Slot Machine

Final Project Report

Dec. 14, 2002

E155

Jason Quach and Daniel Sutoyo

ABSTRACT:

Over the years, slot machines have evolved from its mechanical roots and now are completely run by electronic systems. In light of this paradigm shift in operations, the objective of this project is to design and implement a slot machine that is run by the FPGA and HC11. The project incorporates 3 stepper motors, 3 reels (with eight icons on each reel), a keypad, two seven segment displays, wooden supports, along with the FPGA and HC11 in the creation of the slot machine. The main problem associated with this project is determining where the reels need to spin in each game and how to get them there with consistency. The result of the project is a working slot machine that is very fun to play.

1. INTRODUCTION

1.1. BACKGROUND

Slot machines, unlike traditional table games, require no gambling knowledge. Its simplicity has brought it unparalleled success as the most popular and profitable game in casinos worldwide. Although modern versions of slot machines retain the old classic view, it has steadily evolved over the years and now is run by an electronic system rather than a mechanical system. Thus, modern slot machines may look like the old ones, but they are run on an entirely different principle. The outcome of each game is actually determined by the central computer inside the machine, not the motion of the reels.

The slot machine created in this project is quite similar to that of a modern slot machine. The project consists of three stepper motors, three reels (each with eight icons), wooden supports for the reels, a 4x4 keypad using three buttons (spin, increase bet, and decrease bet), two dual seven segment displays which show the remaining credits and the amount of credits being bet, and a LED that lights up when the player hits the jackpot. When the user presses start, the amount bet is deducted and the reels all begin to spin. The first reel makes about ten full revolutions before stopping, while the second reel makes about twelve, and the third makes about fifteen. There are three spinning speeds for the reels: slow (810 rad/s), medium (1014 rad/s), and fast (1206 rad/s). The reels initially start out at medium speed, increase to fast speed, and finally revert to slow speed as they near their destination. If two or three icons match, the player is rewarded accordingly.

1.2. MOTIVATION

The motivation for creating a slot machine deals with the shift from mechanical to electrical systems in recent years. With modern slot machines being controlled by electrical systems rather than mechanical systems, it is feasible and desirable to create such a device with the tools at our disposal.

1.3. BLOCK DIAGRAM

The following picture is the block diagram of the slot machine. There are five major modules that are used in the implementation of the slot machine: the keypad, random number generator, motor, encoder, and credits module. The keypad module controls the button presses. The random number generator module does what its name suggests and generates pseudo random bit sequences using LFSR (linear feedback shift registers). The encoder module takes as inputs the random numbers and encodes it into the corresponding icons that will be displayed. The motor module's function is to spin the reels to the correct location. Finally, the credits module keeps tracks of the remaining credits, bets, and winnings.

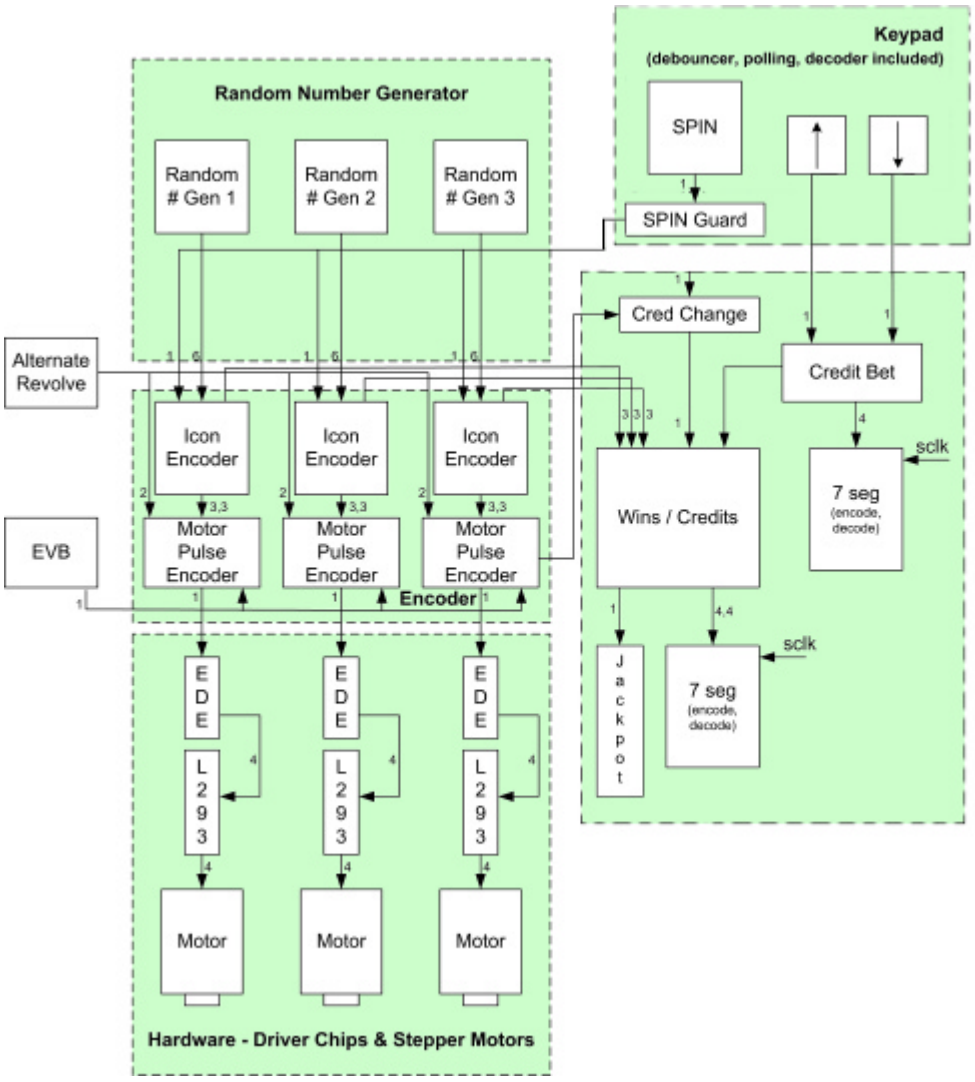


Figure 1: Overall Block Digram

1.4. PARTITIONS OF THE OVERALL SYSTEM

1.4.1. FPGA

At the heart of any modern slot machine is a random number generator, which ensures that each game has an equal chance of hitting the jackpot. The FPGA acts as the central computer of the slot machine and performs three important functions: generating random numbers, controlling the movement of the reels, and keeping track of the number of credits. When the user presses the start button, the FPGA automatically deducts the amount of credits being bet and also obtains three random numbers which will be used to determine the next set of symbols that will appear on the reel. Using this information, the FPGA sends the necessary information that will drive the stepper motors to the correct location. After the reels stop spinning, the FPGA determines if there is a win or not and rewards the amount of credits won; in the case of a jackpot, an LED lights up.

1.4.2. HC11

The microcontroller is used to control the spinning speed of the reels. For the purposes of this project, the microcontroller is used to generate a square wave that exhibits a different frequency for three intervals. The first interval of the square wave has a frequency of 323 Hz, the second has a frequency of 384 Hz, and the final interval has a frequency of 258 Hz. When the start button is pressed, the EVB (polling for this signal) generates a square wave with 323 Hz for about 1 second, then changes to 384 Hz for about 1.2 second, and finally adjusts to 258 Hz until all the motors stop spinning. Once this happens, the EVB reverts to outputting a square wave of 323 Hz until another game is started, which causes the EVB to repeat the same set of instructions. The square wave signal outputted from the EVB is sent to the motor module, which is responsible for generating the pulses which run the stepper motors.

2. NEW HARDWARE

2.1. STEPPER MOTOR

The Airpax unipolar stepper motor is an inexpensive device ($1.95) that offers great precision and control. It has high torque, step angles of 7.5 degrees, the ability to spin either clockwise or counterclockwise, in addition to supporting a wide variety of supply voltages. Also, the rotation speed of a stepper motor is independent of load, provided it has sufficient torque to overcome slipping.

In this project, stepper motors are used to move the reels to their correct location. The team decided to use plastic reels due to its relatively light weight. This would prevent the motors from slipping from insufficient torque. The team connected circular metal plates with a tiny hole in the center to the metal shaft of the stepper motor. The metal plate was super glued to the shaft and acted as a platform to which the reels could be attached. As can be seen, slot machines require consistent and accurate positioning of the reels and stepper motors fulfill both of these requirements nicely.

A unipolar stepper motor is made up of two coils and has 5, 6, or 8 leads. The stepper motors that the team worked with had 6 leads, consisting of: ground, power, and the 4 phase signals that control the two coils of the motors. To drive these motors, the team utilized a motor driver chip and an H bridge to facilitate its operation. More specifically, the team used the EDE1200 chip to translate a pulse into the 4 phase signals needed to drive the motor and the L293D to amplify the current of these 4 phase signals.

2.2. EDE1200 MOTOR DRIVER CHIP

The EDE1200 unipolar stepper motor driver chip is a 5 volt 18 pin device that provides control over 5/6 leads stepper motors. Its two main capabilities are driving motors in stand alone “run” mode or external pulse drive “step” mode. In run mode, the stepper motors freely spin based on the predetermined pin settings. In step mode, the stepper motor takes a step on every falling edge of a pulse. Other features in the motor driver chip include half-stepping, direction, and speed control (only in “run” mode). Inputs to the EDE1200 are power, ground, oscillator connection, and an external pulse if the chip is set to “step” mode.

The main reason the EDE1200 was chosen is because of the simplicity of outputting stepper motor phase sequences through the input of a single pulse. Other reasons included the variety of stepper motor behavior controls. Instead of creating logic to output alternating phase sequences to the 4 leads of the stepper motor, one only needs to output a pulse to the EDE1200. At the falling edge of the pulse, the EDE1200 will generate the phase signals for each of the 4 leads which will drive the motor.

The FPGA outputs the correct number of digital pulses (which corresponds to the number of motor steps) to the pin 9 of the EDE1200 chip. The pin connections for the EDE1200 are: P3, P4, P6, P7, P8, P10, P14 to 5V DC, P5, P11, P12, P13 to ground, and P15, P16 to the FPGA master clock signal (1 MHz). The digital pulse emitted from the FPGA is sent to P9 of the EDE1200. This will generate output drive signals P1, P2, P17, P18 (each of which have a 40mA maximum rating) for the four coils of a stepper motor. Refer to the bread board schematic to see how the chip is hooked up.

