Digital Logic & Design

Final Project

11th December 2020

Contents

1 Acknowledgement

The following document has been typed using Overleaf; online LaTeX editor.

The rest of this page is intentionally left blank.

2 Introduction & Logistics

Project Title: Connect-Four Video Game on FPGA

Group Members

All names appear in alphabetical order, any other apparent sequence is purely coincidental.

1. Ali Ur Rehman - ar05104
2. Maaz Saeed - ms05050
3. Mohammad Sameer Faisal - mf04709
4. Shoaib Mustafa Ali - sa04275

Project Repository

For all intents and purposes, the documents and codes related to our project can be accessed at our public GitHub Repository, available at:

https://github.com/healyyyyyy/DLD-Final-Project

Final Demo

The video for our final demo, including explanation of both, theory and lab components, and game simulation can be viewed at:

https://www.youtube.com/watch?v=yVnVWqK9gzM&feature=youtu.be

A brief inquiry

Connect-Four, is one of the many names ascribed to the widely popular two-player board game designed by Howard Wexler and Ned Strongin. The game was first published in 1974 and has remained popular among people of all ages ever since.

The objective of the game is to form a line of four chips of your color, whether vertically, horizontally, or diagonally. Players take turns dropping a single chip of their respective color, into the chamber until a victor can be determined, or a stalemate is reached.

3 Design & Research

3.1 Finite State Machines (FSM)

3.1.1 Game

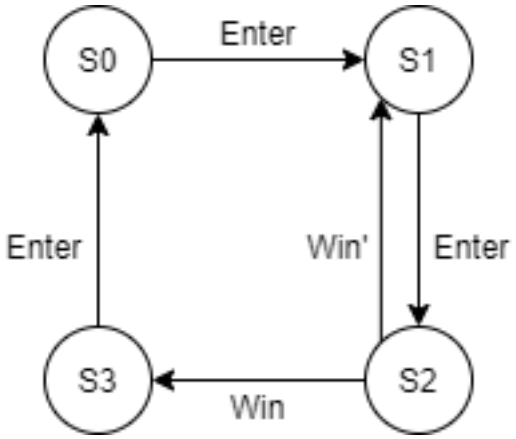
The 'game' finite state machine in our project comprises of the four states S0, S1, S2, & S3, which can be defined as follows:

1. S0 : Default State or Reset State.
   The reset state can only be exited once the user presses the enter button (an input taken from a switch).

2. S1: Column Selection State.
   In this state the player can switch between columns to pick one that he finds suitable to throw his chip in. The user transitions from this state to another only when the enter button is HIGH.

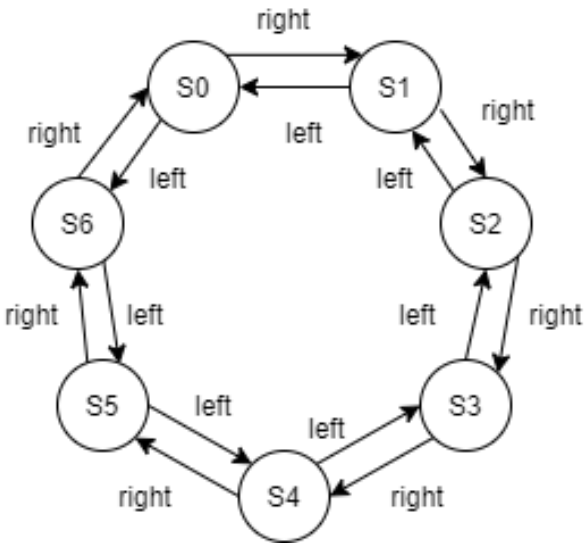
3. S2: Matrix Update and Win Checking State.
   This state updates the matrix so that the new chip is placed in the column selected by the player. It then checks whether the update in the matrix causes any of the win combinations to become HIGH. If it does then we transition to S3. If not then we go back to S1 after switching players.

4. S4: Win State.
   This state shows whether any of the players have one by displaying the winning player's colour on all matrix cells. The players can exit this state by pressing enter again.



3.1.2 Column Selection

The column selection finite state machine is fairly simple. There are seven states, each representing a single column on the game board. By default, at the start of a turn the 0th column is highlighted (and so the machine is at it's 0th state). The player can then either choose to move in either the left or right directions by one column, any number of times until they decide to place their chip. Depending on the direction they choose to move in, the states change. For e.g if you move left from the 1st column, you end up at the 7th column and so the machine moves from state S0 to S6. Similarly moving from column 0 to column 1 would mean going from S0 to S1.

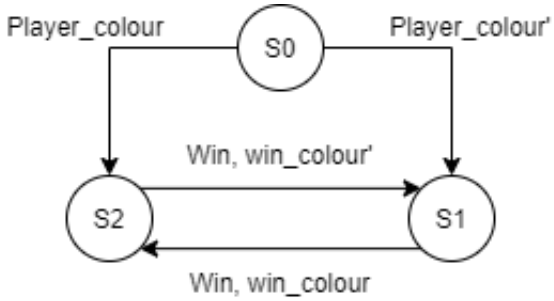


3.1.3 Cell

The game contains 42 cells in total (the matrix is 6x7). Each of these chips can have any of the following states:

1. S0: Inactive State.
   None of the players have placed their chip in this spot
2. S1: Blue State
3. S2: Red State

Due to the presence of three states of any cell, the data has to be represented by 2 bits. The MSB signifies whether or not the cell is active. The LSB signifies the colour of the chip placed in the cell, if active. The state transitions from the inactive state to either of the coloured states depending on the colour of the player who selected the cell. The transition is also only enabled when the column number, cell number and change input (indicating when the matrix has to be updated) is HIGH for this particular cell. The colours will also only change once a win condition has been detected by the win checker which would then promptly enable the colours to switch to display the colour of the player who has won.



3.2 Design Overview

Following is graphical illustration of our machine, along with brief descriptions of each component.

Game FSM

The game finite state machine allows users two options. Either to start a new instance of the game at startup, or to restart the game after a previous round has been won by a player.

Column Choosing FSM

The column choosing FSM takes input from users, to move along the board. Players can move from one column to another until they decide where to deposit their chip.

Player T Flip Flop

The player T Flip Flop enables the changing of the colours representing each player respectively. It determines the colour of the chip being deposited into the matrix.

