The Farmer, Fox, Chicken and Seeds

This lab will create a simple state machine based puzzle that can be implemented and played on an FPGA board. It involves the classic riddle of the farmer, fox, chicken and seeds. The students will build a Verilog based state machine with the information provided in a state table later in this document. The design will then be viewed using the RTL viewer in the Quartus software. Finally, the solution will then be synthesized and loaded into the Altera FPGA board.

The Puzzle:

A farmer is standing on the left bank of a river. He has with him a fox, a chicken and a bag of seeds. He needs to cross the river with all of his possessions. In the river is a small boat. The boat is just big enough for him and, **at most, one other item**. He will therefore need to make multiple trips to transport all his items to the right river bank. There is a catch: if the farmer leaves the fox alone with the chicken, the fox will eat the chicken. If the farmer leaves the chicken alone with the seeds, the chicken will eat the seeds. It is up to you to figure out how the farmer should move his items back and forth across the river with everything eventually ending up on the right river bank intact.

Converting the Puzzle into a State Machine:

A state table with every possible combination of inputs and present states has been created on the following pages. There are 16 states with a total of 64 entries in the table. Some of the entries are invalid and will be described shortly. The user input will be encoded into 2 bits as follows:

2'b00: Move the farmer across the river by himself
2'b01: Move the farmer across the river with the fox
2'b10: Move the farmer across the river with the chicken
2'b11: Move the farmer across the river with the seeds

SW[1] will be used for the upper bit and SW[0] will be used for the lower bit. SW[1] and SW[0] are represented in the state table as X and Y respectively.

The present states of the farmer, fox chicken and seeds are represented by Fa, Fo, Ch and Se respectively. If the state value is 0, the item is on the left bank of the river. If the state value is 1, the item is on the right bank of the river. LEDs will indicate what side of the river each item is on. The following table shows which LEDs represent which item and what side of the river it is on:

Left River Bank	Item	Right River Bank
LEDR[17]	Farmer	LEDR[3]
LEDR[16]	Fox	LEDR[2]
LEDR[15]	Chicken	LEDR[1]
LEDR[14]	Seeds	LEDR[0]

This configuration should provide an intuitive setup for playing the game. Keep in mind that an item can only be on one side of the river at a time. This means, for example, that if the farmer is on the right bank, LEDR[3] should be illuminated and LEDR[17] should be off**. They cannot be both on or both off at the same time.**

The next state of the farmer, fox, chicken and seeds are represented by Fan, Fon, Chn and Sen respectively. There are a total of three outputs: W, L, I which represent win, lose and invalid move. It does not make any sense for the farmer to move an item across the river that is on the opposite bank as

him. An example of this would be the farmer on the left bank and the seeds on the right bank and the user tries inputting a move that has the farmer cross the river with the seeds. It cannot be done because they are not on the same side of the river. This can be seen as the first red entry in the state table. **If this situation happens, the next state should be the same as the current state and the invalid move LED should be turned on as soon as the switches are changed**. The invalid move indicator will be LEDR[10].

There are multiple states that represent a losing situation. If one of these states is reached, the lose LED should illuminate. The next state after a losing state should always reset the game (all items on the left river bank). The lose indicator will be LEDR[11]. There is also a single winning state. If this state is reached, the win LED should be illuminated. The win indicator will be LEDG[8]. The next state after a win should reset the game as well.

Finally, a button needs to be provided so once the user sets up their input, the button can be pressed and the state machine will move to the next state. This button will be KEY[0]. A quick comment on mechanical buttons in digital systems needs to be made. When a mechanical button is used in a digital system, the 'bouncing' of the button can create noise and cause havoc on the system. To deal with this, a 'debounce' circuit is used to clean up the signal. For this lab we will be using a timer circuit to debounce the button. When the first transition of the button is detected, a timer is started and other transitions of the button are ignored until the timer expires. If the state of the button is the same as after the first transition when the timer expires, the button state is changed. **This applies to both pressing and releasing the button**. The debounce circuit is already provided as part of the lab and does not need to be designed by the students.