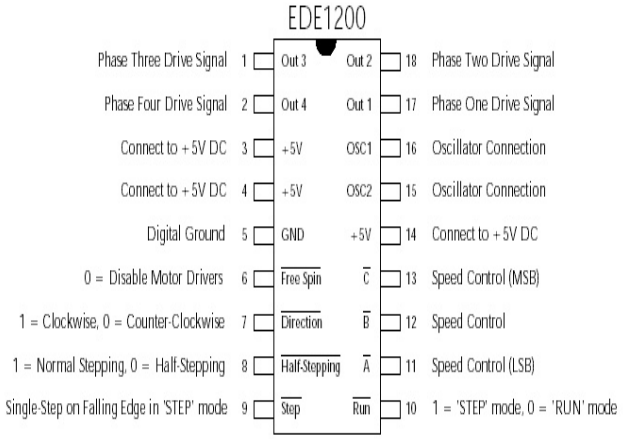


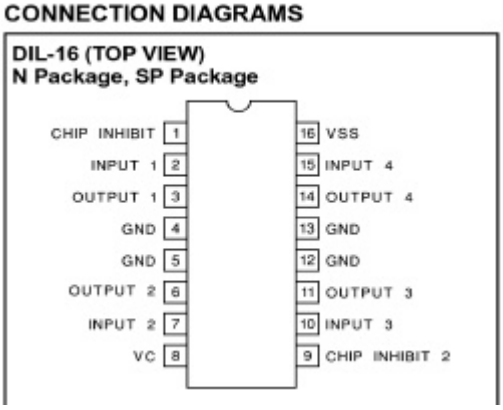
Figure 2: EDE1200 Motor Driver Pin Diagram

In order to ensure that the stepper motor is configured correctly, the team first tested the stepper motor in its "run" mode (P10 on ground) to make sure that it would freely spin. Once the team got the motor working in this mode, the team went on to test it in "step" mode. To test that the motor works in step mode, a square wave signal from the Agilent Wavetek Generator was applied to P9 of the EDE1200 chip. An observation that was made is that the EDE1200 speed control pins only apply in RUN mode. In step mode, the frequency of the square wave applied to P9 and the speed of the stepper motor are directly related.

For references on other key features and specifications of the EDE1200, please refer to http://www.componentkits.com/dslibrary/EDE1200.pdf (trouble shooting document) or http://www.elabinc.com/1200_faq.pdf (provides answers to several frequently asked questions).

2.3. L293D H-BRIDGE

Because the stepper motors require more current to operate than the EDE1200 can output, the team utilized the L293D to remedy this problem. The L293D integrated circuit is a very common motor driver chip that provides up to 600mA, which is more than enough current to sufficiently drive the stepper motors. The inputs to the L293D are power, ground, and the four phase drive signals.



The pin connections for the L293D are: P1, P8, P9, P16 to 5V DC, P4, P5, P12, P13 to ground, and P2, P7, P10, P15 receiving the corresponding EDE1200 drive signals. Then the L293D pins P3, P6, P11, P14 will deliver the same EDE1200 output signals but with much more current. The six lead stepper motor is connected in the following manner: red to 5V DC, orange to P2, yellow to P7, brown to P11, green to P14, and black unconnected (refer to figure 4).

3. SCHEMATICS

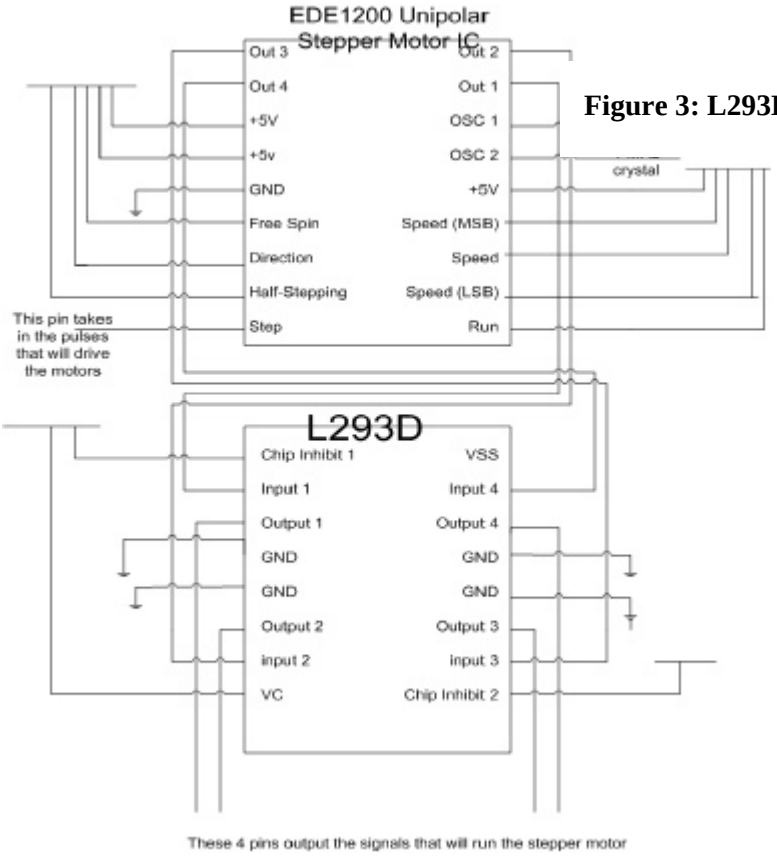


Figure 3: L293D Motor Drive Pin Diagram

Figure 4: EDE1200 and L293D Hookup Diagram

FPGA I / O Pins
Input Pins
P10, P45, P47-49, P81,
Vdd, ground (not shown)

Output Pins
P7-9, P13-14, P18, P23- 29, P46, P50-51, P61-62,
P65-69, P82-83

EVB I / O Pins
Input Pins
PortC[3:0]

Output Pins
PortB[0]

Note
The three blocks with Step, Osc 1 & 2, and OUT 1-4 are the overall block for the EDE1200 and L293D hook up diagram from figure 4.

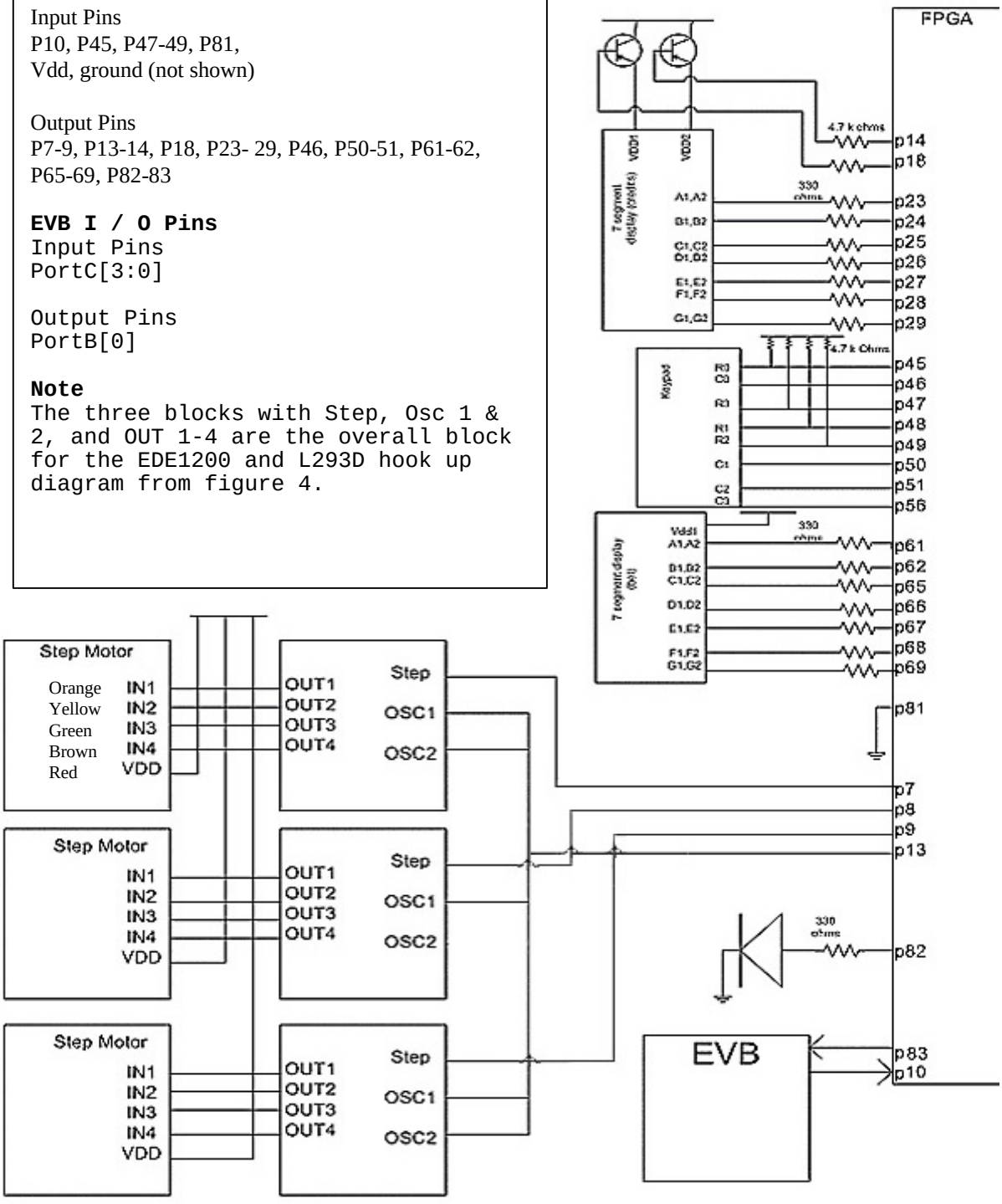


Figure 5: Bread Board Circuitry

4. FPGA

The FPGA takes in as inputs the EVB generated square wave and the user key presses. It outputs the credits and bet display, pulses that will be used to run the stepper motors, and a signal that informs the EVB when to start its instruction. The following descriptions are of the key modules involved in the implementation of the slot machine.