Matrix

The matrix can be imagined as type of stack. It contains all the deposited chips in a column. By default, all columns are empty and the chip falls the bottom most cell in the selected column, then on-wards if the same column is selected the chip will be deposited one step above the bottom and so on. The matrix is also linked to the win-check.

Win Check

At each chip deposit, the matrix will be checked to see if four chips of the same colour have been placed in a row, column or diagonal. If not, the game continues normally. If yes, the win condition is fulfilled and a victor is declared. The game then moves back to the 'game' finite state machine where we can restart and play another round.

Columns

Inside the matrix, are seven columns, each representing one of seven columns on the game board. Players can choose which column to drop their chip inside of.

Cell Selector

Each column on the game board contains six cells, chips are deposited into these cells starting from the bottom-up, stacked on top of one another. The cell selector manages which cells are already occupied and where the newly deposited chip will fall.

Cells

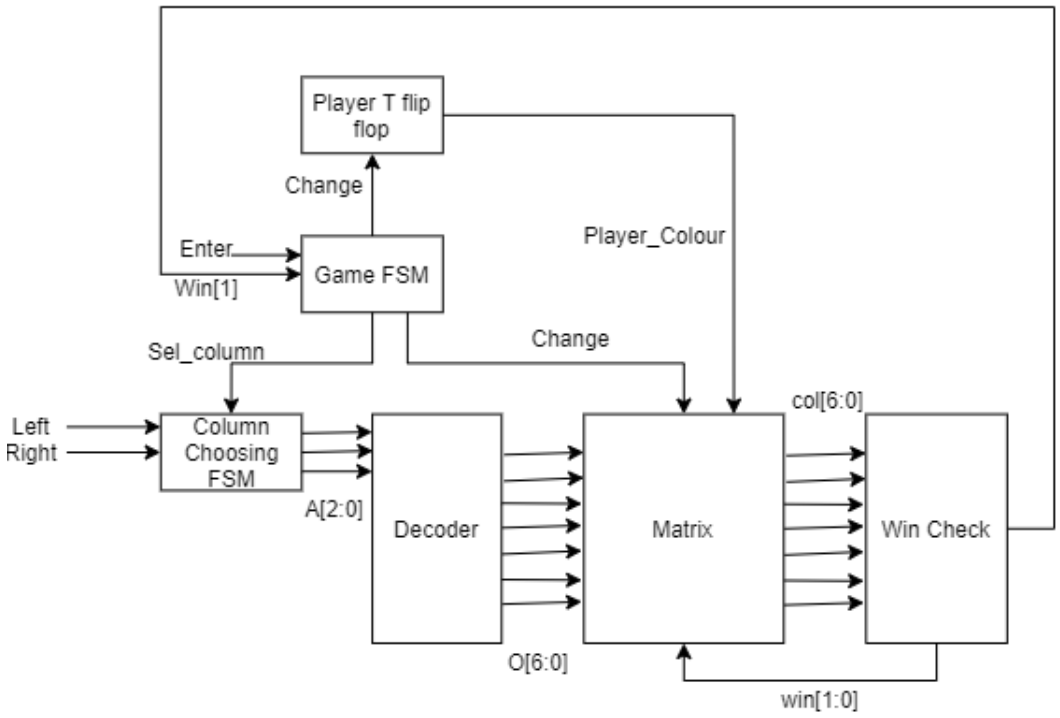
Each cell is able to contain one deposited chip. The cells are connected to the cell selector and enable it to decide whether a chip can be deposited at a certain cell or not (is the cell available or already filled).

Decoders

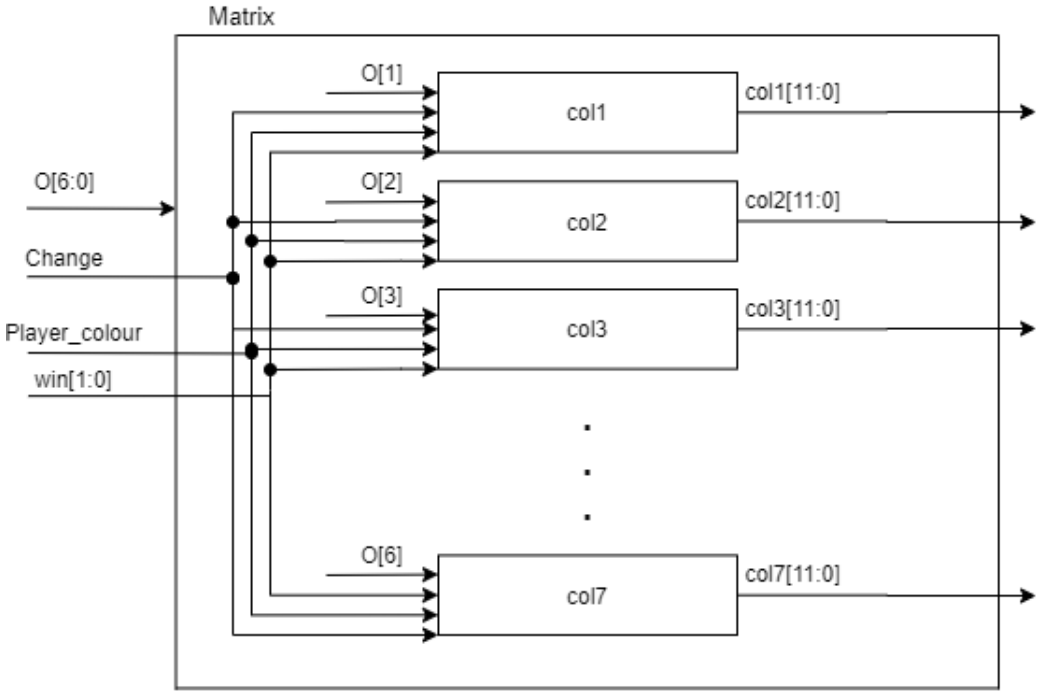
The decoders are used to determine which column as well as which cell has to be activated for the update to occur.

