BOWLING GAME

Final Report

December 15, 2003

E155

Shane OUCHI & Franck LUU

Abstract:

In all computer-bowling games, there are ten pins that are in a triangular formation, and a ball that is rolled down the lane in order to knock down the pins. Each player's goal is to knock down as many pins as possible within the two opportunities they have in each frame, and after ten frames whoever has the most points, wins. In our game, the rules and scoring our implemented by an FPGA, while our PIC controls the LCD screen on which the game is played. We were unable to successfully combine the PIC and FPGA in order produce a fully-functioning game. In this way, we were only able to establish the different pin-outcomes, and the location and speed detectors, which would oscillate on the LCD screen.

Introduction:

As the project for our E155 class, we decided to build a standard bowling game, played on an LCD screen, using implemented logic and functionality from an FPGA and PIC microcontroller. The object of the game is to bowl a ball down a lane in order to knock down a maximum of ten pins per frame, out of which ten frames are played and a maximum of 300 points can be scored. Each frame has two turns, in which a person can attempt to hit down ten pins. Ten pins down in the first turn is a strike; ten pins down by the second turn is a spare. The complications of scoring will be discussed later, along with the fact that we did not score the tenth frame how it is supposed to be done. The location and speed of the ball, which are determined by the player, exact the number of pins that are knocked down. In our rendition of the game, in order to start the game, the FPGA reset button is triggered and all the scores are zeroed out. The scores are displayed on the six seven-segment displays, and the game itself is played on the LCD screen. Ten pins are setup at one end of the screen in a triangular position, while the ball is setup at the other end. There will be a location bar at the same edge of the screen at which the ball begins. This bar will have a location indicator, which will oscillate between six different columns, until the player presses a button, which stops the oscillation at the desired column. The power bar will oscillate with three different strengths, and will also be stopped by the press of the same button. The location bar will scan first, and after the button is triggered, the speed bar will go (more discussed in location and power control section). After the button is pressed again, the designated pin-outcome will result. All of the actions within this game and the LCD display are controlled by the FPGA and PIC. The FPGA calculates the score of the game, the seven-segment displays, and oscillates the location and power bars. The PIC controls the game being displayed on the LCD.

LCD Implementation Intro:

In Appendix D, the diagram shows the overall interactions between the different components of the system. The PIC controls the LCD by connecting the 8-bit data bus to the LCD screen. The ports RC0 through RC2 and RC6 and RC7 connect the reset, controller 1 select, controller 2 select, enable, and data/instruction pins on the LCD, respectively. The read/write pin is always in write mode, so we grounded that pin. These

six pins, along with the data I/O pins, control all of the initialization and implementation of the data for the LCD. The reset pin on the LCD is active low, which means that it must be get kept high at all times for the LCD to work. The Cs1 and Cs2 pins are both needed because there are two controllers on the LCD (one for each side of the screen). Depending on which pin is high, designates which side of the screen we write to. The enable bit is used to send both data and instructions from the PIC to the LCD. The enable bit must be pulsed in order to transfer data or instruction to the LCD screen. Finally, the data/instruction pin designates whether data or instructions are being sent through the data I/O ports. There is also a –9.4 V output voltage pin (Vee), which we used as an input to the Vo pin because the Vo pin was the contrast pin for the LCD. There were also Vdd (+5V) and Vss (GND) terminals, which were the power pins for the LCD.

Remaining Hardware:

The rest of the hardware that we connected together was the four hexadecimal displays, the power transistors, and the various resistors that we needed in order to control the output voltage from the implementation board. In appendix D, we show a diagram of our schematics and what we did to connect the PIC, LCD, FPGA, and Hex displays together. The resistor values for each of the transistors are 4.7 kilo-ohms and the resistor values for hex-displays are 330 ohms. Basically, we designed the hardware such that the PIC can talk to the FPGA internally through serial interface, and the FPGA would send back information from pins 37-40 to the PIC at PORTA[3:0]. Furthermore, we designed the hex-display such that the segments would light up depending on the values coming out of pins 56-62, but depending on which transistor was on (controlled by pins 46-51), would be the specified display at which the segments would light. Finally, the button goes into the FPGA and is modified into a pulse before it is sent to the PIC to control the LCD actions. This is the hardware that we used to run our game.

FPGA:

The FPGA is used in computing most of the outcomes for each location-speed combinations, figuring out the score, and controlling other details such as the button-press and a couple different decoders. The various modules that are needed within this FPGA implementation are player-select, location control, power control, scoring, first turn, second turn, hexadecimal display modules from lab 3 and 4, hexadecimal-decimal converter, a

button unification function, and a switch register (each will be briefly discussed within the next sections of this paper). We tested the all of these modules and made sure that each executed the given tasks within simulation. Furthermore, we tested that our hexadecimal display control worked with our actual implementation of the FPGA. However, the lighting on the hexadecimal displays was quite dim because of such a small amount of power that was going to each display.