The following table provides a summary of the inputs and outputs on the FPGA board used in this lab:

I/O	Board Item	Description
Input	SW[1:0]	2-bit user encoded input
Input	KEY[0]	Enter button used for advancing the state machine
Output	LEDR[17:14]	Indicators for items on the left bank of the river
Output	LEDR[3:0]	Indicators for items on the right bank of the river
Output	LEDR[10]	Invalid move indicator
Output	LEDR[11]	Lose indicator
Output	LEDG[8]	Win indicator.

The state table below has five different colored entries. The colored entries represent the following:

Black: Valid state
Blue: Losing state
Green: Winning state
Red: Invalid input
Orange: Reset next state

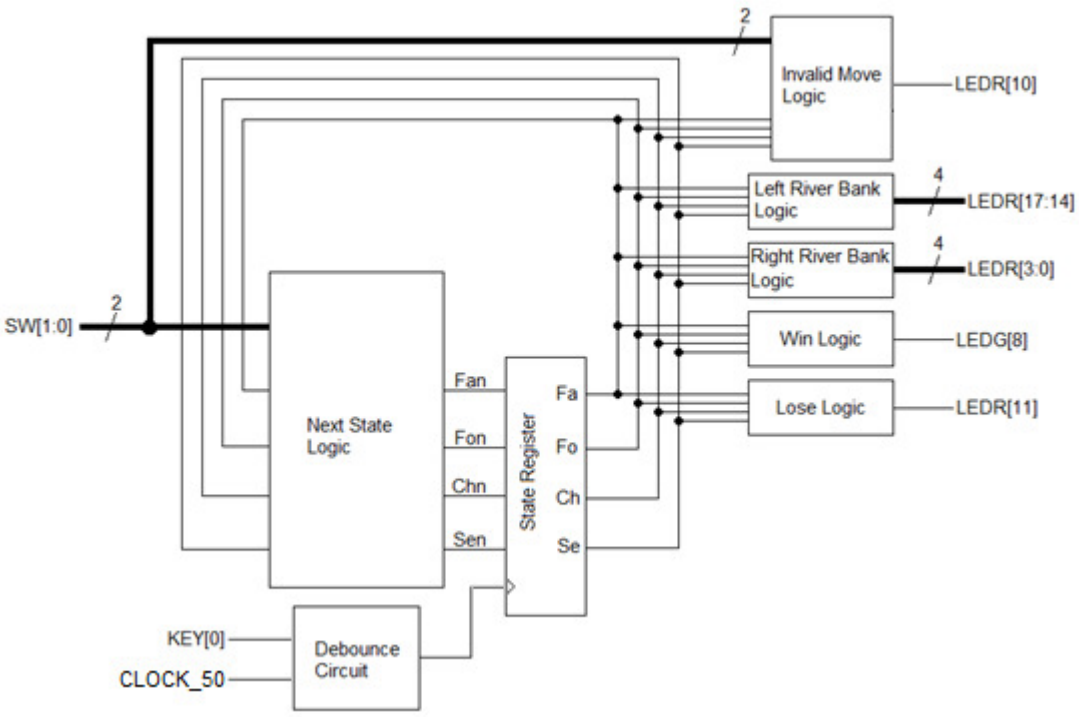
State Table:

Present State				Inputs		Next State				Outputs		
Fa	Fo	Ch	Se	X	Y	Fan	Fon	Chn	Sen	W	L	I
0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	1	1	1	0	0	0	0	0
0	0	0	0	1	0	1	0	1	0	0	0	0
0	0	0	0	1	1	1	0	0	1	0	0	0
0	0	0	1	0	0	1	0	0	1	0	0	0
0	0	0	1	0	1	1	1	0	1	0	0	0
0	0	0	1	1	0	1	0	1	1	0	0	0
0	0	0	1	1	1	0	0	0	1	0	0	1
0	0	1	0	0	0	1	0	1	0	0	0	0
0	0	1	0	0	1	1	1	1	0	0	0	0
0	0	1	0	1	0	0	0	1	0	0	0	1
0	0	1	0	1	1	1	0	1	1	0	0	0
0	0	1	1	0	0	0	0	0	0	0	1	0
0	0	1	1	0	1	0	0	0	0	0	1	0
0	0	1	1	1	0	0	0	0	0	0	1	0
0	0	1	1	1	1	0	0	0	0	0	1	0
0	1	0	0	0	0	1	1	0	0	0	0	0
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	0	1	0	1	1	1	0	0	0	0
0	1	0	0	1	1	1	1	0	1	0	0	0
0	1	0	1	0	0	1	1	0	1	0	0	0
0	1	0	1	0	1	0	1	0	1	0	0	1
0	1	0	1	1	0	1	1	1	1	0	0	0
0	1	0	1	1	1	0	1	0	1	0	0	1
0	1	1	0	0	0	0	0	0	0	0	1	0
0	1	1	0	0	1	0	0	0	0	0	1	0
0	1	1	0	1	0	0	0	0	0	0	1	0
0	1	1	0	1	1	0	0	0	0	0	1	0
0	1	1	1	0	0	0	0	0	0	0	1	0
0	1	1	1	0	1	0	0	0	0	0	1	0
0	1	1	1	1	0	0	0	0	0	0	1	0
0	1	1	1	1	1	0	0	0	0	0	1	0

State Table Continued:

Present State				Inputs		Next State				Outputs		
Fa	Fo	Ch	Se	X	Y	Fan	Fon	Chn	Sen	W	L	I
1	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	1	0	0	0	0	0	1	0
1	0	0	0	1	0	0	0	0	0	0	1	0
1	0	0	0	1	1	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	1	0
1	0	0	1	0	1	0	0	0	0	0	1	0
1	0	0	1	1	0	0	0	0	0	0	1	0
1	0	0	1	1	1	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	1	0	1	0	0	0	1
1	0	1	0	1	0	0	0	0	0	0	0	0
1	0	1	0	1	1	1	0	1	0	0	0	1
1	0	1	1	0	0	0	0	1	1	0	0	0
1	0	1	1	0	1	1	0	1	1	0	0	1
1	0	1	1	1	0	0	0	0	1	0	0	0
1	0	1	1	1	1	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	1	0
1	1	0	0	0	1	0	0	0	0	0	1	0
1	1	0	0	1	0	0	0	0	0	0	1	0
1	1	0	0	1	1	0	0	0	0	0	1	0
1	1	0	1	0	0	0	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0	1	0	0	0
1	1	0	1	1	0	1	1	0	1	0	0	1
1	1	0	1	1	1	0	1	0	0	0	0	0
1	1	1	0	0	0	0	1	1	0	0	0	0
1	1	1	0	0	1	0	0	1	0	0	0	0
1	1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	1	1	1	1	1	0	0	0	1
1	1	1	1	0	0	0	0	0	0	1	0	0
1	1	1	1	0	1	0	0	0	0	1	0	0
1	1	1	1	1	0	0	0	0	0	1	0	0
1	1	1	1	1	1	0	0	0	0	1	0	0

The following is a high level block diagram of the state machine:



Implementing the State Machine:

1. Create a new project in Quartus.
2. Copy Quartus Settings File (.qsf) and Verilog (.v) file into project directory.
3. Import pin assignments from the .qsf file into project.
4. Add .v file to project.
5. Complete unfinished sections of .v file.
6. Synthesize project and load it into an FPGA for testing.
7. View RTL results in the RTL viewer.
8. Demonstrate project to lab instructor by playing and winning the game.

What kind of state machine is this (Mealy or Moore)?
Can something be added/removed to make this the other type of state machine? If so, what?