4 Module Top Views

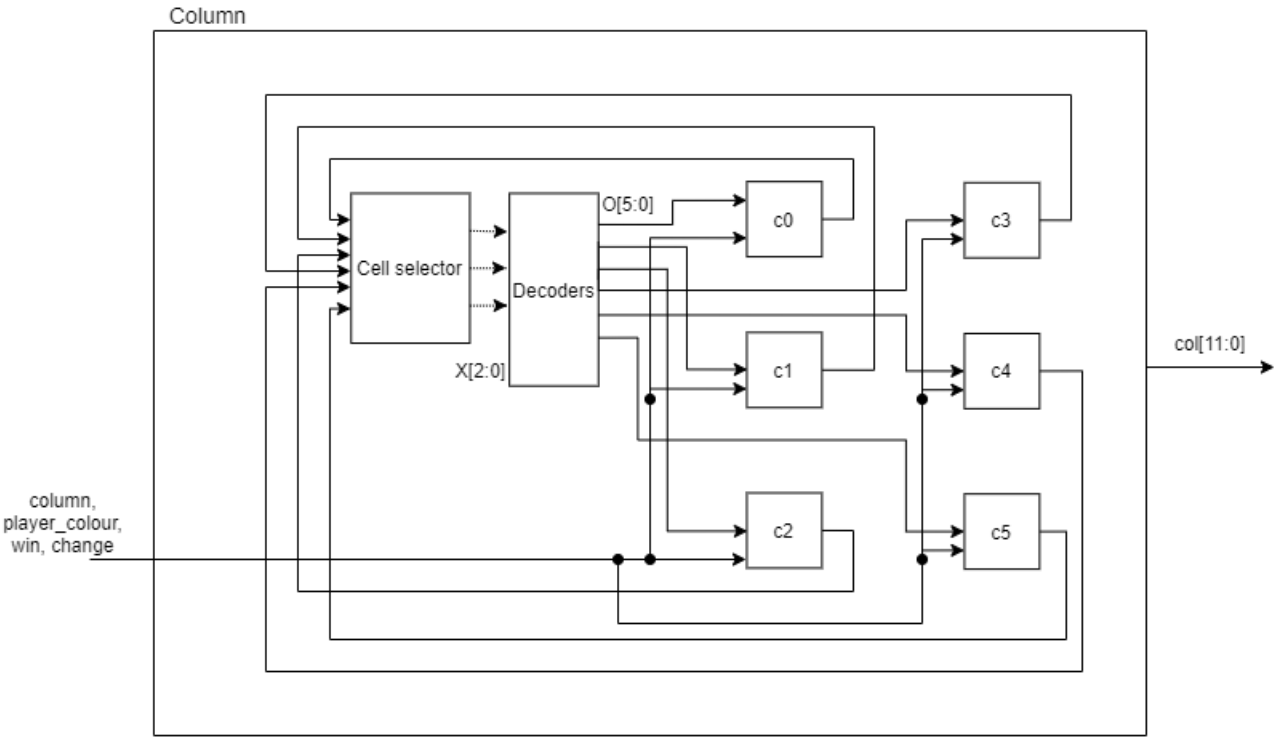
4.1 Game



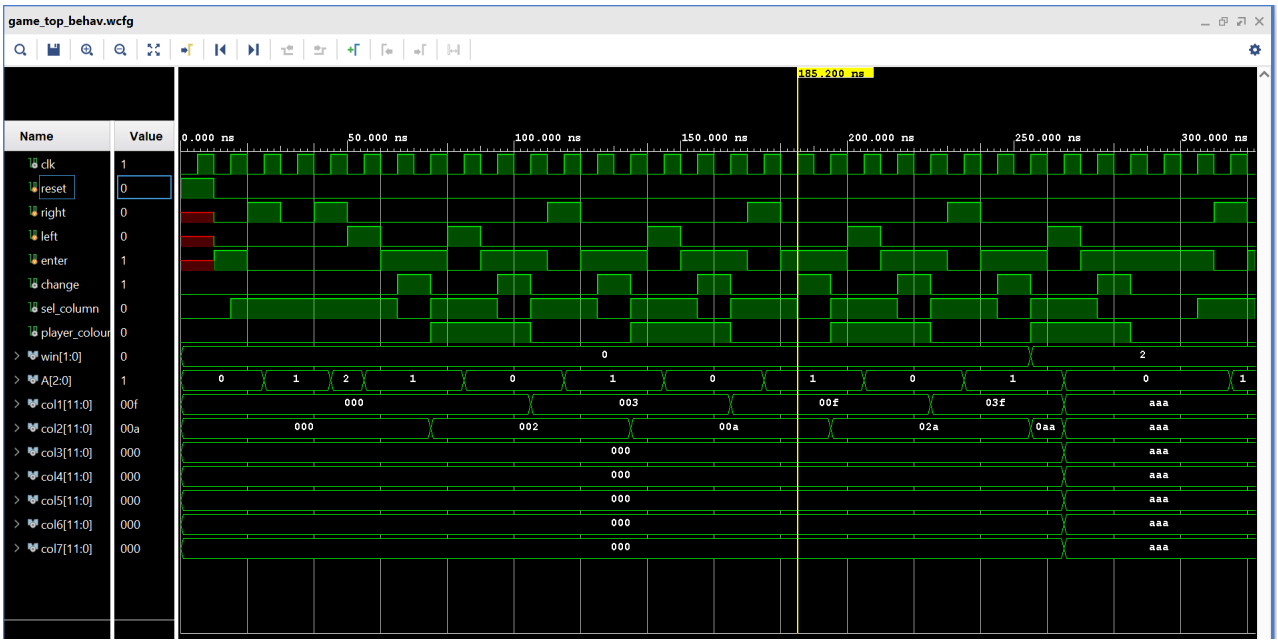
4.2 Matrix



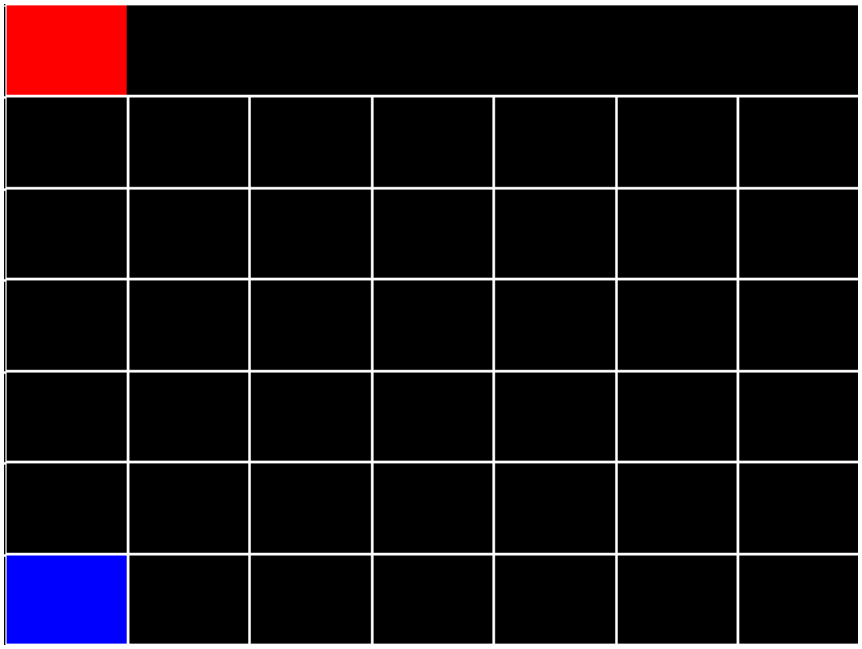
4.3 Column


Column
Cell selector
Decoders
O[5:0]
X[2:0]
c0
c1
c2
c3
c4
c5
column,
player_colour,
win, change
col[11:0]

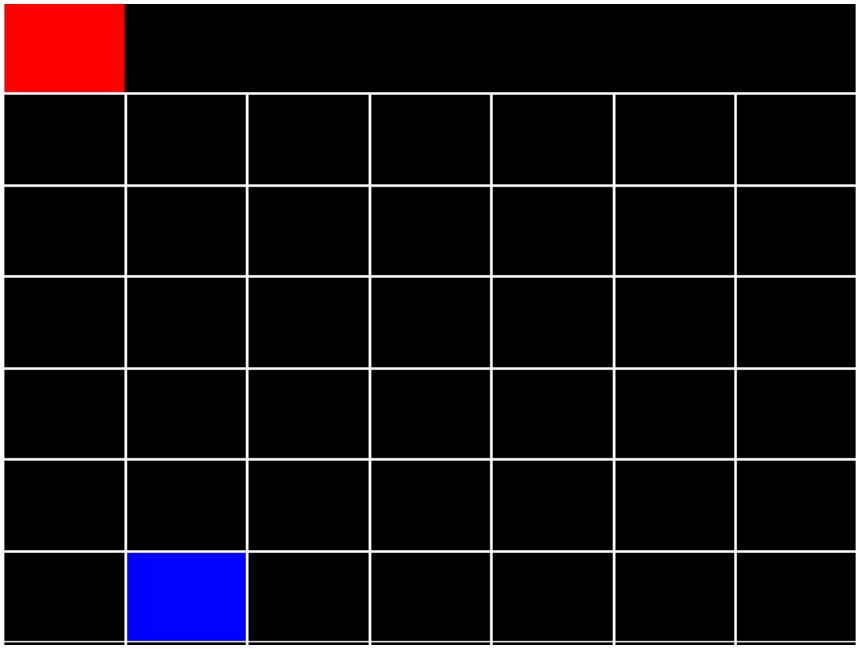
5 Game Simulation



Vivado Simulation: Timing Diagram of the running game.



VGA simulation: Blue player has deposited a chip in the first column, prompting red's turn.



VGA simulation: Blue player has deposited a chip in the second column, prompting red's turn.

6 Appendix: Working for the modules

6.1 Column Selection FSM

6.1.1 Truth Table

	Present State			Inputs		Next State			T FlipFlop Inputs		
	A2	A1	A0	Right	Left	A2(t+1)	A1(t+1)	A0(t+1)	T2	T1	T0
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1	1	1	1
2	0	0	0	1	0	0	0	1	0	0	1
3	0	0	0	1	1	0	0	0	0	0	0
4	0	0	1	0	0	0	0	1	0	0	0
5	0	0	1	0	1	0	0	0	0	0	1
6	0	0	1	1	0	0	1	0	0	1	1
7	0	0	1	1	1	0	0	1	0	0	0
8	0	1	0	0	0	0	1	0	0	0	0
9	0	1	0	0	1	0	0	1	0	1	1
10	0	1	0	1	0	0	1	1	0	0	1
11	0	1	0	1	1	0	1	0	0	0	0
12	0	1	1	0	0	0	1	1	0	0	0
13	0	1	1	0	1	0	1	0	0	0	1
14	0	1	1	1	0	1	0	0	1	1	1