Player-select:

The player control module is the brain of the FPGA program. By counting every time the button is pressed, it is able to tell which modules what to do. It determines the player, the location or speed control, the first-turn or second-turn, and counts the frame. The way this is done is by using a FSM, which stays in its current state until the button is pressed. After both players have gone, it goes back to the first state and starts again. Appendix A shows the FSM for this module.

Location Control and Power Control:

The way that the location and power control works is that the PIC sends an eight-bit number through the SPI such that the first four bits of the number is the speed and the last four bits of the number are location. It goes through the one-hot decoder module, and when it comes out, the speed and location are both in one-hot form in order to inputted into either the first bowl or second bowl, depending upon the variables from the player-select module.

First Bowl:

Within this module, an output of tryone is assigned a value, depending upon the various combinations of the three different speeds and the six different locations. This assigned value is a four-bit number that, when sent to the PIC, will display a certain pin-outcome on the LCD. A score is assigned to each combination of the location and speed, depending on how many pins are knocked down (pin-outcome). Thus, this module outputs both a score for the first turn as well as a number designated to a certain shape within the LCD.

Second Bowl:

After the first-bowl is implemented, the second-bowl module takes whatever pin-outcome from the first bowl, and depending on the speed and location, again, assigns a pin-

outcome that is determined beforehand. The way that the second bowl uses the first bowl outcomes is that, there are seven pin-outcomes assigned to tryone. For each case allotted for the tryone's, we take each possible location and power combination that could occur for each of these tryone outcomes, in order to determine the fifteen different pin-outcomes that could possibly occur within this module. These outcomes would be assigned as numbers to the output trytwo. The first seven outcome possibilities for trytwo are the same as all the possible combinations for tryone.

Scoring:

In all of these modules, we set up a huge lookup-table for the various possibilities that occur for each frame. The scoring module was modeled a few times before coming to the algorithm that we currently use. The first draft of the module split the game into ten frames and scored each frame. The second cased it out by strike and spare combination possibilities, and scored it from there. The third and most compact method is the one that we currently implemented into our design (Appendix B). It consists of a previous strike-counting register, a two-strike in a row register and spare-counting register that are within the upper module itself. The previous strike register counts whether or not a strike occurred within the previous frame, and sends a value of one if it is true. The two in a row register checks whether or not there were two strikes in a row in the past two frames, and if so, sends out a one value. The spare counting register counts if there was a spare in the previous frame. By implementing all three of these registers, we found a simple algorithm that would accurately score through each of the first nine frames. However, because of memory issues, we could not score the tenth frame in its correct fashion, and instead, scored it identically to the first nine frames. The algorithm for this scoring is as follows:

finalscore <= finalscore + score1 + score2 + (score1 + score2)*pstrike
+ (score1)*ppstrike + (score1)*pspare,

where ppstrike is two strikes in a row counter, pstrike is one strike in a row counter, pspare is one spare in a row, and score1 and score2 are the two shots in the current frame.

Hexadecimal Display:

For the Hex Display, we designed it such that it would multiplex six different output displays from seven different inputs depending on the displays, which would be enabled at the time. Looking back at Appendix D, the pins P56- P62 are assigned to

transferring a seven-bit code to the Hex-Display in order to light up the corresponding segments. Furthermore, six transistors controlled by pins P46-P51 are used to toggle the data such that one of the six displays would be on at a time. Since the data would be switched at such a fast rate compared to the human eye, all six displays would appear to be on. Furthermore, we built a shift register to put the scores in one at a time, shifting it over every clock cycle, such that the scores would be displayed appropriately for each player.

PIC:

The PIC 18F452 was used in this project to drive the LCD display (Appendix C). The features used were:

- Interrupt
- SPI module

Here is how the PIC program is composed:

- Variable allocation
- Variable settings
- Instructions to set the addresses for the screens
- Instructions to clear the screens
- Instructions to draw ten pins on the left screen
- Loop drawing the location bar
- Loop drawing the speed bar
- Instructions to draw pins

The program jumps into the different loops by using the INT0 interrupt, which depending on certain variables (i.e. location, speed or draw pins), tells the program where to go next.

Variables settings:

In this part, we mainly initialize variables and set the different registers in order to control the input-output pins or functionality of the PIC by setting ADCON and ADCON1, in order to disable the A/D module and use PORTA as digital input.

Instructions to set the addresses for the screens

The way the program controls the LCD display is sending values to PORTC, which correspond to control bits on the LCD. By doing so, we can send instructions to set the addresses for the pages and the columns oh which the program is going to write. All the

values are stored in a database at address 0x2000. This procedure has to be performed on both sides of the screen.

Instructions to clear the screens

The set of instructions here is a loop setting PORTC and clearing the both screens.

