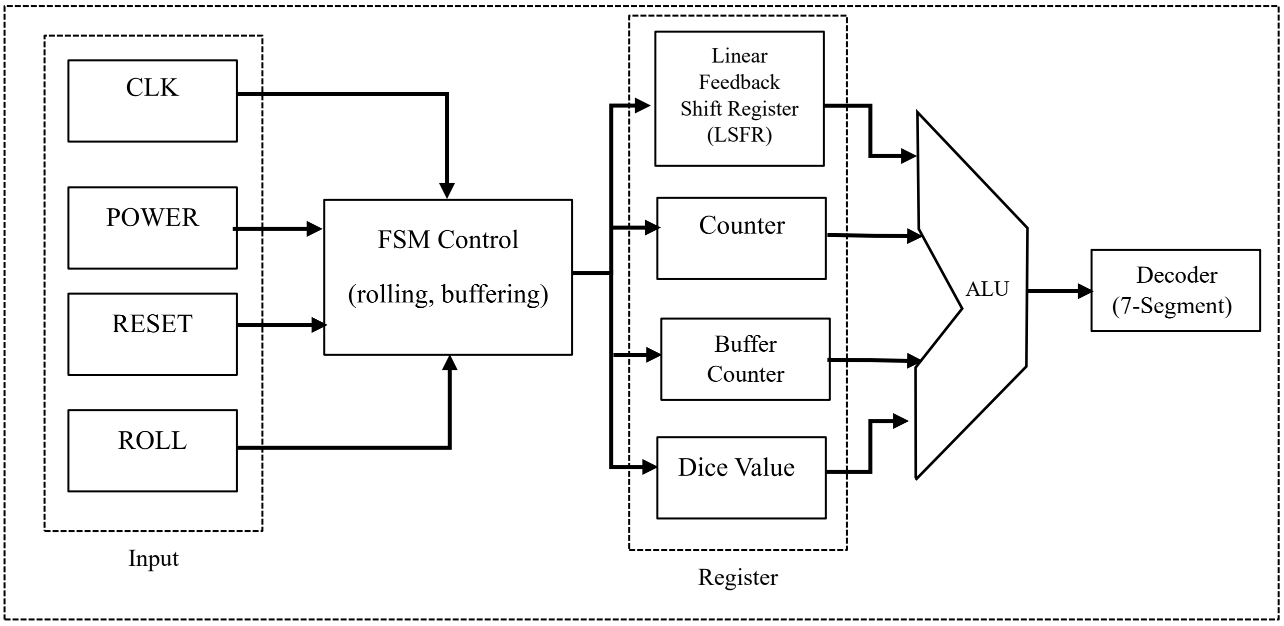


Figure 3 : Moore's FSM representation

3.2 Datapath Description

The first part of the data is the Player Dice Controller Block, where it consists of registers and control logic for single player to perform dice rolling:

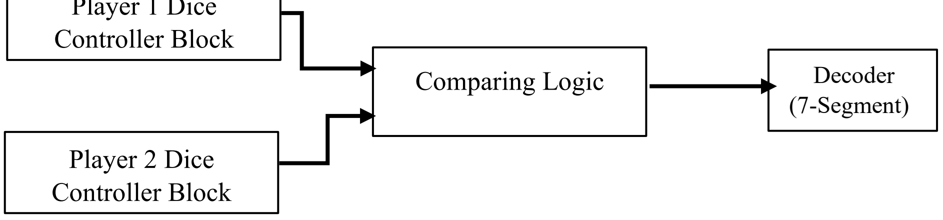
- **Programmer Visible Registers:** Internal Counter, Counter, Buffer Counter, Dice Value
- **Control Logic:** FSM Control, where it functions through inputs (CLK, POWER, RESET, ROLL)
- **Functional Unit:** ALU calculate and process the counter used in the program to get the dice value
- **Output unit:** 7-Segment Decoder to display the dice value



Player Dice Controller Block

The second part is the Comparing Logic Block to perform comparison between Player 1 and Player 2 dice results, for this part it consists of:

- **Inputs:** Dice Value from Player 1 and Player 2 from Dice Controller Block.
- **Functional Unit:** Compare Logic using simple combinational comparator logic to determines which player rolled higher number or if the results is a draw.
- **Output Functional Unit:** 7-Segment Decoder to display the winner (Player 1 as "1", Player 2 as "2" or Draw as "0")


Player 1 Dice Controller Block
Player 2 Dice Controller Block
Comparing Logic
Decoder (7-Segment)