15	0	1	1	1	1	0	1	1	0	0	0
16	1	0	0	0	0	1	0	0	0	0	0
17	1	0	0	0	1	0	1	1	1	1	1
18	1	0	0	1	0	1	0	1	0	0	1
19	1	0	0	1	1	1	0	0	0	0	0
20	1	0	1	0	0	1	0	1	0	0	0
21	1	0	1	0	1	1	0	0	0	0	1
22	1	0	1	1	0	1	1	0	0	1	1
23	1	0	1	1	1	1	0	1	0	0	0
24	1	1	0	0	0	1	1	0	0	0	0
25	1	1	0	0	1	1	0	1	0	1	1
26	1	1	0	1	0	1	1	1	0	0	1
27	1	1	0	1	1	1	1	0	0	0	0
28	1	1	1	0	0	x	x	x	x	x	x
29	1	1	1	0	1	x	x	x	x	x	x
30	1	1	1	1	0	x	x	x	x	x	x
31	1	1	1	1	1	x	x	x	x	x	x

6.1.2 State Equations

$$T_2 = A_1' \cdot A_0' \cdot Right' \cdot Left + A_1 \cdot A_0 \cdot Right \cdot Left'$$

$$T_1 = A_0' \cdot Right' \cdot Left + A_0 \cdot Right \cdot Left'$$

$$T_0 = Right' \cdot Left + Right \cdot Left'$$

6.2 Game State Machine

6.2.1 Truth Table

	Present State		Inputs		Next State		Outputs	
	A1	A0	enter	win	A1(t+1)	A0(t+1)	Change	Sel_column
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0
2	0	0	1	0	0	1	0	0
3	0	0	1	1	0	0	0	0
4	0	1	0	0	0	1	0	1
5	0	1	0	1	0	1	0	1
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	0	1
8	1	0	0	0	0	1	1	0
9	1	0	0	1	1	1	1	0
10	1	0	1	0	0	1	1	0
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	1	0	0
13	1	1	0	1	1	1	0	0
14	1	1	1	0	0	0	0	0
15	1	1	1	1	0	0	0	0