4.1. KEYPAD, CREDITS DISPLAY, AND BET DISPLAY

The keypad decoder module polls a 4x4 matrix keypad for input. The slot machine uses three input keys: spin, up, and down. The amount bet is controlled by the up and down buttons, ranges from 1 to 3, and is displayed on a dual seven segment display. The credits remaining are shown on a separate dual seven segment display and is initially set to 40. When the spin button is pressed, the credits remaining decreases based on the amount of credits bet. Once the player reaches zero, the game will not start until it is reset.

In our design, the number of credits available range from 0-99 and is represented in the verilog code as 8 bits. The 4 most significant bits are used to represent the ten's place and the 4 least significant bits are used to represent the one's place. In the verilog code, the ten's place is designated as "dig1" and the ones place designated as "dig0." This is done because the seven-segment decoder is limited to decoding single digit values. By splitting up the credits into two separate numbers, it will be easier to decode and display them. For more information, refer to appendix A – module 20.

4.2. RANDOM NUMBER GENERATOR

The random number generator used in this slot machine is emulated by the FPGA through the use of LFSR. Our design constantly generates three random numbers that range from 0 to 63. This is accomplished by using three different lengths of LFSR (6, 7 and 8 bit). Three random numbers are acquired directly from the three LFSR when the player presses spin.

4.3. ICON ENCODER

The encoder takes in, as inputs, each of the three random numbers. It outputs the current set of symbols and the next set of symbols. When the FPGA is turned on or when it is reset, the current set of symbols is defaulted to the watermelon icons. However, the reel positions will have to be manually reset because the FPGA has no memory of the previous position of the reels. On the rising edge of spin, three random numbers are sent to the encoder. At this point, the current set of symbols gets the values of the next set of symbols while the next set of symbols gets new values based on the three random numbers. In effect, this module acts as an encoder and shift register. The random numbers range from 0 to 63 and each of these values is mapped to one of the eight slot machine icons. Some icons have more than one number associated with it. Based on the icon LUT, the random numbers are encoded into their corresponding icons. Since there are eight different icons, they are encoded by using three bits. The following is a table of the icon encoding used in this project.

Table 1 – Icon Encoding Table

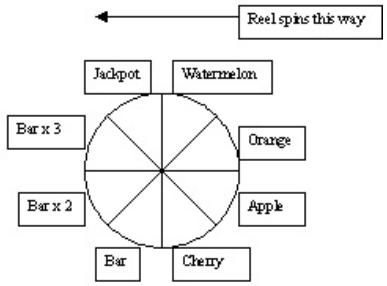
Dec. Val.	6-bit Random Sequence	Icon encoding	Icon
63	111111	111	Jackpot
61 – 62	111101 to 111110	110	Triple Bar
57 – 60	111001 to 111100	101	Double Bar
52 – 56	110100 to 111000	100	Bar
47 – 51	101111 to 110011	011	Cherry
37 – 46	100101 to 101110	010	Apple
27 – 36	011011 to 100100	001	Oranges
00 – 26	000000 to 011010	000	Watermelon

4.4. MOTOR PULSE ENCODER

The step motors rotate counterclockwise at 7.5 degree steps on the falling edge of a pulse. Since there are 8 icons on the reel, it will take 45 degrees to move from one icon to the next. Thus, it will take 6 steps at 7.5 degrees to move from one icon to the next. In order to determine the number of steps needed to go from one icon to the next, the encoder needs to take in as inputs the current set of symbols and next set of symbols. It is not difficult to create logic (Appendix A module 12 - 14) that will determine the number of steps that the current icon is away from the next icon. First, the number of icons between the current and next icon are found. Multiplying this number by six will yield the number of steps needed to go from the current icon to the next icon.

Table 2 – Steps Needed by Motor to move from Current Symbol to Next Symbol

How to read this chart: the current icons are located in column one and the next icons are located in row one, i.e. to go from orange to cherry is 36 steps.



	Jackpot	Bar x 3	Bar x 2	Bar	Cherry	Apple	Orange	W.Melon
Jackpot	0	6	12	18	24	30	36	42
Bar x 3	42	0	6	12	18	24	30	36
Bar x 2	36	42	0	6	12	18	24	30
Bar	30	36	42	0	6	12	18	24
Cherry	24	30	36	42	0	6	12	18
Apple	18	24	30	36	42	0	6	12
Orange	12	18	24	30	36	42	0	6
W.Melon	6	12	18	24	30	36	42	0

Each of the three reels will complete several revolutions before stopping at their destination. The first reel spins an extra 10 revolutions, the second reel spins an extra 12 revolutions, and the third reel spins an extra 15 revolutions. Since 48 steps constitute a complete revolution, 48*10 will be added to the number of steps for the first reel, 48*12 steps will be added to the second, and 48*15 steps will be added to the third.

The number of steps needed to move from the current icon to the next icon is then used to create a series of pulses that will drive the step motors. As previously mentioned,

stepper motors only respond on the falling edge of a pulse. Thus, setting the pulse signal equal to the least significant bit of a counter that counts up to 2 times the number of steps needed will accomplish this end. When the counter is equal to 2 times the number of steps needed, then the pulses will stop emitting until the user starts another game.

This module takes in as input the square wave generated by the EVB, which has a direct effect on the speed of the reels. By decreasing/increasing the frequency of the square wave that drives the series of pulses, the pulses themselves will occur at a lower/higher frequency which will in turn cause the step motor angular velocity to increase/decrease accordingly. Thus, the reels initially spin at medium speed, change to high, and then change to slow as they near their destination. This achieves the desired effect of gradually slowing down the spinning as the reel approaches its destination.

4.5. WINNING CREDITS

There are two ways to win when playing the slot machine. A player can either match two icons or match all three icons. This encoder takes in the next set of symbols and determines if there is a win. If there is a win, it will output the corresponding prize money. Else, it will output zero as the prize money. The prize money will not be added until the reels are finished spinning. The team created a signal named cred_change which informs the FPGA when to subtract/add credits. (refer to appendix A – module 19 for more information on this). In addition, if the user wins the jackpot, the LED will light up.

Table 3 Probabilities and Payoff

Table shown is credit bet = 1, for other payoffs refer to appendix A- module 18

3 Matching Icons	Prob. Of Winning Seq.	Numerical Value	Credits Won	2 Matching Icons	Prob. of Winning Seq.	Numerical Value	Credits Won
Jackpot	(1/64)^3	.0000038	LEDlight	Jackpot	(1/64)^2	.0002441	10
Triple Bar	(2/64)^3	.0000305	50	Triple Bar	(2/64)^2	.0009765	7
Double Bar	(4/64)^3	.0002441	30	Double Bar	(4/64)^2	.0002441	6
Bar	(5/64)^3	.0004768	10	Bar	(5/64)^2	.0039063	5
Cherry	(5/64)^3	.0004768	10	Cherry	(5/64)^2	.0061035	4
Apple	(10/64)^3	.0038146	5	Apple	(10/64)^2	.0244146	2
Oranges	(10/64)^3	.0038146	5	Oranges	(10/64)^2	.0244146	2
W. melon	(27/64)^3	.0750846	2	W. melon	(27/64)^2	.177978	0

5. MICROCONTROLLER DESIGN

5.1. OVERVIEW

The microcontroller design functions as a multiple frequency square wave generator. Its purpose is to provide a frequency varying square wave to the verilog module driveMotors and thereby accomplishing varying rotation speeds for the motors. A one-bit signal spinning is the only input PORTC[0] to the microcontroller design, and indicates when the instruction should be started. Spinning is a signal that goes high once the start button is pressed, and goes low once the motors are all finished rotating. The following finite state machine diagram explains the process of outputting the modulated square wave signal. Actual values calculated and used will be described in the next subsection

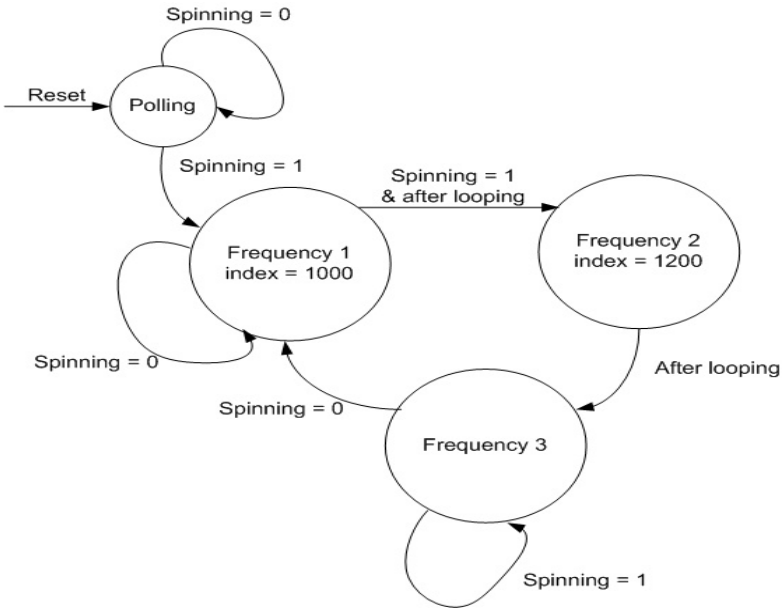


Figure 6: FSM of HC11 Varying Frequency Square Wave Generator

The actual output is not shown in this state diagram, but it should be noted that the output bit Port B[0] is toggled on and off every time the HC11 loops to the same state. On reset, the HC11 polls to detect when the spin button is pressed. Once an initial input is provided, the HC11 goes to the Frequency 1 state. Every time the HC11 loops to the same state the index is decremented by 1 as long as spinning is high. Thus, the higher the index, the longer the interval of that particular square wave frequency. When the index is reduced to zero in the Frequency 1 state, the HC11 jumps to the Frequency 2 state and repeats the process. Once the HC11 reaches the third state, it stays there until all the motors are finishing spinning (in other words, when spinning = 0), at which point it will return to the first state and await for another game to begin (spinning = 1).

5.2. DESIGN

Designing a varying square wave generator can be accomplished by two general time delays. The first is an index looping that can dictate the time period spent in each frequency state. The second is the output compare function with the timer counter, which can accurately determine when the output bit is toggled (half of the square wave period). Index looping is a fairly trivial concept. For example, if it takes the HC11 .01 seconds to execute all the commands within a loop, then looping a hundred times will now result in the HC11 taking an entire second to execute the instructions. The index values are selected by first calculating the total number of cycles within the loop and then dividing by the HC11 operating speed of 2.46 MHz to find the time period to complete one loop. Multiplying by the index value will determine the time period in each state. With some further experimentation, the index values 1000 and 1200 were found to produce the desired results for the project.