3.3 Modules Used

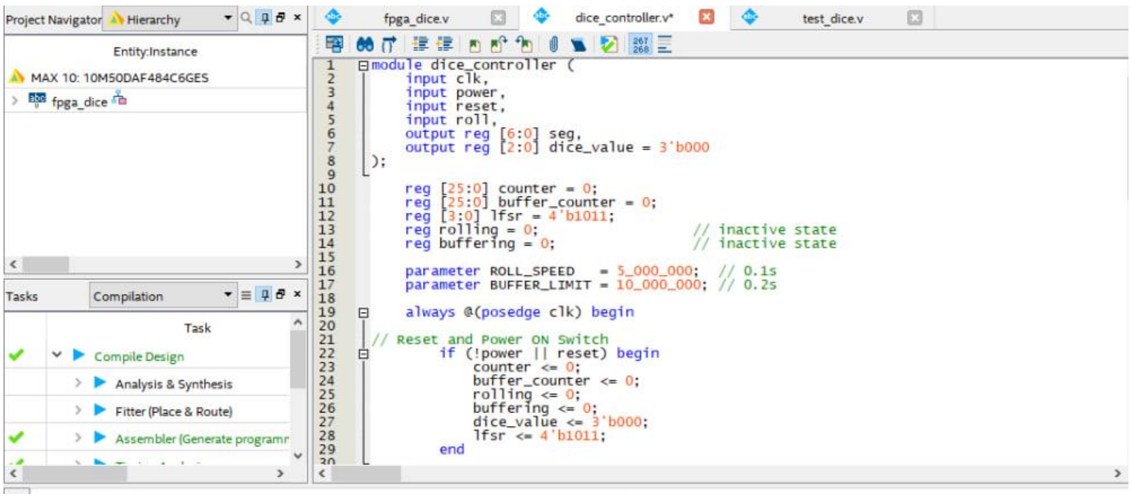
For this project, the module is separate into two different modules in order to make the system easier to manage and debugging.

I. dice_controller module

The module implements the logic from a single player's dice rolling process. In this part, the module has been included with a simple Finite State Machine (FSM) with two main states which are the rolling and buffering. The rolling state is activated when the player turns on the Roll Switch, triggering the dice to display a changing dice value at a fixed rate. When the player turns off the switch, the controller will enter a buffering state, which is a short delay before the animation is stopped. The module uses internal registers for its operation:

- **Internal Counter:** Continuously increments to provide a dynamic value for dice generation
- **Counter and Buffer Counter:** Manage timing for animation updates and buffering delay
- **Dice Value:** Holds the current dice number (1 to 6)

The dice value is being calculated using modulo operation to ensure a valid range. This module also receives inputs of clock (clk) power, reset and roll trigger.



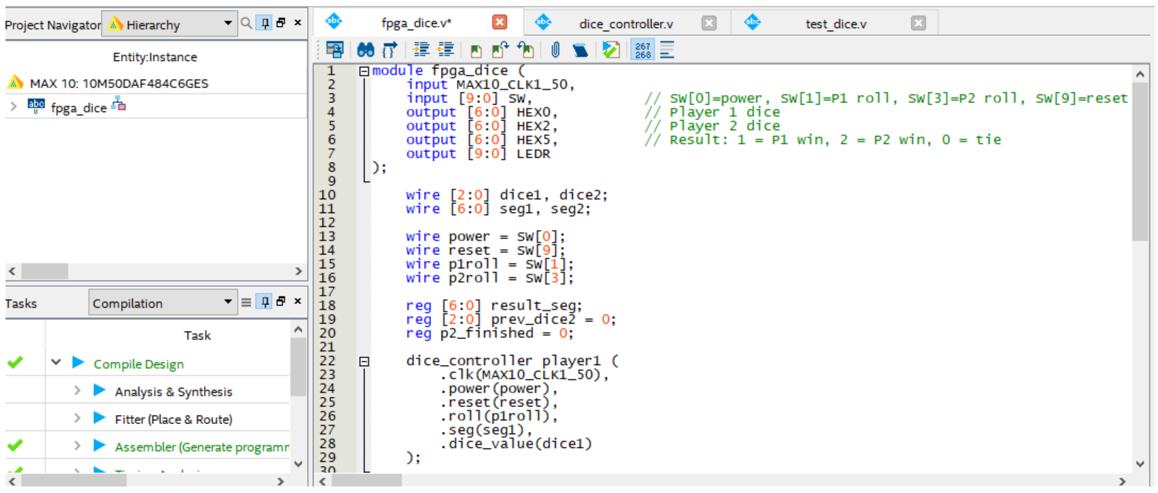
```
module dice_controller (
    input clk,
    input power,
    input reset,
    input roll,
    output reg [6:0] seg,
    output reg [2:0] dice_value = 3'b000
);

    reg [25:0] counter = 0;
    reg [25:0] buffer_counter = 0;
    reg [3:0] lfsr = 4'b1011;
    reg rolling = 0;                        // inactive state
    reg buffering = 0;                      // inactive state

    parameter ROLL_SPEED   = 5_000_000;  // 0.1s
    parameter BUFFER_LIMIT = 10_000_000; // 0.2s

    always @(posedge clk) begin

// Reset and Power ON Switch
        if (!power || reset) begin
            counter <= 0;
            buffer_counter <= 0;
            rolling <= 0;
            buffering <= 0;
            dice_value <= 3'b000;
            lfsr <= 4'b1011;
        end
```

Figure 4: dice_controller module overview

II. fpga_dice module

The fpga_dice module serves as the top-level integration or top-level hierarchy for this project. It differentiates two independent **dice_controller modules,** one for **Player 1** and the other is for **Player 2** to allow both players to roll their own dice independently. Each player's dice value will be display on two separate 7-Segment displays.