6.2.2 State Equations for D Flip-Flop inputs

$$D_1 = A_2' \cdot A_1 \cdot enter + A_2 \cdot A_1' \cdot win + A_2 \cdot A_1 \cdot enter'$$

$$D_0 = A_1 \cdot enter' + A_2 \cdot A_1' + A_1' \cdot enter \cdot win'$$

6.3 Cell FSM

6.3.1 Truth Table

	Present State		Inputs			Next State	
	A1	A0	player_colour	win	win_colour	A1(t+1)	A0(t+1)
0	0	0	0	0	0	1	0
1	0	0	0	0	1	1	0
2	0	0	0	1	0	1	0
3	0	0	0	1	1	1	1
4	0	0	1	0	0	1	1
5	0	0	1	0	1	1	1
6	0	0	1	1	0	1	0
7	0	0	1	1	1	1	1
8	0	1	0	0	0	1	0
9	0	1	0	0	1	1	0
10	0	1	0	1	0	1	0
11	0	1	0	1	1	1	1
12	0	1	1	0	0	1	1
13	0	1	1	0	1	1	1
14	0	1	1	1	0	1	0
15	0	1	1	1	1	1	1
16	1	0	0	0	0	1	0
17	1	0	0	0	1	1	0
18	1	0	0	1	0	1	0
19	1	0	0	1	1	1	1
20	1	0	1	0	0	1	1
21	1	0	1	0	1	1	1
22	1	0	1	1	0	1	0
23	1	0	1	1	1	1	1
24	1	1	0	0	0	1	0
25	1	1	0	0	1	1	0
26	1	1	0	1	0	1	0
27	1	1	0	1	1	1	1
28	1	1	1	0	0	1	1
29	1	1	1	0	1	1	1
30	1	1	1	1	0	1	0
31	1	1	1	1	1	1	1

6.3.2 State Equations for D flip flop

$$D_1 = column \cdot cell \cdot change$$

$$D_0 = win \cdot win_colour + D_0 \cdot win' + D_1' \cdot player_colour \cdot win'$$

6.4 Cell Selector

6.4.1 Truth Table

Inputs						Outputs		
C6	C5	C4	C3	C2	C1	F2	F1	F0
0	0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	1	0
0	0	0	0	1	1	0	1	1
0	0	0	1	1	1	1	0	0
0	0	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	0
1	1	1	1	1	1	0	0	0

6.4.2 Equations

$$F_2 = C_6' \cdot C_3$$

$$F_1 = C_3' \cdot C_1 + C_6' \cdot C_5$$

$$F_0 = C_3' \cdot C_2 + C_5' \cdot C_4 + C_1'$$

7 Appendix: Verilog Code for Modules

Note: All the modules run on timescale 1ns/1ps.

7.1 Game.v

The game module consists of three main components; **T_FF** which is representative of the **current player**, the **game_FSM** which is the main game finite state machine as discussed earlier, and the **update_and_check_win**. The **update_and_check_win** component is responsible for updating and checking the win conditions for all matrix cells.
This is a top level module for the game operation.

```
module game(
    input clk,
    input reset,
    input right,
    input left,
    input enter,
    output [11:0] col1,
    output [11:0] col2,
    output [11:0] col3,
    output [11:0] col4,
    output [11:0] col5,
    output [11:0] col6,
    output [11:0] col7,
    output [2:0] A,
    output player_colour,
    output change,
    output sel_column,
    output [1:0] win
    );

    T_FF player(change, clk, reset, change, player_colour);
    game_FSM mainfsm(clk, reset, enter, win, change, sel_column);
    update_and_check_win checker(player_colour, change, sel_column, right, left, clk,
        reset, win, col1, col2, col3, col4, col5, col6, col7, A);

endmodule
```

7.2 game_FSM.v

The game FSM is a moore FSM which contains two D flip flops which maintain the state of the game FSM. The wires **d1** and **d2** are inputs for the flip flops for this FSM. The FSM then outputs **change** and **sel_column** depending on the current state of the FSM.

```
module game_FSM(
    input clk,
    input reset,
    input enter,
    input [1:0] win,
    output change,
    output sel_column
    );
    wire q2;
    wire q1;
    wire d2;
    wire d1;
    wire won;
    assign won = win[1];
    assign d1 = (~q1 & enter) | (q1 & ~enter) | (q2 & ~q1);
    assign d2 = (~q2 & q1 & enter) | (q2 & ~q1 & won) | (q2 & q1 & ~enter);
    D_ff ff1(d1, clk, reset, q1);
    D_ff ff2(d2, clk, reset, q2);

    assign change = q2 & ~q1;
    assign sel_column = ~q2 & q1;
endmodule
```

7.3 update_and_check_win.v

The **update_and_check_win** is a top level component which contains multiple modules within itself; **column_choosing_FSM**, **decoder**, **matrix**, **win_check**. The column choosing FSM selects an column for the current player while the **sel_column** input is asserted. Once it is de-asserted and **change** is asserted, the output of **column_choosing_FSM** is then taken as an input for the **decoder**. The **decoder** output then goes into the **matrix** and the output of **matrix** is taken as an input for the **win_check** system.

```
module update_and_check_win(
    input player_colour,
    input change,
    input sel_column,
    input right,
    input left,
    input clk,
    input reset,
    output [1:0] win,
    output [11:0] col1,
    output [11:0] col2,
    output [11:0] col3,
    output [11:0] col4,
    output [11:0] col5,
    output [11:0] col6,
    output [11:0] col7,
    output [2:0] A
    );
    wire [6:0] O;
    column_choosing_FSM FSM(right, left, reset, clk, sel_column, A);
    decoder d2(A, O);
    matrix m(O, player_colour, change, clk, reset, win, col1, col2, col3, col4, col5,
        col6, col7);
    win_check wc(col1, col2, col3, col4, col5, col6, col7, clk, win);
endmodule
```

7.4 column_choosing_FSM.v

The **column_choosing_FSM** selects a column for the current player to insert his chip in. If the player enters left, the column selector decrements and if they enter right then the column selector increments. This is only done when the **sel_column** input is asserted which acts as an enable for the T flip-flops in this FSM. The output is a 3 bit value which then goes into the decoder.

```
module column_choosing_FSM(
    input right,
    input left,
    input reset,
    input clk,
    input sel_column,
    output [2:0] A
    );
    wire t0;
    wire t1;
    wire t2;
    assign t0 = (~A[2] & right & ~left) | (~A[1] & right & ~left) |
        (A[0] & ~right & left) | (A[1] & ~right & left) | (A[2] & ~right & left);
    assign t1 = (~A[0] & ~right & left) | (A[0] & right & ~left) |
        (A[2] & A[1] & right & ~left);
    assign t2 = (~A[1] & ~A[0] & ~right & left) | (A[1] & A[0] & right & ~left) |
        (A[2] & A[1] & right & ~left);
    T_FF tff0(t0, clk, reset, sel_column, A[0]);
    T_FF tff1(t1, clk, reset, sel_column, A[1]);
    T_FF tff2(t2, clk, reset, sel_column, A[2]);
endmodule
```