Output compare time delays determine the half period of the square wave. A 10 ms delay program using the output compare can be found in *Software and Hardware Engineering* by Fredrick M. Cady Pg 194. Using the example as the basic algorithm and

changing the offsets to correspond to the right frequency, the remaining task is to output the square wave. The output should alternate high low to achieve a square wave signal. Therefore each output compare also includes an output check to toggle the output on / off. This is accomplished by simply loading 1 (toggles on) or clearing the register (toggles off).

In Appendix B, the offsets chosen are 20490, 20492, and 20495 which yields respective frequencies of 258, 323, and 385 Hz. Typically, taking the offset and dividing by the clock frequency will give the time delay. Calculations were done on offset values 20000 and 10000 which yielded the expected results of 121 and 242 Hz for a 2.4567 MHz oscillation. However, similar calculations were unsuccessful for other desired frequencies. This could possibly be due to the timer flag overflow during the other commands such as the output toggle. The team chose the simple solution of trial and error and obtained the desired frequency by verifying the results with the logic analyzer.

6. RESULTS

The slot machine has three reels each with eight icons. The player begins with 40 available credits and has the ability to increase the credit bet to a maximum of three. The credit bet is deducted from the credits available at the beginning of each play. The reels spin at three speeds, starting with medium, changing to high, and ending with slow as they approach their destination. The amount won is added to the credits available after the last reel is done spinning. The user can reach a maximum of 99 credits and will not be able to play if the remaining credits available are insufficient. In the case of a jackpot, an LED will light up. A prominent feature in the final results that were not in the initial proposal was a safe guard against a player continually pressing or holding down the start button while the reels are spinning. Without this safeguard, these actions can potentially disrupt the system and cause the reels to spin to the wrong position.

The most difficult part of this design was ensuring that the motors would spin to the correct location. The electrical aspect of this design was not too difficult, while the mechanical aspect proved quite problematic. First, the team had to determine the logic that would give the number of steps going from one motor to the next. Getting the motors to spin at the correct speeds and verifying that the motors would spin to the correct location consistently took many hours of testing or debugging. There were three main reasons why the motors would not spin to the correct location. First, the L293D chip was initially not set up properly. Second, the electrical wiring was not secure, which resulted in the motor vibrating rather than stepping. The final reason was the connection between the reel and the stepper motor. The team realized that when the icons did not display correctly, it was not because of the bugs in the program or the electrical wiring. It was because the reels were not securely attached to the stepper motors, which would cause the stepper motor and reel to not spin in synchronization. Once the connection between the stepper motor and reel were secure, the icons displayed correctly

Some improvements that could be made in this project would be to rely more on the HC11 to ease the burden of using too many CLBs on the FPGA. Also, since the random number generator is such an integral part of the modern slot machine, the project would be well served if a genuine random number generator could be implemented as opposed to the pseudo bit sequences of the LFSR.

7. REFERENCES

[1] Fredrick M. Cady, *Software and Hardware Engineering*, Oxford University Press, 1997.
[2] EDE1200 Spec Sheet http://www.componentkits.com/dslibrary/EDE1200.pdf
[3] EDE1200 FAQ http://www.elabinc.com/1200_faq.pdf

8. PARTS LIST

Part	Source	Vendor Part #	Price
EDE1200	Jameco	141532	3 x 8.95 = 26.85
L293	Digikey	296-9518-5-ND	3 x 2.70 = 8.10
Stepper Motor	Jameco	164056	3 x 1.95 = 5.85

Total = $40.80

APPENDIX A: VERILOG MODULES

1.
```
module top_level(clk,reset,evbClk,rows,cols,led,sclk,sclk2,
                 segments,segments2,pulse1,pulse2,pulse3,
                 start,spins);
   input clk, evbClk;
   input reset;
   input [3:0] rows, cols;
   output start, sclk, sclk2;
   output led,pulse1,pulse2,pulse3;
   output [6:0] segments,segments2;
   output spins;

   wire en;
   wire start2;
   wire [1:0] bet;
   wire [3:0] press;
   wire [3:0] dig0,dig1;
   wire [7:0] prizeMoney;

   //this module used to create a slower clock signal
   //which will be used in the dual seven seg display, keypad,
   //and debouncer
   delayclk             delayclk(clk,reset,sclk,sclk2);

   //these two modules are used to control the keypad
   keypad               keypad(sclk,reset,rows,cols,press);
   debouncer            debouncer(sclk,reset,press,en);

   //start only goes high if enough credits available.
   //This ensures that the player
   //cannot go under zero credits.
   assign start = ~press[3] && ~((dig0 < bet)&& (dig1==0));

   //this module makes sure that the game only starts at the
   //appropiate times.  in other words, it guards against the
   //player holding on to the start button and pressing it rapidly
   spinning       spinning(clk,reset,add,start,spins);

   //this module allows the user to change their bet
   creditBet            creditBet(clk,reset,en,~press[2],~press[1],bet);

   //this module creates the random numbers used in the slot machine
   //In addition, it determines if the player has won and it also
   //creates the pulses that will run the stepper motors
   randomNumGen_Encode randomNumGen_Encode(clk,evbClk,reset,start,bet,
                              pulse1, pulse2, pulse3, prizeMoney, add);

   //this module keeps track of when the credits need to be change
   //it tells the win_credits module when to add and subtract credits
   creditchange   creditchange(clk,reset,spins,add,cred_change);

   //this module adds and subtracts the credits when necessary
   wins_credits   wins_credits(clk,reset,bet,prizeMoney,
                            cred_change,dig0,dig1,led);
   //this module displays the credits remaining
   disp                 disp(sclk,reset,dig1,dig0,segments);

   //this module displays the number of credits being bet
   sevenseg       sevenseg({2'b00,bet}, segments2);
```

```
endmodule
```

2.
```
module delayclk(clk,reset,sclk,sclk2);
   input clk;
   input reset;
   output sclk;
   output sclk2;
   /* this module will be used to control the
   mux for the dual segment display. it generates
   a clock that is much slower, which is used
   to alternately power the two transistors
   that control the dual segment display */

   reg [10:0] count;
   always @(posedge clk or posedge reset)
   if (reset) count <= 0;
   else count <= count + 1;

      assign sclk = count[10];
      assign sclk2 = ~count[10];

   endmodule
```

3.
```
module keypad(clk,reset,rows,cols,press);
    input clk;
    input reset;
    input [3:0] rows;
    output [3:0] cols;
    output [3:0] press;
   // this module determines when one of the three keys are pressed

    reg       state;
    reg [3:0] cols;
    reg [3:0] keys;
    reg [3:0] press;

    always @(posedge clk or posedge reset)
           if (reset) begin
                 state <= 0;
                 cols <= 4'b1110;
                 press <= 4'b1111;
      end else if (&rows) begin //if &rows =1, then no button being pressed
                 state <= 0;   //otherwise,a button being pressed
                 press <= 4'b1111;
                 cols <= {cols[0],cols[3:1]};
                 //polling the columns,since no button is being pressed
      end else if (~state) begin
                 state <= 1;
                 press <= keys;// one of the buttons being pressed
      end

      always @ (rows or cols)   //determines which button is being pressed
                 case ({rows,cols})
                              //we are only concerned with three buttons
                 8'b0111_1110: keys <= 4'b0111;
                 8'b1011_1110: keys <= 4'b1011;
                 8'b1101_1110: keys <= 4'b1101;
                 default: keys <= 4'b1111;
           endcase
   endmodule
```

4. **module debouncer**(clk,reset,din,en);

```
   input clk;
   input reset;
   input [3:0] din;          // row input from matrix keypad
   wire press;               // press signal, is any row shorted?
   output en;                // if an input is debounced it will
                             // be a signal when we want to shift inputs

   reg [1:0] state, nextstate;

   assign press = ~& din;    // NAND of rows, which means true when there
                             // is a 0 since we know a button is pushed
                             // when a row is shorted ( having a 0 in 4
                             // bit)

   // STATE REGISTER
   always @ (posedge clk or posedge reset)
     if (reset) state <= 2'b0;
     else state <= nextstate;

   // NEXT STATE LOGIC
   always @ (state)          // This code basically is 4 states, and will
                                // generate through until it hits the
  4th
                                // state
      case (state)

      2'b0: if(press) nextstate <= 2'b01;
                  else nextstate <= 2'b0;
      2'b01: if(press) nextstate <= 2'b10;
                  else nextstate <= 2'b0;
      2'b10: if(press) nextstate <= 2'b11;
                  else nextstate <= 2'b0;
       2'b11: if(press) nextstate <= 2'b11;
                  else nextstate <= 2'b0;
      default: nextstate <= 2'b0;
      endcase

   assign en = (state == 2'b11); // Simply whenever it hits final state,
                                 // the inputs need to be shifted since the
                                 // signal is debounced
endmodule
```

5. **module spinning**(clk,reset,add,spin,spinning);

```
   input clk,reset,add,spin;
   output spinning;

   reg state,nextstate;
   //This module makes sure that while the motors are spinning,
   //another game cannot be started.

   always @ (posedge clk or posedge reset)
     if (reset) state <= 0;
     else          state <= nextstate;

   /*
   The time between posedge spin and posedge add is the time
```

```
   wherein we want to guard against the player pushing and holding the spin
   button. On posedge of spin, we want to make sure from that point
   that any instances where the player presses start will not restart the
   game. Thus, on posedge spin, we set state = 1 if add = 0.  Once add
   becomes high, we set state = 0.
   */

   always @ (spin or add)
     if(spin)
           if(add)     nextstate <=0;
           else        nextstate <=1;
     else if(add)
           nextstate <= 0;
     else  nextstate <= state;

   assign spinning = state;

endmodule
```