For further improving the game experience, this module also contains Comparing Logic to determine the winner and output the result ("1" for Player 1, "2" for Player 2, "0" is it a draw) on a third 7-Segment display. The module handles the input from user or the FPGA board which are switches (SW) and connects output to the display (HEX0, HEX2, and HEX5).



```verilog
module fpga_dice (
    input MAX10_CLK1_50,
    input [9:0] SW,              // SW[0]=power, SW[1]=P1 roll, SW[3]=P2 roll, SW[9]=reset
    output [6:0] HEX0,           // Player 1 dice
    output [6:0] HEX2,           // Player 2 dice
    output [6:0] HEX5,           // Result: 1 = P1 win, 2 = P2 win, 0 = tie
    output [9:0] LEDR
);

    wire [2:0] dice1, dice2;
    wire [6:0] seg1, seg2;

    wire power = SW[0];
    wire reset = SW[9];
    wire p1roll = SW[1];
    wire p2roll = SW[3];

    reg [6:0] result_seg;
    reg [2:0] prev_dice2 = 0;
    reg p2_finished = 0;

    dice_controller player1 (
        .clk(MAX10_CLK1_50),
        .power(power),
        .reset(reset),
        .roll(p1roll),
        .seg(seg1),
        .dice_value(dice1)
    );
```

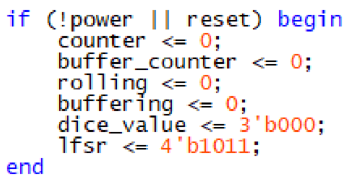
Figure 5: fpga_dice module overview

4.0 Verilog Code Highlight

4.1 FSM Design in dice_controller

The dice_controller module uses a simple Finite State Machine (FSM) implemented with two internal flags which are “rolling” and “buffering”. The FSM controls the behaviour how the dice will roll. We can see this part on different line from the coding such as:

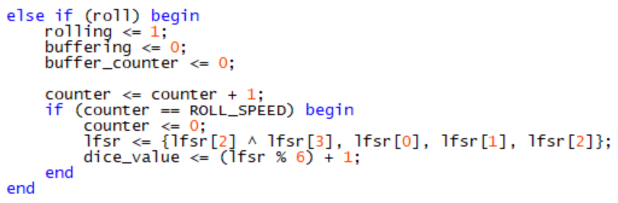
- The FSM starts as inactive state after Power Switch is turn on



```
if (!power || reset) begin
    counter <= 0;
    buffer_counter <= 0;
    rolling <= 0;
    buffering <= 0;
    dice_value <= 3'b000;
    lfsr <= 4'b1011;
end
```

Figure 6

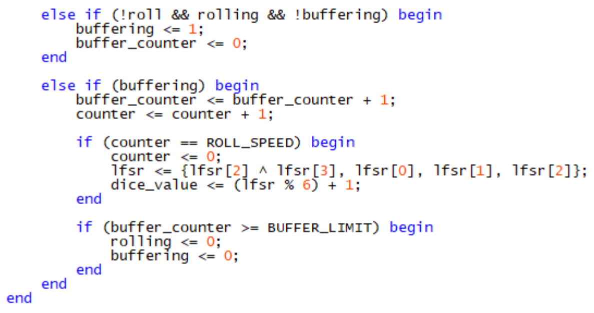
- When the player wants to roll their dice, the Roll Switch is turn on, then FSM enters the rolling state, where the dice values will being update periodically based on the (counter) to imitate real dice.



```
else if (roll) begin
    rolling <= 1;
    buffering <= 0;
    buffer_counter <= 0;

    counter <= counter + 1;
    if (counter == ROLL_SPEED) begin
        counter <= 0;
        lfsr <= {lfsr[2] ^ lfsr[3], lfsr[0], lfsr[1], lfsr[2]};
        dice_value <= (lfsr % 6) + 1;
    end
end
```

Figure 7

- When the Roll Switch is turn off, the FRM transitions to the buffering state introducing a delay using “buffer_counter”



```
    else if (!roll && rolling && !buffering) begin
        buffering <= 1;
        buffer_counter <= 0;
    end

    else if (buffering) begin
        buffer_counter <= buffer_counter + 1;
        counter <= counter + 1;

        if (counter == ROLL_SPEED) begin
            counter <= 0;
            lfsr <= {lfsr[2] ^ lfsr[3], lfsr[0], lfsr[1], lfsr[2]};
            dice_value <= (lfsr % 6) + 1;
        end

        if (buffer_counter >= BUFFER_LIMIT) begin
            rolling <= 0;
            buffering <= 0;
        end
    end
end
```

Figure 8

- After the buffering period ends, the FSM returns to the inactive state (Figure 6) if the player turns on the Reset Switch.