7.5 decoder.v

```
module decoder(
    input [2:0] F,
    output [6:0] O
    );
    assign O[0] = ~F[2] & ~F[1] & ~F[0];
    assign O[1] = ~F[2] & ~F[1] & F[0];
    assign O[2] = ~F[2] & F[1] & ~F[0];
    assign O[3] = ~F[2] & F[1] & F[0];
    assign O[4] = F[2] & ~F[1] & ~F[0];
    assign O[5] = F[2] & ~F[1] & F[0];
    assign O[6] = F[2] & F[1] & ~F[0];
endmodule
```

7.6 matrix.v

The **matrix** contains 7 different columns which are 12 bit buses. This component acts as a top level module for the columns.

```
module matrix(
    input [6:0] O,
    input player_colour,
    input change,
    input clk,
    input reset,
    input [1:0] win,
    output [11:0] col1,
    output [11:0] col2,
    output [11:0] col3,
    output [11:0] col4,
    output [11:0] col5,
    output [11:0] col6,
    output [11:0] col7
    );
    column c1(O[0], player_colour, change, win, clk, reset, col1);
    column c2(O[1], player_colour, change, win, clk, reset, col2);
    column c3(O[2], player_colour, change, win, clk, reset, col3);
    column c4(O[3], player_colour, change, win, clk, reset, col4);
    column c5(O[4], player_colour, change, win, clk, reset, col5);
    column c6(O[5], player_colour, change, win, clk, reset, col6);
    column c7(O[6], player_colour, change, win, clk, reset, col7);
endmodule
```

7.7 column.v

The **column** is responsible for the changes made within a particular column of the matrix. Each cell in the column outputs a 2 bit value, the first bit is responsible for determining whether the cell is active or not and the second bit represents the colour of the cell if active. This is determined through the **cell_fsm**s for each of the 6 cells in the column. **Cell_occupied** represents whether each of the cell in the column is occupied or not. Depending on the number of cells occupied, the **cell_selector** then selects the appropriate

position for the incoming chip. The output of the **cell_selector** acts as a input for the decoder. This decoder then decodes the value to then enable the respective cell which contains other inputs as well.

```
module column(
    input column_,
    input player_colour,
    input change,
    input [1:0] win,
    input clk,
    input reset,
    output [11:0] column_vector
    );
    wire [2:0] X;
    wire [6:0] O;
    wire [5:0] cell_occupied;

    assign cell_occupied[5] = column_vector[11];
    assign cell_occupied[4] = column_vector[9];
    assign cell_occupied[3] = column_vector[7];
    assign cell_occupied[2] = column_vector[5];
    assign cell_occupied[1] = column_vector[3];
    assign cell_occupied[0] = column_vector[1];

    cell_fsm c0(column_, O[1], change, player_colour, clk, reset, win,
        column_vector[1:0]);
    cell_fsm c1(column_, O[2], change, player_colour, clk, reset, win,
        column_vector[3:2]);
    cell_fsm c2(column_, O[3], change, player_colour, clk, reset, win,
        column_vector[5:4]);
    cell_fsm c3(column_, O[4], change, player_colour, clk, reset, win,
        column_vector[7:6]);
    cell_fsm c4(column_, O[5], change, player_colour, clk, reset, win,
        column_vector[9:8]);
    cell_fsm c5(column_, O[6], change, player_colour, clk, reset, win,
        column_vector[11:10]);

    cell_selector sel(
        cell_occupied[0],
        cell_occupied[1],
        cell_occupied[2],
        cell_occupied[3],
        cell_occupied[4],
        cell_occupied[5],
        X);
    decoder d1(X, O);
endmodule
```

7.8 cell_fsm.v

The cell FSM works through two D flip-flops. The first D flip-flop determines whether or not the cell is active and the second determines the colour if active.

```
module cell_fsm(
    input column,
    input cell_,
    input change,
    input player_colour,
    input clk,
    input reset,
    input [1:0] win,
    output [1:0] cell_colour
    );
    wire col_cel_change;
    assign col_cel_change = column & cell_ & change;
    wire D2;
    wire D1;
    assign D2 = col_cel_change | cell_colour[1] | win[1];
    D_ff ff2(D2, clk, reset, cell_colour[1]);
    D_ff ff1(D1, clk, reset, cell_colour[0]);
    assign D1 = (win[1] & win[0]) |
        (cell_colour[0] & ~win[1]) |
        (~cell_colour[1] & player_colour & col_cel_change & ~win[1]);
endmodule
```

7.9 cell_selector.v

The cell selector selects the appropriate position of the next chip in the column. If all the positions are occupied, and the player still selects this column, no change is made to the column and the next player gets their turn.

```
module cell_selector(
    input [1:0] F,
    input [1:0] E,
    input [1:0] D,
    input [1:0] C,
    input [1:0] B,
    input [1:0] A,
    output [2:0] X
    );
    assign X[0] = ~F | (~D & E) | (~B & C);
    assign X[1] = (~D & F) | (~A &  B);
    assign X[2] = ~A & D;
endmodule
```