6. `module creditBet(clk,reset,en,up,down,val);`

```
   /* this module controls the betting mechanism in the slot machine.
     there is an increase and decrease button. the betting range is
     1 to 3. if you try to go higher than 3 it recycles to 1. if you try
     to go lower than 1, it goes to 3.
   */
   input clk,reset,up,down,en;
   output [1:0] val;

   parameter ONE = 2'b01;
   parameter TWO = 2'b10;
   parameter THREE = 2'b11;

   reg [1:0] state;
   reg [1:0] nextstate;

   // STATE REGISTER

   always @ (posedge clk or posedge reset)
     if (reset)   state <= ONE;
     else         state <= nextstate;

   //enable is acquired through the debouncer module
   always @ (posedge en or posedge reset)
     if (reset)   nextstate <= ONE;
     else

      case (state)
           ONE:  begin
                 //if you press up, bet goes to two
                       if(up) nextstate <= TWO;
                 //if you press down, bet goes to one
                       else if (down) nextstate <= THREE;
                       else nextstate <= ONE;
                 end
           TWO:  begin
                       if(up) nextstate <= THREE;
                       else if (down) nextstate <= ONE;
                       else nextstate <= TWO;
                 end
           THREE:begin
```

```
                        if(up) nextstate <= ONE;
                        else if (down) nextstate <= TWO;
                        else nextstate <= THREE;
                    end
              default:    nextstate <= ONE;

       endcase

   // OUTPUT LOGIC

     assign val = state; //val is the amount of the bet

endmodule
```

7.
```
module randomNumGen_Encode(clk,evbClk,reset,start,bet,
                           pulse1, pulse2, pulse3,
                           prizeMoney, add);

   input clk,evbClk, reset,start;
   input [1:0] bet;
   wire [5:0] randNum1,randNum2,randNum3;
   wire [2:0] currentIcon1, currentIcon2, currentIcon3;
   wire [2:0] nextIcon1, nextIcon2, nextIcon3;
   wire [10:0] motorData1, motorData2, motorData3;
   wire revolve;
   output [7:0] prizeMoney;
   output add;
   output pulse1,pulse2,pulse3;


   //These three modules are the 6,7,8 bit LSFRs.
   randomGen1     randomGen1(clk,reset, randNum1);
   randomGen2     randomGen2(clk,reset, randNum2);
   randomGen3     randomGen3(clk,reset, randNum3);


   //This module encodes the random numbers into the corresponding icons
   //upon the posedge of start.
   iconEncoding
     iconEncoding1(reset,start,randNum1,currentIcon1,nextIcon1);

   iconEncoding
     iconEncoding2(reset,start,randNum1,currentIcon2,nextIcon2);

   iconEncoding
     iconEncoding3(reset,start,randNum1,currentIcon3,nextIcon3);

   // These modules encodes the current and next icons into the
   // corresponding motor information. The alternator module is used to
   // facilitate the encoding of motor information
   alternator     alternator(reset, start, alternate,revolve);

   motorEncoding
     motorEncoding(currentIcon1,nextIcon1,alternate,revolve,motorData1);

   motorEncoding2
     motorEncoding2(currentIcon2,nextIcon2,alternate,revolve,motorData2);

   motorEncoding3
     motorEncoding3(currentIcon3,nextIcon3,alternate,revolve,motorData3);
```

```
   /*
   This module encodes the motor information into the pulses that drive the
   stepper motors. The third driveMotor module outputs a signal add, which
   tells you when the credits won should be added.  This add signal goes
   high after the final motor has stopped spinning.
   */

   driveMotors          driveMotor1(evbClk,reset,  motorData1 ,pulse1);
   driveMotors          driveMotor2(evbClk,reset,  motorData2 ,pulse2);
   driveMotorsAdd       driveMotor3(evbClk,reset,  motorData3 ,pulse3,add);


   // This module encodes the next icons to determine if there is a win and
   // it also outputs the money won.

   winsequence2 winsequence2(bet,nextIcon1,nextIcon2,nextIcon3,prizeMoney);

endmodule

8. module randomGen1(clk,reset,randNum1);
   input clk;
   input reset;
   output [5:0] randNum1;

   //this module is an 6 bit LFSR

   reg [5:0] rand;

   always @(posedge clk or posedge reset)

      if(reset)
            rand <= 6'b111111;

      else
            begin
            //polynomial for 6 bit LFSR: 1 + x^5 + x^6
            rand[0] <= rand[4]^rand[5];
            rand[5:1] <= rand[4:0];
            end

   assign randNum1 = rand[5:0]; //outputs the random 6 bit sequence

endmodule

9. module randomGen2(clk,reset,randNum2);
   input clk;
   input reset;
   output [5:0] randNum2;

   //this module is a 7 bit LFSR
   reg [6:0] rand2;

   always @(posedge clk or posedge reset)

      if(reset)
            rand2 <= 7'b1111111;
      else
            begin
            //polynomial for 7-bit LFSR is: 1 + x^6 + x^7
            rand2[0] <= rand2[5]^rand2[6];
```

```
            rand2[6:1] <= rand2[5:0];
            end
   assign randNum2 = rand2[5:0]; //outputs the random 6 bit sequence
endmodule
```

10.
```
module randomGen3(clk,reset,randNum3);
   input clk;
   input reset;
   output [5:0] randNum3;
   //this module is an 8 bit LFSR

   reg [7:0] rand3;

   always @(posedge clk or posedge reset)

      if(reset)
            rand3 <= 8'b11111111;

      else
            begin
            //polynomial for 8 bit LFSR is: 1 + x + x^6 + x^7 + x^8
            rand3[0] <= rand3[1]^rand3[5]^rand3[6]^rand3[7];
            rand3[7:1] <= rand3[6:0];
            end

   assign randNum3 = rand3[5:0];   //outputs the random 6 bit sequence

endmodule
```

11.
```
module iconEncoding(reset,start,randNum,currentIcon,nextIcon);

   input reset,start;
   input[5:0] randNum;
   output [2:0] currentIcon,nextIcon;

   //this module takes in random number and outputs the corresponding icon

   //randNum will be between 0 and 63

   parameter watermelon = 3'b000;
   parameter orange     = 3'b001;
   parameter apple      = 3'b010;
   parameter cherry     = 3'b011;
   parameter bar        = 3'b100;
   parameter bar2       = 3'b101;
   parameter bar3       = 3'b110;
   parameter jackpot    = 3'b111;

   reg[2:0] currentIcon,nextIcon;

   always @(posedge start or posedge reset)
     //will encode the random number
     //once user lets go of start button

      if (reset)
            begin
                  nextIcon <= watermelon;
                  currentIcon <= watermelon;
            end
      else
```

```
            begin
                  if(randNum <= 6'b011010)
                        nextIcon <= jackpot;

                  else if (randNum > 6'b011010 & randNum <= 6'b100100)
                        nextIcon <= orange;

                  else if (randNum > 6'b100100 & randNum <= 6'b101110)
                        nextIcon <= apple;

                  else if (randNum > 6'b101110 & randNum <= 6'b110011)
                        nextIcon <= cherry;

                  else if (randNum > 6'b110011 & randNum <= 6'b111000)
                        nextIcon <= bar;

                  else if (randNum > 6'b111000 & randNum <= 6'b111100)
                        nextIcon <= bar2;

                  else if (randNum > 6'b111100 & randNum <= 6'b111110)
                        nextIcon <= bar3;

                  else
                        nextIcon <= watermelon;
                        currentIcon <= nextIcon;
            end
endmodule
```

12. module alternator(reset,start,alternate,revolve);

```
   input reset, start;
   output alternate, revolve;

   /*the purpose of this module is to facilitate the interaction between
   motorEncoding and drivedata module.  Alternate is one bit and swithes
   back and from to 1 and 0 everytime slot machine is played.  This means
   that the motor coming into DriveData will always be different than the
   previous entry, meaning that the data will change everytime a new game
   is played.  Revolve is used so that upon reset, the stepper motors do
   not spin when reset is implemented.
   */

   reg alternate, revolve;
   always@(posedge start or posedge reset)

      if (reset) begin
                        alternate <= 0;
                        revolve <= 0;
                 end
     else       begin
                        alternate <= alternate + 1;
                        revolve <= 1;
                 end
endmodule
```

13. module motorEncoding(currentIcon,nextIcon,alternate,revolve,stepData);

```
   input alternate, revolve;
   input [2:0] currentIcon,nextIcon;
   output [10:0] stepData;
```

```
   //this module takes in the current state and the next state of icons
   //and outputs the number of steps it will take to go from
   //the current icon to the next icon

   parameter watermelon = 3'b000;
   parameter orange     = 3'b001;
   parameter apple      = 3'b010;
   parameter cherry     = 3'b011;
   parameter bar        = 3'b100;
   parameter bar2       = 3'b101;
   parameter bar3       = 3'b110;
   parameter jackpot    = 3'b111;

   reg [2:0] motorData;
   always @ (nextIcon or currentIcon)
     //motorData is the number of icons that the current icon is away
     //from the next icon. the following is the algorithm for solving
     //this number

      if(currentIcon > nextIcon)
            motorData <= 8  + nextIcon - currentIcon;
      else
            motorData <= nextIcon - currentIcon;



      assign stepData = 6*{4'b0, motorData}+ 480*revolve + 48*alternate;

     /*
      The first term is the number of actual steps to next icon. Each step
      is 7.5 degrees and each icon covers 45 degrees on the reel so you
      need 6 steps to go from one icon to the next.

      The second term is the number of extra revolutions we want the reel
      to spin.  Since one full revolution is 48 steps,
      then the number of extra revolutions in this case is
      10. The revolve variable is zero on reset and one after start has
      been pressed

      The third term is used so that each set of stepData will always be
      different on each successive play.  Since the module that codes for
      the motor pulse depends on stepData being different everytime, the
      slot machine would work incorrectly if stepData was the same on two
      consecutive plays.  This case would happen if a player were to
      get the same icon on the same reel on two consecutive plays.

     */

endmodule
```