FSM control is used in the main function “always @(posedge clk0)” to ensure the logic inside is only triggered on the rising edge of the clock, when the clock changes from 0 to 1.

4.2 Pseudo-random Dice Value Generation

To imitate the real-life dice behaviour, the dice value is generated using a pseudo-random method that updates the number as it in rolling and buffering states of the FSM. The target is to produce a dice number between 1-6 that appear random on the display.

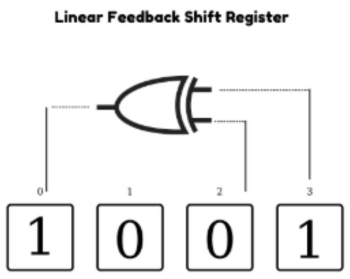


Figure 9: LSFR Circuit

Thus, the number is being generated using a 4-bit Linear Feedback Shift Register (LSFR). The LSFR is a simple circuit that generates a sequence of binary values by using a XOR gate as the based making the number to be appeared random. In this design, the same principle is applied by applying an XOR operation between certain bits of the register.

```
counter <= counter + 1;
if (counter == ROLL_SPEED) begin
    counter <= 0;
    lfsr <= {lfsr[2] ^ lfsr[3], lfsr[0], lfsr[1], lfsr[2]};
    dice_value <= (lfsr % 6) + 1;
end
```

Figure 10

At each clock cycle during the rolling process, the bits of LSFR are shifted to the right, and the new MSB bit is determined by XOR bit 2 and bit 3 of the previous state. This creates a repeating but an almost random sequence of 4-bit values.

To control the limit of the 4-bit values to be in between 1 to 6, the LSFR will be through a modulo arithmetic.

```
dice_value <= (lfsr % 6) + 1;
```

Figure 11

4.3 Random Number Generation: LSFR vs Internal Counter

I. Linear Feedback Shift Register

As has been mentioned before from **Section 4.2,** the dice value was generated using a 4-bit Linear Feedback Shift Register (LSFR). This system however has some problem which caused the sequence value to be only appear 2,3, and 6 only, which is not as a real-life dice rolling state.

This is cause by the fact that initial value of LSFR has been set to 4'b1011 and the concept of XOR and shifted right causing some numbers are impossible to be produce. The sequence of the LSFR can be shown as:

LSFR sequence (Shifted right flow)	Modulo arithmetic	Output (7-Segment display)
1011	(11 % 6) + 1 = (5) + 1 = 6	6
1101	(13 % 6) + 1 = (1) + 1 = 2	2
1110	(14 % 6) + 1 = (2) + 1 = 3	3
0111	(7 % 6) + 1 = (1) + 1 = 2	2

Table 1

As can be seen from the table the number sequence will only have 2,3, and 6 on the display, making the dice controller game's experience is not as real-life like.

II. Internal Counter

To further improve this random number sequence, an improved version of the previous design (LSFR) has been replaced with a **32-bit internal counter** that will runs continuously and increments on every clock cycle.

The internal counter is defined as 32-bit to ensure it can run for a long time before return back to bit 1 to avoid overflowing. As the typical FPGA frequency is 50Mhz, this will be divided by total bit or total toggles cycle of the bit (2^n), helping generator to reset toggle at longer period since it has lower frequency. It can be represented by:

$$\frac{50\ MHz}{2^{32}} = 0.012\ Hz$$

This made the bit will be reset after 30.52 µs ($period = \frac{1}{f}$) ensuring there is no overflow and the FPGA can process the input bit accurately. But for this random number we using only bit 13 until bit 15, since the number 6 is 3'b110, and these bits are changing slow enough for human eyes to see the changing of the dice value and better game feel. The frequency of which this bit occur can be represented:

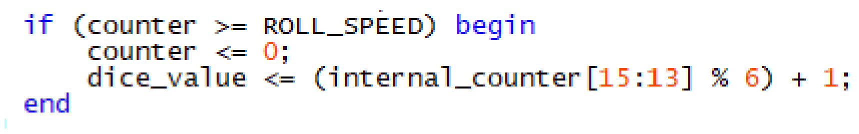
Bit number (n)	Toggles every (2^n)	Frequency($\frac{50\ MHz}{2^n}$)
13	8192 cycles	6.1 kHz
14	16384 cycles	3.0 kHz
15	32768 cycles	1.5 kHz

Table 2

The bit used **(bit 13 to bit 15)** has be defined in the modulo coding (Figure 12) and this internal counter will count from the **initial value** define in the coding (Figure 13) **until 7** due to possible maximum combination of 3b'111 like a usual counter process, but this has been limited to 6 using modulo arithmetic:

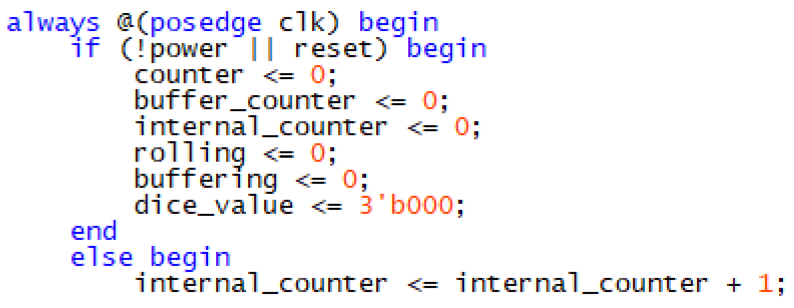
Internal counter [15:13]	Modulo arithmetic	Output (7-Segment display)
000	(0 % 6) + 1 = 1	1
001	(1 % 6) + 1 = 2	2
010	(2 % 6) + 1 = 3	3
011	(3 % 6) + 1 = 4	4
100	(4 % 6) + 1 = 5	5
101	(5 % 6) + 1 = 6	6
110	(6 % 6) + 1 = 1	1
111	(7 % 6) + 1 = 2	2

Table 3



```
if (counter >= ROLL_SPEED) begin
    counter <= 0;
    dice_value <= (internal_counter[15:13] % 6) + 1;
end
```

Figure 12



```
always @(posedge clk) begin
    if (!power || reset) begin
        counter <= 0;
        buffer_counter <= 0;
        internal_counter <= 0;
        rolling <= 0;
        buffering <= 0;
        dice_value <= 3'b000;
    end
    else begin
        internal_counter <= internal_counter + 1;
```

Figure 13

From this result, it can be seen that by using internal counter as the register instead of LSFR is much more accurate and dice-like behaviour, which is more suitable for the user experience and the project objective. Thus, in this version of design internal counter method will be used instead of LSFR as the FSM.

5.0 Simulation Results

5.1 Testbench Logic

The testbench for this project is designed as **test_dice** to verify the functionality of the **internal counter** function in **dice_controller** module using ModelSim Altera for the simulation. It generates a 100MHz clock (Figure 14) and provides inputs for power, reset, and roll signals (Figure 15).

```
initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
```

Figure 14

```
// Initial state
        power  = 0;
        reset  = 0;
        roll = 0;
```

Figure 15

The roll button is held high for 100μs to simulate a rolling action and then released to test the buffering state, to update the dice value several times like how player will be using the dice game controller. This allows verification of both rolling and buffering transitions in the FSM. The timing is control by:

```
// Roll ON
        roll = 1;
        #100_000;

// Roll OFF
        roll = 0;
        #100_000;     // Wait for buffer to finish
```

Figure 16

However, a second version of the module from **dice_controller** need to be edit and named as **dice_controller_simu** to allow the dice value to update faster within the short simulation time, making the output waveform easier to analyse. The modified version uses lower bits of the **internal counter and** also the **BUFFER_LIMIT** is also reduced to speed up the buffering phase in the simulation. The edited part can be seen as:

```
dice_value <= (internal_counter[5:3] % 6) + 1;
```

Figure 17

```
parameter BUFFER_LIMIT = 200;
```

Figure 18

The rolling time from the previous module is also being removed in order for the dice value to be shown on every clock cycle, which help with the simulation purposes.

5.2 Output Waveform

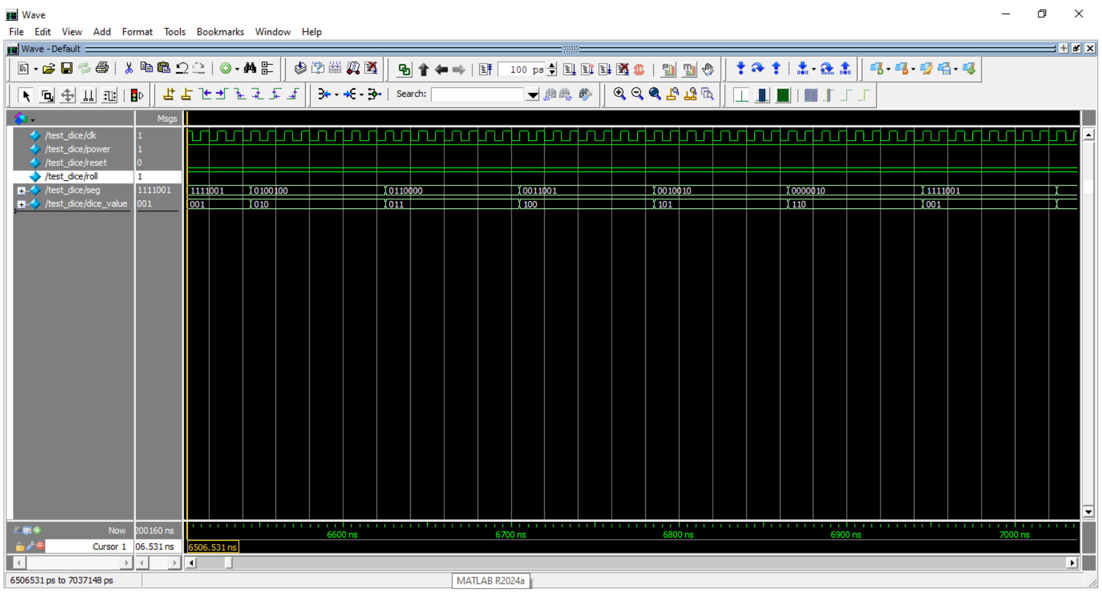


Figure 19

Figure 19 is the simulation output waveform has confirmed correct behaviour of the dice controller module. As can be seen from the graph in ModelSim:

- The top signal (clk) is a 100Mhz clock, generated by the testbench
- The power signal is active high showing the power is turn on, and reset is held low since we do not use this switch yet.
- The roll signal is always high during the rolling phase (100µs), allowing the dice value to update.
- The signal dice_value displays a sequence of valid dice numbers from 1 to 6, represented as a 3-bit binary value

In the waveform:

- At time around 6500 ns to 7000 ns, roll is set as high and dice_value is being update every clock cycle, moving from values 3'b010 (d'2), 3'b011 (d'3), 3'b100 (d'4), 3'b101 (d'5), 3'b110 (d'6), then back to 3'b001 (d'1).
- The 7-Segment output also (seg) updates correctly for each dice_value, showing the expected segment pattern for each dice face.
- The transitions can be seen as smooth and continuous during roll high input, validate the FSM and internal counter are functioning properly.

This confirms that the dice_value generator and FSM control logic are correct, and the 7-Segment display is responding as intended.

6.0 FPGA Implementation

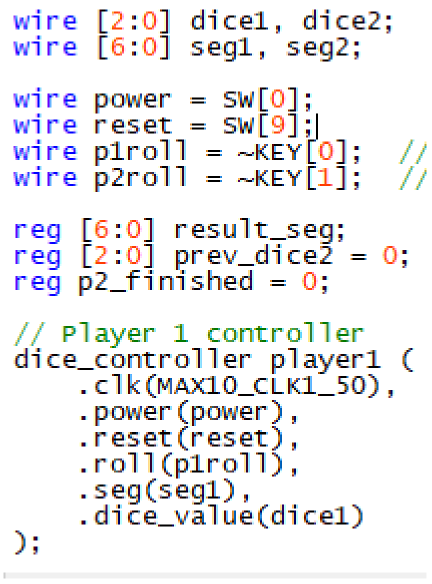
6.1 I/O Assignments

For FPGA hardware implementation on the DE10-Lite board, the pin used for this project has been defined in **test_dice** module with the specific name of the pin, allowing Quartus to automatically assign the corresponding pins based on the DE10-Lite board

Signal	FPGA Pin / Board Mapping
clk	MAX10_CLK1_50 (50 MHz onboard clock)
power	SW[0] (on-board slide switch)
reset	SW[9] (on-board slide switch)
roll	KEY[0] or KEY[1] (pushbutton)
seg	HEX0 or HEX2 (7-segment display)

Table 4

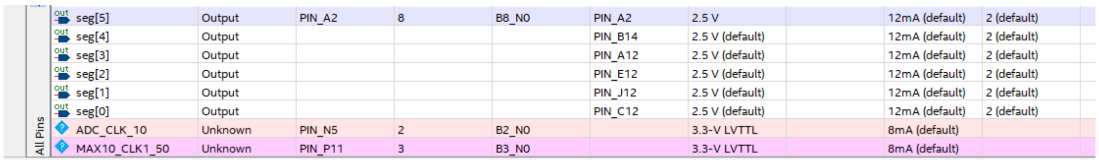
This simplifies the process of pin assignment and ensure correct mapping between the design and the physical FPGA pins. The coding meant can be seen in Figure 20.



```
wire [2:0] dice1, dice2;
wire [6:0] seg1, seg2;

wire power = SW[0];
wire reset = SW[9];
wire p1roll = ~KEY[0];   //
wire p2roll = ~KEY[1];   //

reg [6:0] result_seg;
reg [2:0] prev_dice2 = 0;
reg p2_finished = 0;

// Player 1 controller
dice_controller player1 (
    .clk(MAX10_CLK1_50),
    .power(power),
    .reset(reset),
    .roll(p1roll),
    .seg(seg1),
    .dice_value(dice1)
);
```

Figure 20

The I/O pin assignment also can be verified in the pin planner as in Figure 21. This further confirms that the pin definitions set in the **fpga_dice** module has link necessary pin correctly.



seg[5]	Output	PIN_A2	8	B8_N0	PIN_A2	2.5 V		12mA (default)	2 (default)
seg[4]	Output				PIN_B14	2.5 V (default)		12mA (default)	2 (default)
seg[3]	Output				PIN_A12	2.5 V (default)		12mA (default)	2 (default)
seg[2]	Output				PIN_E12	2.5 V (default)		12mA (default)	2 (default)
seg[1]	Output				PIN_J12	2.5 V (default)		12mA (default)	2 (default)
seg[0]	Output				PIN_C12	2.5 V (default)		12mA (default)	2 (default)
ADC_CLK_10	Unknown	PIN_N5	2	B2_N0		3.3-V LVTTL		8mA (default)	
MAX10_CLK1_50	Unknown	PIN_P11	3	B3_N0		3.3-V LVTTL		8mA (default)	