7.10 win_check.v

Win check contains all the different 69 combinations of how the win can be generated. If any of the 69 combinations return a HIGH value, an overall win is generated.

```
module win_check(
    input [11:0] col1,
    input [11:0] col2,
    input [11:0] col3,
    input [11:0] col4,
    input [11:0] col5,
    input [11:0] col6,
    input [11:0] col7,
    input clk,
    output [1:0]win
    );
    wire [1:0] win_c [68:0];
    reg [1:0] m_ [41:0];
    always @ (*) begin
        m_[0] = col1[1:0];
        m_[1] = col1[3:2];
        m_[2] = col1[5:4];
        m_[3] = col1[7:6];
        m_[4] = col1[9:8];
        m_[5] = col1[11:10];
        m_[6] = col2[1:0];
        m_[7] = col2[3:2];
        m_[8] = col2[5:4];
        m_[9] = col2[7:6];
        m_[10] = col2[9:8];
        m_[11] = col2[11:10];
        m_[12] = col3[1:0];
        m_[13] = col3[3:2];
        m_[14] = col3[5:4];
        m_[15] = col3[7:6];
        m_[16] = col3[9:8];
        m_[17] = col3[11:10];
        m_[18] = col4[1:0];
        m_[19] = col4[3:2];
        m_[20] = col4[5:4];
        m_[21] = col4[7:6];
        m_[22] = col4[9:8];
        m_[23] = col4[11:10];
        m_[24] = col5[1:0];
        m_[25] = col5[3:2];
        m_[26] = col5[5:4];
        m_[27] = col5[7:6];
        m_[28] = col5[9:8];
        m_[29] = col5[11:10];
        m_[30] = col6[1:0];
        m_[31] = col6[3:2];
        m_[32] = col6[5:4];
        m_[33] = col6[7:6];
        m_[34] = col6[9:8];
```

```
        m_[35] = col6[11:10];
        m_[36] = col7[1:0];
        m_[37] = col7[3:2];
        m_[38] = col7[5:4];
        m_[39] = col7[7:6];
        m_[40] = col7[9:8];
        m_[41] = col7[11:10];
    end
    //check columns
    cell_check m0(m_[0], m_[1], m_[2], m_[3], win_c[0]);
    cell_check m1(m_[1], m_[2], m_[3], m_[4], win_c[1]);
    cell_check m2(m_[2], m_[3], m_[4], m_[5], win_c[2]);
    cell_check m3(m_[6], m_[7], m_[8], m_[9], win_c[3]);
    cell_check m4(m_[7], m_[8], m_[9], m_[10], win_c[4]);
    cell_check m5(m_[8], m_[9], m_[10], m_[11], win_c[5]);
    cell_check m6(m_[12], m_[13], m_[14], m_[15], win_c[6]);
    cell_check m7(m_[13], m_[14], m_[15], m_[16], win_c[7]);
    cell_check m8(m_[14], m_[15], m_[16], m_[17], win_c[8]);
    cell_check m9(m_[18], m_[19], m_[20], m_[21], win_c[9]);
    cell_check m10(m_[19], m_[20], m_[21], m_[22], win_c[10]);
    cell_check m11(m_[20], m_[21], m_[22], m_[23], win_c[11]);
    cell_check m12(m_[24], m_[25], m_[26], m_[27], win_c[12]);
    cell_check m13(m_[25], m_[26], m_[27], m_[28], win_c[13]);
    cell_check m14(m_[26], m_[27], m_[28], m_[29], win_c[14]);
    cell_check m15(m_[30], m_[31], m_[32], m_[33], win_c[15]);
    cell_check m16(m_[31], m_[32], m_[33], m_[34], win_c[16]);
    cell_check m17(m_[32], m_[33], m_[34], m_[35], win_c[17]);
    cell_check m18(m_[36], m_[37], m_[38], m_[39], win_c[18]);
    cell_check m19(m_[37], m_[38], m_[39], m_[40], win_c[19]);
    cell_check m20(m_[38], m_[39], m_[40], m_[41], win_c[20]);

    //check rows
    cell_check m21(m_[0], m_[6], m_[12], m_[18], win_c[21]);
    cell_check m22(m_[6], m_[12], m_[18], m_[24], win_c[22]);
    cell_check m23(m_[12], m_[18], m_[24], m_[30], win_c[23]);
    cell_check m24(m_[18], m_[24], m_[30], m_[36], win_c[24]);//
    cell_check m25(m_[1], m_[7], m_[13], m_[19], win_c[25]);
    cell_check m26(m_[7], m_[13], m_[19], m_[25], win_c[26]);
    cell_check m27(m_[13], m_[19], m_[25], m_[31], win_c[27]);
    cell_check m28(m_[19], m_[25], m_[31], m_[37], win_c[28]);//
    cell_check m29(m_[2], m_[8], m_[14], m_[20], win_c[29]);
    cell_check m30(m_[8], m_[14], m_[20], m_[26], win_c[30]);
    cell_check m31(m_[14], m_[20], m_[26], m_[32], win_c[31]);
    cell_check m32(m_[20], m_[26], m_[23], m_[38], win_c[32]);//
    cell_check m33(m_[3], m_[9], m_[15], m_[21], win_c[33]);
    cell_check m34(m_[9], m_[15], m_[21], m_[27], win_c[34]);
    cell_check m35(m_[15], m_[21], m_[27], m_[33], win_c[35]);
    cell_check m36(m_[21], m_[27], m_[33], m_[39], win_c[36]);//
    cell_check m37(m_[4], m_[10], m_[16], m_[22], win_c[37]);
    cell_check m38(m_[10], m_[16], m_[22], m_[28], win_c[38]);
    cell_check m39(m_[16], m_[22], m_[28], m_[34], win_c[39]);
    cell_check m40(m_[22], m_[28], m_[34], m_[40], win_c[40]);//
```

```
    cell_check m41(m_[5], m_[11], m_[17], m_[23], win_c[41]);
    cell_check m42(m_[11], m_[17], m_[23], m_[29], win_c[42]);
    cell_check m43(m_[17], m_[23], m_[29], m_[35], win_c[43]);
    cell_check m44(m_[23], m_[29], m_[35], m_[41], win_c[44]);

    //check diagnols
    cell_check m45(m_[2], m_[9], m_[16], m_[23], win_c[45]);
    cell_check m46(m_[1], m_[8], m_[15], m_[22], win_c[46]);
    cell_check m47(m_[8], m_[15], m_[22], m_[29], win_c[47]);
    cell_check m48(m_[0], m_[7], m_[14], m_[21], win_c[48]);
    cell_check m49(m_[7], m_[14], m_[21], m_[28], win_c[49]);
    cell_check m50(m_[14], m_[21], m_[28], m_[35], win_c[50]);
    cell_check m51(m_[6], m_[13], m_[20], m_[27], win_c[51]);
    cell_check m52(m_[13], m_[20], m_[27], m_[34], win_c[52]);
    cell_check m53(m_[20], m_[27], m_[34], m_[41], win_c[53]);
    cell_check m54(m_[12], m_[19], m_[26], m_[33], win_c[54]);
    cell_check m55(m_[19], m_[26], m_[33], m_[40], win_c[55]);
    cell_check m56(m_[18], m_[25], m_[32], m_[39], win_c[56]);//
    cell_check m57(m_[3], m_[8], m_[13], m_[18], win_c[57]);
    cell_check m58(m_[4], m_[9], m_[14], m_[19], win_c[58]);
    cell_check m59(m_[9], m_[14], m_[19], m_[24], win_c[59]);
    cell_check m60(m_[5], m_[10], m_[15], m_[20], win_c[60]);
    cell_check m61(m_[10], m_[15], m_[20], m_[25], win_c[61]);
    cell_check m62(m_[15], m_[20], m_[25], m_[30], win_c[62]);
    cell_check m63(m_[11], m_[16], m_[21], m_[26], win_c[63]);
    cell_check m64(m_[16], m_[21], m_[26], m_[31], win_c[64]);
    cell_check m65(m_[21], m_[26], m_[31], m_[36], win_c[65]);
    cell_check m66(m_[17], m_[22], m_[27], m_[32], win_c[66]);
    cell_check m67(m_[22], m_[27], m_[32], m_[37], win_c[67]);
    cell_check m68(m_[23], m_[28], m_[33], m_[38], win_c[68]);

    assign win[0] = win_c[0][0] || win_c[1][0] ||
        win_c[2][0] || win_c[3][0] || win_c[4][0] ||
        win_c[5][0] || win_c[6][0] || win_c[7][0] ||
        win_c[8][0] || win_c[9][0] || win_c[10][0] ||
        win_c[11][0] || win_c[12][0] || win_c[13][0] ||
        win_c[14][0] || win_c[15][0] || win_c[16][0] ||
        win_c[17][0] || win_c[18][0] || win_c[19][0] ||
        win_c[20][0] || win_c[21][0] || win_c[22][0] ||
        win_c[23][0] || win_c[24][0] || win_c[25][0] ||
        win_c[26][0] || win_c[27][0] || win_c[28][0] ||
        win_c[29][0] || win_c[30][0] || win_c[31][0] ||
        win_c[32][0] || win_c[33][0] || win_c[34][0] ||
        win_c[35][0] || win_c[36][0] || win_c[37][0] ||
        win_c[38][0] || win_c[39][0] || win_c[40][0] ||
        win_c[41][0] || win_c[42][0] || win_c[43][0] ||
        win_c[44][0] || win_c[45][0] || win_c[46][0] ||
        win_c[47][0] || win_c[48][0] || win_c[49][0] ||
        win_c[50][0] || win_c[51][0] || win_c[52][0] ||
        win_c[53][0] || win_c[54][0] || win_c[55][0] ||
        win_c[56][0] || win_c[57][0] || win_c[58][0] ||
        win_c[59][0] || win_c[60][0] || win_c[61][0] ||
```

```
        win_c[62][0] || win_c[63][0] || win_c[64][0] ||
        win_c[65][0] || win_c[66][0] || win_c[67][0] ||
        win_c[68][0];

    assign win[1] = win_c[0][1] || win_c[1][1] ||
        win_c[2][1] || win_c[3][1] || win_c[4][1] ||
        win_c[5][1] || win_c[6][1] || win_c[7][1] ||
        win_c[8][1] || win_c[9][1] || win_c[10][1] ||
        win_c[11][1] || win_c[12][1] || win_c[13][1] ||
        win_c[14][1] || win_c[15][1] || win_c[16][1] ||
        win_c[17][1] || win_c[18][1] || win_c[19][1] ||
        win_c[20][1] || win_c[21][1] || win_c[22][1] ||
        win_c[23][1] || win_c[24][1] || win_c[25][1] ||
        win_c[26][1] || win_c[27][1] || win_c[28][1] ||
        win_c[29][1] || win_c[30][1] || win_c[31][1] ||
        win_c[32][1] || win_c[33][1] || win_c[34][1] ||
        win_c[35][1] || win_c[36][1] || win_c[37][1] ||
        win_c[38][1] || win_c[39][1] || win_c[40][1] ||
        win_c[41][1] || win_c[42][1] || win_c[43][1] ||
        win_c[44][1] || win_c[45][1] || win_c[46][1] ||
        win_c[47][1] || win_c[48][1] || win_c[49][1] ||
        win_c[50][1] || win_c[51][1] || win_c[52][1] ||
        win_c[53][1] || win_c[54][1] || win_c[55][1] ||
        win_c[56][1] || win_c[57][1] || win_c[58][1] ||
        win_c[59][1] || win_c[60][1] || win_c[61][1] ||
        win_c[62][1] || win_c[63][1] || win_c[64][1] ||
        win_c[65][1] || win_c[66][1] || win_c[67][1] ||
        win_c[68][1];
endmodule
```