14. module motorEncoding2(currentIcon,nextIcon,alternate,revolve,stepData);

```
   input alternate,revolve;
   input [2:0] currentIcon,nextIcon;
   output [10:0] stepData;

   //this module takes in the current state and the next state of icons
   //and outputs the number of steps it will take to go from
   //the current icon to the next icon

   parameter watermelon = 3'b000;
```

```
   parameter orange     = 3'b001;
   parameter apple      = 3'b010;
   parameter cherry     = 3'b011;
   parameter bar        = 3'b100;
   parameter bar2       = 3'b101;
   parameter bar3       = 3'b110;
   parameter jackpot    = 3'b111;

   reg [2:0] motorData;
   always @ (nextIcon or currentIcon)
     //motorData is the number of icons that the current icon is away
     //from the next icon. the following is the algorithm for solving
     //this number

      if(currentIcon > nextIcon)
            motorData <= 8  + nextIcon - currentIcon;
      else
            motorData <= nextIcon - currentIcon;



     assign stepData = 6*{4'b0, motorData}+ 576*revolve + 48*alternate;


     /*
       The first term is the number of actual steps to next icon.
       Each step is 7.5 degrees and each icon covers 45 degrees on the reel
       so you need 6 steps to go from one icon to the next.

       The second term is the number of extra revolutions we want the reel
       to spin.  Since one full revolution is 48 steps,
       then the number of extra revolutions in this case is
       12.  The revolve variable is zero on reset and one after start has
       been pressed

       The third term is used so that each set of stepData will always be
       different on each successive play.  Since the module that codes for
       the motor pulse depends on stepData being different everytime, the
       slot machine would work incorrectly if stepData was the same on two
       consecutive plays.  This case would happen if a player were to
       get the same icon on the same reel on two consecutive plays.
     */
endmodule
```

15. module motorEncoding3(currentIcon,nextIcon,alternate,revolve,stepData);

```
   input alternate,revolve;
   input [2:0] currentIcon,nextIcon;
   output [10:0] stepData;

   //this module takes in the current state and the next state of icons
   //and outputs the number of steps it will take to go from
   //the current icon to the next icon

   parameter watermelon = 3'b000;
   parameter orange     = 3'b001;
   parameter apple      = 3'b010;
   parameter cherry     = 3'b011;
   parameter bar        = 3'b100;
   parameter bar2       = 3'b101;
```

```
    parameter bar3       = 3'b110;
    parameter jackpot    = 3'b111;

    reg [2:0] motorData;
    always @ (nextIcon or currentIcon)
      //motorData is the number of icons that the current icon is away
      //from the next icon. the following is the algorithm for solving
      //this number

       if(currentIcon > nextIcon)
             motorData <= 8  + nextIcon - currentIcon;
       else
             motorData <= nextIcon - currentIcon;

      assign stepData = 6*{4'b0, motorData}+ 720*revolve + 48*alternate;

      /*
        The first term is the number of actual steps to next icon.
        Each step is 7.5 degrees and each icon covers 45 degrees on the reel
        so you need 6 steps to go from one icon to the next.

        The second term is the number of extra revolutions we want the reel
        to spin.  Since one full revolution is 48 steps,
        then the number of extra revolutions in this case is
        15.  The revolve variable is zero on reset and one after start has
        been pressed

        The third term is used so that each set of stepData will always be
        different on each successive play.  Since the module that codes for
        the motor pulse depends on stepData being different everytime, the
        slot machine would work incorrectly if stepData was the same on two
        consecutive plays.  This case would happen if a player were to
        get the same icon on the same reel on two consecutive plays.
      */

endmodule
```

16. module driveMotors(clk,reset,motorData,pulse);

```
    input clk,reset;
    input [10:0] motorData;
    output pulse;

    //motorData is the number of steps needed to get to next icon

    reg [10:0] currentData, nextData;
    reg [10:0] state,nextState;

    // the following is the state machine that creates the pulses
    // that will drive the step motor


    always @(posedge clk or posedge reset )

       if (reset)  begin
                         state <= 0;
                         nextData <= 0;
                         currentData <= 0;
                   end
       else        begin
                         state <= nextState;
```

```
                        currentData <= nextData;
                        nextData <= motorData;
                end
   // he step motor spins on the falling edge the pulse.  thus,
   // the newData and currentData are used to tell when a new set of
   // motorData arrives, so that state will be reset to zero


always @(state or motorData or nextData or currentData)

   /*we need to count to 2 * motorData to get the appropiate amount
     of falling edges needed to run the motor*/

      if (state == motorData<<1)
              nextState <= state;

   /* if (state == 2* motorData), we don't want to create any more pulses.
     we want to reset state to zero only when we get a new motorData
     input. And if (state  < 2*motorData), we want to add 1 to state*/


      else if ( motorData > currentData)
             if (state == currentData<<1)
                 nextState <= 0;
              else
                 nextState <= state + 1;

      else if ( motorData < currentData)
                 if (state ==  currentData<<1)
                 nextState <= 0;
                 else  nextState <= state + 1;

      else nextState <= state+1;

     assign pulse = state[0];

   //the pulse is generated from the least significant bit of state

endmodule
```

17. `module driveMotorsAdd(clk,reset,motorData,pulse,add);`

```
   input clk,reset;
   input [11:0] motorData;
   output pulse,add;

   // motorData is the number of steps needed to get to next icon
   // add is a signal that will go high once the pulse is done being
   // emitted. It will be used to determine when the credits should be
   // added.

   reg [11:0] currentData, nextData;
   reg [11:0] state,nextState;
   reg add,nextAdd,spike,nextSpike;

   // the following is the state machine that creates the pulses
   // that will drive the step motor


   always @(posedge clk or posedge reset )
```

```
if (reset) begin
      state <= 0;
      nextData <= 0;
      currentData <= 0;
      add <=1;
      spike <=0;
      // spike is used so that add will behave like a 'spike',
      // meaning that add will go high only for one clock cycle

end
else begin
      state <= nextState;
      currentData <= nextData;
      nextData <= motorData;
      add <=nextAdd;
      spike <= nextSpike;
end



// the step motor spins on the falling edge the pulse.  thus,
// the newData and currentData are used to tell when a new set
// of motorData arrives, so that state will be reset to zero


always @(state or motorData or nextData or currentData
            or add or spike)

      /*we need to count to 2 * motorData to get the appropiate
      amount of falling edges needed to run the motor*/

      /* if (state == 2* motorData), we don't want to create any more
      pulses. We want to reset state to zero only when we get a new
      motorData input. If (state  < 2*motorData), we want to add 1 to
      state*/

if (state == motorData<<1)
      if(spike == 1)
            begin
                  nextState <=state;
                  nextAdd <= 0;
                  //add goes low again, remaining high for only one
                  nextSpike <= spike;     //clock cycle
            end
      else

            begin
            nextState <= state;
                  nextAdd <= 1;
                  //here add goes high.  This happens only when
                  nextSpike <= 1;
                  //spike=0 and state =2*motorData.
            end



// at this point, add will remain low because state is still
// increasing.

else if ( motorData > currentData)
      begin
```

```
                if (state == currentData<<1)
                      nextState <= 0;
                else
                      nextState <= state + 1;

                nextAdd <= 0;
                nextSpike <= 0;
          end

     else if ( motorData < currentData)
          begin
                if (state ==  currentData<<1)
                      nextState <= 0;
                else
                      nextState <= state + 1;

                nextAdd <= 0;
                nextSpike <= 0;
          end

     else
          begin
                nextState <= state+1;
                nextAdd <= 0;
                nextSpike <= 0;
          end


    assign pulse = state[0];
    //the pulse is generated from the least significant bit of state

endmodule
```

18. module winsequence2(bet,icon1,icon2,icon3,prizeMoney);

```
   input [1:0] bet;
   input [2:0] icon1,icon2,icon3;

   output [7:0] prizeMoney;

   // win goes high if all three icons are the same


   parameter watermelon2      = 4'b0000;
   parameter orange2          = 4'b0001;
   parameter apple2           = 4'b0010;
   parameter cherry2          = 4'b0011;
   parameter bar2             = 4'b0100;
   parameter bar22            = 4'b0101;
   parameter bar32            = 4'b0110;
   parameter jackpot2         = 4'b0111;

   parameter watermelon3      = 4'b1000;
   parameter orange3          = 4'b1001;
   parameter apple3           = 4'b1010;
   parameter cherry3          = 4'b1011;
   parameter bar3             = 4'b1100;
   parameter bar23            = 4'b1101;
   parameter bar33            = 4'b1110;
   parameter jackpot3         = 4'b1111;
```

```
reg [3:0] lookicon;
reg [7:0] winnings;

always @(icon1 or icon2 or icon3)
//goes through here anytime any of the icons change

//Checks here if any two icons match.
if ((icon1 == icon2) || (icon1 == icon3) || (icon2 == icon3))

  //Checks to see if all three icons match.
  if ((icon1==icon2) && (icon2 == icon3))
   lookicon <= icon1 + 4'b1000;

  //Following "else if" statements check for which two icons match.
  else if (icon1==icon2)
   lookicon <= icon1;
  else if (icon2==icon3)
   lookicon <= icon2;
  else if (icon1==icon3)
   lookicon <= icon3;
  else
   lookicon <= 4'b0000;

//If no two icons match, then there is a lost.
else
  lookicon <= 4'b0000;

always @(lookicon or bet)

  if (bet==2'b01)  //if bet = 1
   begin
         case(lookicon)
               // prizeMoney: 4 most significant bits represent 10's
               // place 4 least significant bits represent 1's place

         watermelon2:            winnings <= 8'b0000_0000;
         orange2:                winnings <= 8'b0000_0010;
         apple2:                 winnings <= 8'b0000_0010;
         cherry2:                winnings <= 8'b0000_0100;
         bar2:                   winnings <= 8'b0000_0101;
         bar22:                  winnings <= 8'b0000_0110;
         bar32:                  winnings <= 8'b0000_0111;
         jackpot2:               winnings <= 8'b0001_0000;
         //winnings for two icons matching

         watermelon3:            winnings <= 8'b0000_0010;
         orange3:                winnings <= 8'b0000_0101;
         apple3:                 winnings <= 8'b0000_0101;
         cherry3:                winnings <= 8'b0001_0000;
         bar3:                   winnings <= 8'b0001_0000;
         bar23:                  winnings <= 8'b0011_0000;
         bar33:                  winnings <= 8'b0101_0000;
         jackpot3:               winnings <= 8'b1111_1111;
         //winnings for three icons matching

         default:                winnings <= 8'b0000_0000;
         endcase
   end

  else if (bet==2'b10) //if bet = 2, winnings are doubled
```

```
      begin
            case(lookicon)

            watermelon2:            winnings <= 8'b0000_0000;
            orange2:                winnings <= 8'b0000_0100;
            apple2:                 winnings <= 8'b0000_0100;
            cherry2:                winnings <= 8'b0000_1000;
            bar2:                   winnings <= 8'b0001_0000;
            bar22:                  winnings <= 8'b0001_0010;
            bar32:                  winnings <= 8'b0001_0100;
            jackpot2:               winnings <= 8'b0010_0000;


            watermelon3:            winnings <= 8'b0000_0100;
            orange3:                winnings <= 8'b0001_0000;
            apple3:                 winnings <= 8'b0001_0000;
            cherry3:                winnings <= 8'b0010_0000;
            bar3:                   winnings <= 8'b0010_0000;
            bar23:                  winnings <= 8'b0110_0000;
            bar33:                  winnings <= 8'b1010_0000;
            jackpot3:               winnings <= 8'b1111_1111;

            default:                winnings <= 8'b0000_0000;
            endcase
      end

     else   //if bet = 3, then winnings are tripled
      begin
            case(lookicon)

            watermelon2:            winnings <= 8'b0000_0000;
            orange2:                winnings <= 8'b0000_0110;
            apple2:                 winnings <= 8'b0000_0110;
            cherry2:                winnings <= 8'b0001_0010;
            bar2:                   winnings <= 8'b0001_0101;
            bar22:                  winnings <= 8'b0001_1000;
            bar32:                  winnings <= 8'b0010_0001;
            jackpot2:               winnings <= 8'b0011_0100;

            watermelon3:            winnings <= 8'b0000_0101;
            orange3:                winnings <= 8'b0001_0101;
            apple3:                 winnings <= 8'b0001_0101;
            cherry3:                winnings <= 8'b0011_0000;
            bar3:                   winnings <= 8'b0011_0000;
            bar23:                  winnings <= 8'b1001_0000;
            bar33:                  winnings <= 8'b1010_0000;
            jackpot3:               winnings <= 8'b1111_1111;

            default:                winnings <= 8'b0000_0000;
            endcase
      end


   assign prizeMoney = winnings;


endmodule
```