Figure 21

6.2 RTL Viewer Screenshot

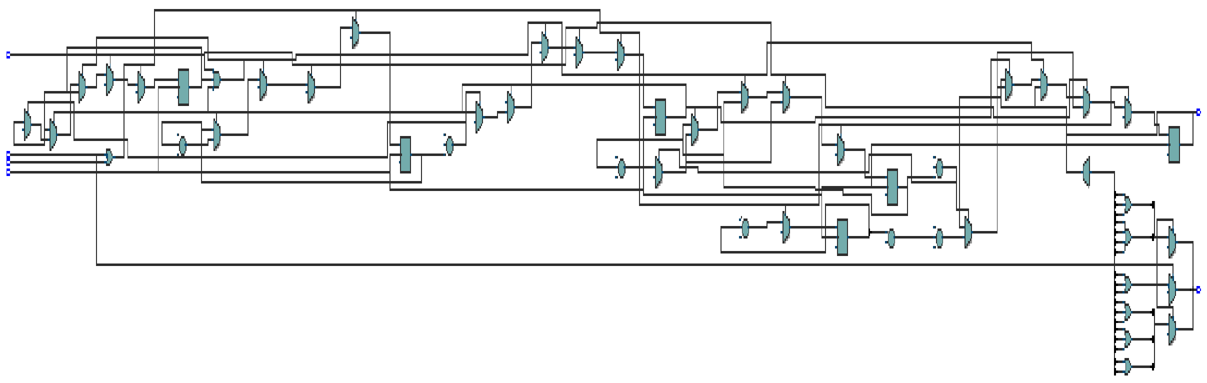


Figure 22

In the RTL diagram it can be seen:

- The design consists mainly of FSM control logic, internal counters, and a simple 7-Segment decoder at the end.
- Registers are used to hold the internal counter, buffer counter, and the dice value
- Combinational logic handles the modulo arithmetic and segment decoding

The RTL diagram shows a relatively simple design, but it consists of many same type components although it is not too complex of a design.

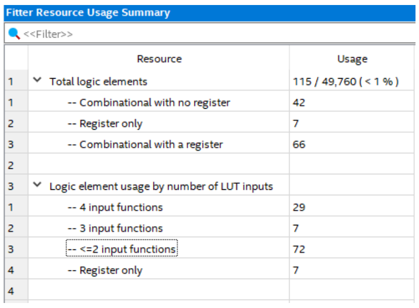
7.0 Performance Evaluation

7.1 Performance Table

Parameter	Value
Logic utilization (%)	< 1 %
Total combinational functions	108
Register count	73
Maximum clock frequency	275.25 MHz
Critical path	3.665 ns

Table 5

The result indicates the dice controller design is very lightweight in terms of logic utilization, making it suitable for a small FPGA such as MAX10 family, the hardware being used for this project.



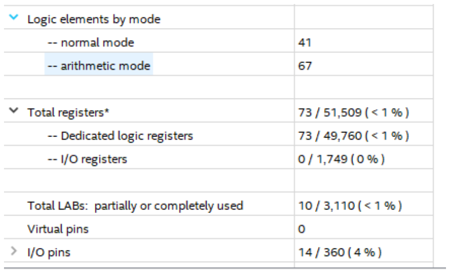
Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Total logic elements	115 / 49,760 (< 1 %)
1	-- Combinational with no register	42
2	-- Register only	7
3	-- Combinational with a register	66
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	29
2	-- 3 input functions	7
3	-- <=2 input functions	72
4	-- Register only	7
4		

Figure 23

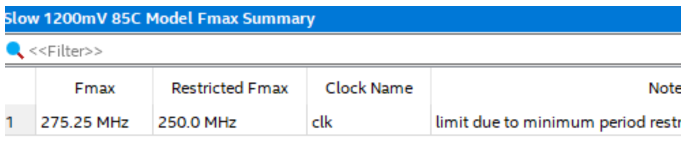
108 combinational functions and 73 dedicated registers are used, saving plenty of resources available for future improvements or additional features.



Logic elements by mode	
-- normal mode	41
-- arithmetic mode	67
Total registers*	73 / 51,509 (< 1 %)
-- Dedicated logic registers	73 / 49,760 (< 1 %)
-- I/O registers	0 / 1,749 (0 %)
Total LABs: partially or completely used	10 / 3,110 (< 1 %)
Virtual pins	0
I/O pins	14 / 360 (4 %)

Figure 24

The timing results show that the design can operate at a maximum frequency of 272.5MHz higher than 50MHz clock used in this project, indicate the design can run safely without timing violations.



Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	275.25 MHz	250.0 MHz	clk	limit due to minimum period restr

Figure 25

The critical path delay of 3.665 ns is very short, confirming that the FSM and combinational logic are simple and efficient.