7.11 cell_check.v

The cell check takes in 4 different cells and checks whether the cell is occupied **and** all the occupied cells have the same colour. If they do then a win is generated for that particular colour. If not then a no win output [00] is generated.

```
module cell_check(
    input [1:0] c1,
    input [1:0] c2,
    input [1:0] c3,
    input [1:0] c4,
    output [1:0] win
    );
    wire blue_win;
    wire red_win;
    reg [1:0] win_;
    and g1(red_win, c1[1], c2[1], c3[1], c4[1], c1[0], c2[0], c3[0], c4[0]);
    and g2(blue_win, c1[1], c2[1], c3[1], c4[1], ~c1[0], ~c2[0], ~c3[0], ~c4[0]);
    always @ (*) begin
        if (blue_win) begin
            win_ = 2'b10;
        end
        else if (red_win) begin
            win_ = 2'b11;
        end
        else begin
            win_ = 2'b00;
        end
    end
    assign win = win_;
endmodule
```

7.12 D_ff.v

```
module D_ff(
    input D,
    input clk,
    input reset,
    output reg Q
    );
    always @ (posedge clk or posedge reset) begin
     if(reset==1'b1)begin
        Q = 1'b0;
     end
     else begin
       Q = D;
     end
    end
endmodule
```

7.13 T_FF.v

```
module T_FF(
    input T,
    input clk,
    input reset,
    input enable,
    output reg Q
    );
    always @ (posedge clk or posedge reset) begin
     if(reset==1'b1)begin
        Q = 1'b0;
     end
     else begin
        if (T == 1'b1 & enable == 1'b1) begin
            Q = ~Q;
        end
        else begin
            Q = Q;
        end
     end
    end
endmodule
```