19. `module creditchange(clk,reset,spinning,add,out);`

```
   input clk,reset,spinning,add;
```

```
output out;
reg change, nextchange,en,nexten;

// This module gives the signal determining when to add/subtract credits
// In each play, there should be two pulses that will indicate credit
// change
// 1) Spin Keypress should deduct the credit bet from credit available
// 2) After all motors are done spinning, credit won adds to credit
// available

// change is the pulse that determines when to subtract
// en is a signal that assists to create the change


always @ (posedge clk or posedge reset)
  if (reset)
   begin
         change <= 0;
         en <= 1;
   end
  else
   begin
         change <= nextchange;
         en <= nexten;
   end


// Nextstate Logic
// only interested in creating a short pulse when we see posedge
// spinning

always @ (change or en or spinning)
  if (change == 0 && en == 1)
  // change goes high when it initially sees posedge spinning
   if(spinning)
         begin
               nextchange <= 1;
               nexten <= 0;
         end
   else
  // change goes low while motors are spinning
         begin
               nextchange <= 0;
               nexten <= 1;
         end
  else
   if(spinning)
  // change goes low while motors are spinning
         begin
               nextchange <= 0;
               nexten <= 0;
         end
   else
  // spinning is low, go back to state that detects spinning
         begin
               nextchange <= 0;
               nexten <= 1;
         end


  // change determines when to subtract
```

```
      // oring with add we can have a signal
      // that determines when to change the display of credits

   assign out = change || add;

endmodule

20. module wins_credits(clk,reset,bet,prizeMoney,cred_change,
                        dig0,dig1,led);
   input clk,reset;
   input [1:0] bet;      //this input is the amount of money that was bet
   input cred_change;    //this input tells when credits need to be
                         //added/subtracted
   input [7:0] prizeMoney; //this input is the amount of money won
   output [3:0] dig0,dig1; //dig1 stands for the 10's place
                           //dig0 stands for the 1's place
   output led;

   reg [3:0] nextdig0,nextdig1,dig0,dig1;
   reg enable, nextEnable, led;


   always @ (posedge cred_change or posedge reset)
     if (reset)
      begin
            dig1 <= 4'b0100;  //user gets 40 credits to start off with
            dig0 <= 4'b0000;
            enable <= 0;
      end
     else
      begin
            dig1 <= nextdig1;
            dig0 <= nextdig0;
            enable <= nextEnable;
      end


   always @ (dig1 or dig0 or enable or prizeMoney or bet)

   /*
   The following is our "adding" algorithim.  Since we represent
   the ten's digit with the 4 most significant bits and the one's digit
   with the 4 least significant bits, we cannot add the numbers
   directly.
   */

     if(enable)
      begin
            led <= 0;
            //If credits + prizeMoney > 99, then stay at 99.
            if ((dig1 + prizeMoney[7:4]) > 9)
                  begin
                        nextdig1 <= 9;
                        nextdig0 <= 9;
                  end


            else
                  // If there is a carry from the ones place then:
                  if (dig0 + prizeMoney[3:0] > 9)
```

```
                        //First, check if that will push you over 99.
                        //If so, stay at 99.
                        if ((dig1 + 1 + prizeMoney[7:4]) > 9)
                              begin
                                    nextdig1 <= 9;
                                    nextdig0 <= 9;
                              end
                        //If not, then perform a carry operation.
                        else
                              begin
                                    nextdig1 <= dig1 + 1 + prizeMoney[7:4];
                                    nextdig0 <= dig0 + prizeMoney[3:0] -
                                                                        10;
                              end
                  //If there is no carry operation, then add accordingly.
                  else
                        begin
                              nextdig1 <= dig1 + prizeMoney[7:4];
                              nextdig0 <= dig0 + prizeMoney[3:0];
                        end

                  //next Enable goes low once adding credits is complete.
                  nextEnable <=0;
      end

     else
      begin
            led <= &prizeMoney;
            //This is when there is carry.
            if (bet > dig0)
                  begin
                        nextdig1 <= dig1 - 1;
                        nextdig0 <= dig0 - bet + 10;
                  end

            //This is when there is no carry.
            else
                  begin
                        nextdig1 <= dig1;
                        nextdig0 <= dig0 - bet;
                  end

            //nextEnable goes high once subtracting credits is complete.
            nextEnable <=1;
      end

endmodule
```

21.
```
module disp(slowclk,reset,dig0,dig1,segs);
    input slowclk;
    input reset;
    input [3:0] dig0;
    input [3:0] dig1;
    output [6:0] segs;

      wire [3:0]  data;

     // when slowclk is high, then dig0 will be shown. otherwise, dig1 is
     // shown
      assign data = slowclk ? dig0 : dig1;
```

```
      //displays dig1 or dig2
       sevenseg sevenseg(data,segs);

endmodule

22. module sevenseg(data,segments);
    input   [3:0] data;
    output  [6:0] segments;

    reg     [6:0] segments;

// Taken from Verilog Handout Pg 27
// Segment  #              abc_defg
parameter   BLANK       =     7'b111_1111;
parameter   ZERO        =     7'b000_0001;
parameter   ONE         =     7'b100_1111;
parameter   TWO         =     7'b001_0010;
parameter   THREE       =     7'b000_0110;
parameter   FOUR        =     7'b100_1100;
parameter   FIVE        =     7'b010_0100;
parameter   SIX         =     7'b010_0000;
parameter   SEVEN       =     7'b000_1111;
parameter   EIGHT       =     7'b000_0000;
parameter   NINE        =     7'b000_0100;

always @ (data)
     case (data)

     0: segments <= ZERO;
     1: segments <= ONE;
     2: segments <= TWO;
     3: segments <= THREE;
     4: segments <= FOUR;
     5: segments <= FIVE;
     6: segments <= SIX;
     7: segments <= SEVEN;
     8: segments <= EIGHT;
     9: segments <= NINE;
     default: segments <= BLANK;
     endcase

endmodule
```

APPENDIX B: HC11 ASSEMBLY CODE

```
* Source File: Variable Frequency Clock Generator
* Author: Jason Quach and Daniel Sutoyo
* The basic structure of the code is taken from Miller's
* handbook example 10ms timer.

* Takes the SPINS signal from FPGA as an input and outputs
* a square wave that changes its frequency three times.
* The SPINS signal is a signal that goes high when
* the user presses start on the keypad,and goes low only after
* the last motor is done spinning, regardless of any other
* attempts by the player to press start again.  Thus,
* the user must wait before the last motor stops spinning
* before they can play again.

* TABLE OF CONSTANT OFFSET FOR DESIRED FREQUENCY
*********************************
TEN_MS      EQU   #20492                *323HZ
TEN_MS2     EQU   #20490                *385HZ
TEN_MS3     EQU   #20495                *258Hz

* I/O Register Equates
*********************************
REGS  EQU   $1000
TCNT  EQU   $0E
TFLG1 EQU   $23
TOC1  EQU   $16
OC1F  EQU   %10000000
PORTB EQU   $1004
PORTC EQU   $1003
DDRC  EQU   $1007

* Memory Map Equates
*********************************
PROG  EQU   $C000
DATA  EQU   $D000
STACK EQU   $DFFF

* Main Program
*********************************

      ORG   PROG
      lds   #STACK
      ldx   #REGS
      CLR   PORTB
      LDAA  #%11110000
      STAA  DDRC

POLL  LDAA  PORTC             * The Polling Mechanism determines
      CMPA  #%00000001        * if there is a spin keypress.
      BHS   START             * Once SPIN goes high, it will
      JMP   POLL              * run through the remainder of the code


* Time Loops
*********************************
* The EVB generates a square wave in port B[0].
```

```
* The Time Loops are divided into three sections
* and all three produce the corresponding frequency
* signal with the above constant offsets.
* The code utilizes the timer with output compares
* to determine when to set the signal high or low

START LDY   #1000
TIME  ldd   TCNT,X            * Retrive timer counter value
      addd  TEN_MS            * Add constant offset
      std   TOC1,X            * Store as output compare
      ldab  PORTB             * Port B is the output
      CMPB  #00               * If outputting 0, want to force bit to 1
      BEQ   BIT1
      CLR   PORTB             * If outputting 1, want to force bit to 0
      JMP   FLAG
BIT1  ldab  #01               * Forcing PortB[0] to 1
      stab  PORTB
FLAG  ldaa  #OC1F
      staa  TFLG1,X
spin  brclr TFLG1,X OC1F spin
      LDAA  PORTC             * PortC[0] is the SPINS input
      CMPA  #%00000001        * If SPINS is low, continue to loop
      BLO   TIME              * Until the SPINS is detected
      DEY                     * Once SPINS is detected, loop through
      BNE   TIME              * With the index y

      LDY   #1200             * The second section of time loop
TIME2 ldd   TCNT,X            * Is constructed in a similar fashion
      addd  TEN_MS2           * Without the SPINS input check
      std   TOC1,X            * It loops through with the specified
      ldab  PORTB             * Y index amount
      CMPB  #00
      BEQ   BIT2
      CLR   PORTB
      JMP   FLAG2
BIT2  ldab  #01
      stab  PORTB
FLAG2 ldaa  #OC1F
      staa  TFLG1,X
spin2 brclr TFLG1,X OC1F spin2
      DEY
      BNE   TIME2


TIME3 ldd   TCNT,X            * The third timer loop continues
      addd  TEN_MS3           * to loop until the motors are done spinning
      std   TOC1,X            * Afterwards it jumps back to START
      ldab  PORTB             * This mechanism can be accomplished
      CMPB  #00               * below
      BEQ   BIT3
      CLR   PORTB
      JMP   FLAG3
BIT3  ldab  #01
      stab  PORTB
FLAG3 ldaa  #OC1F
      staa  TFLG1,X
spin3 brclr TFLG1,X OC1F spin3

      LDAA  PORTC             * Check if SPINS is still high
      CMPA  #%00000001        * if high (which means motors are
      BHS   TIME3             * still spinning) continue to output
```

```
JMP   START             * the last frequency clock signal
```