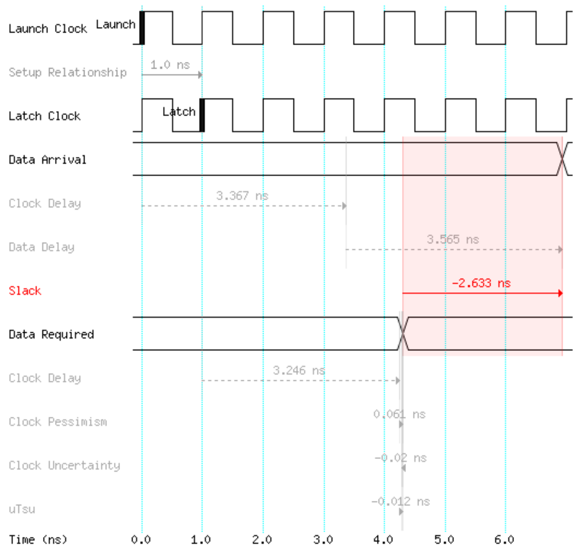


Figure 26

7.2 Performance and Complexity

From the performance table (Table 5) it shows that the **dice_controller** module was designed to be simple and efficient, prioritizing reliable functionality and low resource utilization rather than complex features. The FSM consists of only a few states (idle, rolling, buffering), and the random number generation uses a straightforward internal counter, which keeps the logic small and fast.

One trade-off is that using an internal counter for random number generation is simple, but does not provide true randomness. However, this is acceptable for this project, where the goal is to simulate a user-friendly rolling dice, rather than generate a random number. However, but using the buffer time in the module, it replicates the randomness, since although player stops the number, the buffer will stop it at further sequence giving an almost random dice value

Another trade-off is that the design does not use complex features such as pipelining, advanced arithmetic, or dynamic clock scaling due to the fact that the required performance is easily met with simple combinational logic and counters. This approach results in a very small logic footprint (<1% utilization) and high Fmax (275.25 MHz), far exceeding the system clock, making it a robust and reliable implementation suitable for low-cost FPGA platforms.

Overall, the design achieves a good balance between performance, resource usage, and design simplicity, making it easy to verify, maintain, and scale if additional features are desired in the future.

8.0 Hardware Testing Implementation

The demonstration video of the hardware testing can be found at the link below:

- https://youtu.be/d24OxNyF8ho

In the first part of the video, the FPGA runs the version using LSFR (Linear Feedback Shift Register) as the FSM for random number generation.

In the second part, the improved version using internal counter as the FSM is shown. Additionally, the input method is also changed from a simple switch input to a push button input, improving the user experience and making the dice game feel more interactive.

9.0 Conclusion and Reflection

In summary, this project successfully achieved the objective of designing and implementing a hardware-based dice game controller using Verilog HDL on the DE10-Lite FPGA board.

Through the development process, the system has been improved from an initial LFSR-based design to a finer version of FSM using an internal counter for pseudo-random number generation, which provided more accurate and realistic dice behaviour. The FSM control and buffering mechanism further enhanced the user experience by simulating the rolling and stopping of the dice, similar to a physical game.

Simulation results validated the liability of the FSM and number generation, while the FPGA implementation demonstrated low resource utilization (<1%), high operating frequency (275.25 MHz), and reliable operation on actual hardware.

Throughout the project, our group gained practical experience in:

- Designing finite state machines (FSM)
- Implementing and optimizing random number generators
- Verifying designs using ModelSim simulation
- Managing timing and resource constraints in FPGA
- Testing and debugging hardware implementations

This project also highlighted the trade-offs between design complexity and performance, showing that a simple, well-optimized solution can be very effective for this type of application.

The final hardware demonstration showed that the dice controller performs as intended, providing an interactive and user-friendly experience. This project has helped strengthen my understanding of digital system design, FPGA implementation, and HDL-based development.

10.0 References

- Intel Quartus Prime Software User Guide, available at: https://www.intel.com
- Intel MAX10 FPGA Device Handbook, available at: https://www.intel.com
- ModelSim-Altera User Manual, available at: https://www.intel.com
- Verilog HDL IEEE Standard 1364, freely available summary at: https://standards.ieee.org
- Pong P. Chu, *FPGA Prototyping by Verilog Examples*, 3rd Edition, Wiley (example code and support: https://github.com/chu-fpgabook)
- Enes Çangür, "Linear Feedback Shift Register" [GitHub repository], available at: https://github.com/enescang/linear-feedback-shift-register
- OpenCores Project Repository -Open Source IP Cores, available at: https://opencores.org
- ASIC-World Verilog Tutorials, available at: https://www.asic-world.com/verilog/veritut.html
- Digital System Design Tutorials, OpenFPGA project — https://openfpga.org
- FPGA4Student, "Verilog Microcontroller Code Example", available at: https://www.fpga4student.com/2016/11/verilog-microcontroller-code.html