APPENDIX C: PSEUDO RANDOM NUMBER ANALYSIS

The team had to consider the issue of whether the three different lengths of LFSR would properly generate all triple combinations of 6 bit random sequences. In any length LFSR, the repeating sequence length is 2^N-1. Thus, in the 6 bit LFSR, the repeating sequence is length 63. In the 7 bit LFSR, the repeating sequence is 127. And in the 8 bit LFSR, the repeating sequence length is 255. Since all of these lengths are relatively prime to each other, it is indeed possible to generate all possible combinations of three random 6 bit sequences (except that the 6 bit LFSR cannot generate 6 consecutive zeros).

Another issue that must be considered is the odds of winning. In any length LFSR, the maximum number of zeros that occur consecutively is N-1. Thus, in a 6 bit LFSR, it is impossible to obtain a 6 bit sequence of 000000. This is significant because it greatly increases the chances of winning the jackpot. In a 7 bit LFSR, a 6 bit sequence of 000000 is possible, but there are two possible 6 bit sequences of 111111. In an 8 bit LFSE, two 6 bit sequences of 000000 are possible, but now there are four possible 6 bit sequences of 111111. Because the icon encoding for the jackpot symbol is 111111, the odds of winning the jackpot are not exactly as stated in the table of probabilities and payoff.

The actual odds of winning the jackpot are: (1/63) * (2/127) * (4/255) = 3.921*10^-6, Compared with the listed odds of winning a jackpot (1/64)^3=3.815*10^-6. Thus, the difference between these two odds is winning the jackpot one extra time in 10 million plays. It should be noted that the odds of winning for the seven other icons are not exactly as shown in the project proposal, but remain close enough for the purposes of this project. If we assume that the three LFSR act as a true random number generator, then the table of probabilities become more intuitive

APPENDIX D: FINAL PROJECT

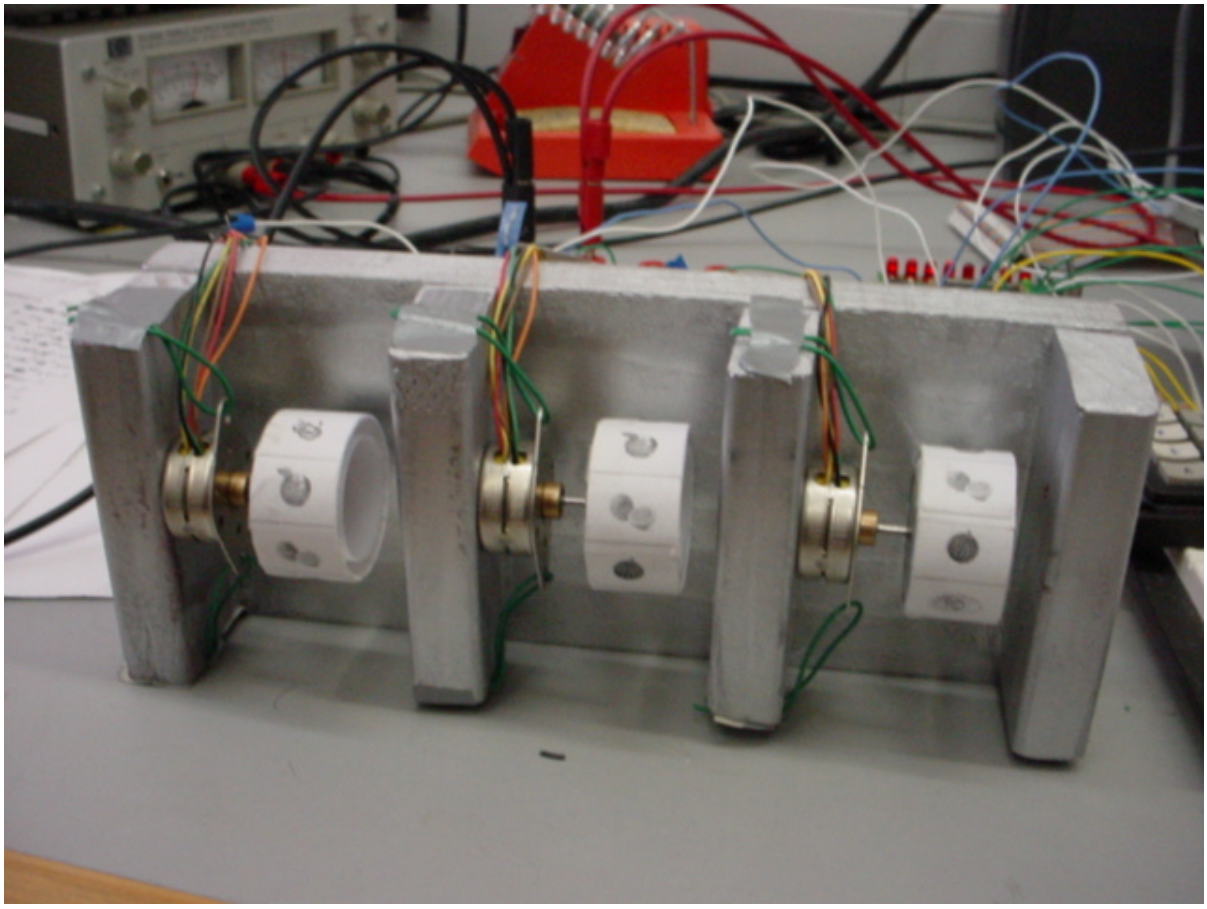


Figure 7: Slot machine reel display

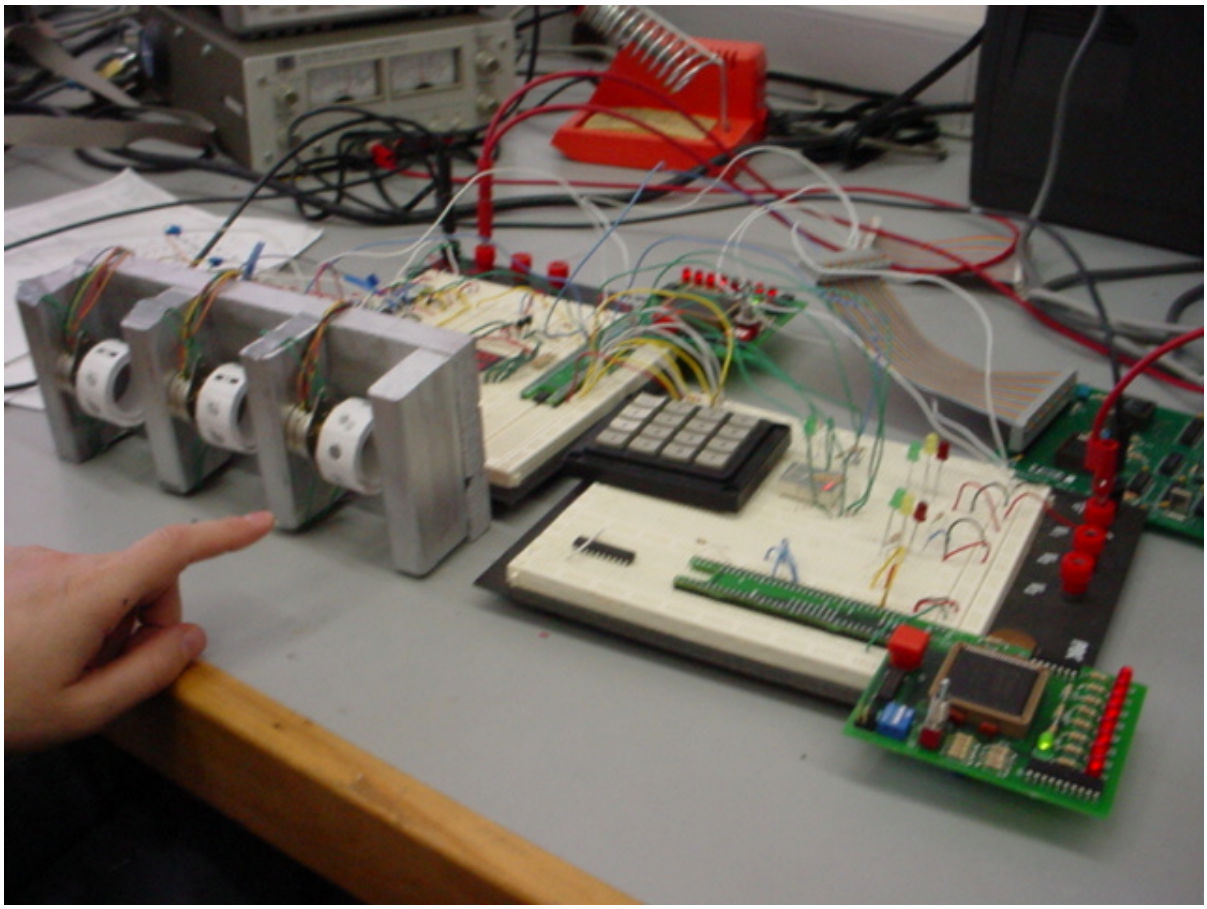


Figure 8: Slot machine hooked up to FPGA and HC11