Instructions to draw ten pins on the left screen

After the both sides of the screen have been set and cleared, the program starts to draw ten pins. Again to do so, we have to configure PORTC and send wanted value to PORTD. The pattern for the ten pins is stored in a database at address 0x1000.

Loop drawing the location bar

Up to this point, the interrupt is disabled such that the players are not allowed to stop the program while the LCD was being configured.

The program draws a bar for the location and loops until an interrupt occurs on the bit 0 of PORTB (button pressed).

The data is sent to PORTD, after being read in the database, which is stored at address 0x1C.

A variable "loc" is incremented for each position of the bar on the screen, the values are from 0x01 to 0x06. This variable is used to identify where the bar stopped and will be sent through the SPI module.

Loop drawing the speed bar

This part of the program is the same format as that of the location bar, except that we do not have to change pages. The program will also wait for an interrupt.

The data is sent to PORTD, after being read in the database, which is stored at address 0x1D.

A variable "sp" is incremented from 0x10 to 0x30, which is used to inform the PIC of the position of the speed bar.

Instructions to draw pins

At this point, the interrupt is disabled again. The program checks value entering from PORTA and has to set the pointers in order to read the tables (TBLPTRH, TBLPTRL). The patterns for all the pins' positions are stored from address 0x1000 to 0x1B00. The PIC was reading the value from PORTA, which has the values 0x01 to 0x0C. We add 0x0F to

PORTA in order to obtain the corresponding address for the corresponding pattern. This value is then sent into TBLPTRH in order to index the matching database.

The program also checks if a strike has been executed, and if so, it writes "STRIKE" on the screen. (Pattern stored at address 0x1E)

The interruption

In this game, we use the interrupt INT0. An interruption occurs when a state is changing on the bit0 of PORTB. To do so, we have to configure some registers as INTCON (enable GIE – Global interrupt flag – and RBIE - RB Port Change Interrupt Enable bit).

The interruption is set at address 0x08, where it is redirected to the "interrupt" label.

The interrupt is only enabled during two phases: drawing the location bar and drawing the speed bar.

The main utility of the interrupt is to guide the program where to jump, depending on the variable button. The program could jump from the interrupt into the "speed" label or the "draw_pins" label.

Before redirecting to the "draw_pins" label, the interrupt sends a value through the SPI module. This value is the sum of the "loc" and "sp" variables.

Loop delay

The whole program calls a delay loop several times in order to either display the location and speed bars for a certain time or delay the program to process certain commands.

PORTC implementation

The PORTC is configured as follows:

RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0
RES	CS2	X	X	X	CS1	E	I/D

X = Used for something else

We configure these five bits accordingly to what we want to display on the LCD.

RES: Active Low, reset the screen

CS2: If set, instruction or data send to the right side of the screen

CS1: If set, instruction or data send to the left side of the screen3

E: Enable bit, has to act like a pulse

I/D: If set Data, if clear Instruction

There is another bit R/W on the LCD display board but since we only write on the LCD display, we didn't use this bit and connected it directly to the ground. This sets the LCD to write.

In order to send instructions or data to PORTD, we have to configure PORTC and send specific values.

Here is a commented sample of code, in order to write on the LCD display:

```
movlw 0x85          ; RES is high (not active), CS1 = 1, set to instruction
movwf PORTC2
movlw 0x87          ; E =1
movwf PORTC
movlw 0xFF          ; send code to PORTD to be displayed
movwf PORTD
movlw 0x85          ; E=0
movwf PORTC
```

The data is sent to PORTD but is read at the falling edge of E.

To send data or instruction to PORTD, the same pattern of enabling is used, except varying values are sent through PORTC depending upon what the LCD should do.

RESULTS:

Within our project, we learned a lot about both programming the PIC to control a LCD screen, and also about FPGA programming for bowling games. We were able to develop a semi-functional replication of a bowling game, in that the FPGA and PIC functioned when run separately, but when installed together, they would not take each other's inputs and outputs. They were having difficulties communicating with each other, and the main problem that we discovered was that we had trouble making our FPGA export data to the PIC through PORTA, and had trouble importing the data through the SPI connection from the PIC. Therefore, we could not get the FPGA to output anything to the hexadecimal displays. Another problem that we had was that our PIC was not always

generating the exact images that we wanted on the LCD screen. We were able to get the pin-combinations to appear on the screen, and were able to generate both the location and power indication oscillations, but there were certain malfunctions when pressing the button. For the most part, the button would trigger the location and power oscillations to work correctly, but random dots and lines would appear at the same spot on the screen all of the time. With the combination of both of these problems, we were not able to get our game to function nearly as well as we had expected it to. In the following sections, we will go into detail about what went wrong and how we tried to ameliorate the problems that existed.

Communication Problems:

Within the FPGA, the main problem that occurred was the communication between the PIC through the SPI control, and outputting through the I/O ports to PORTA of the PIC. The first thing we did was to test our shift-register module to see if it was taking in the correct values and putting out anything. Within the simulation, the shift-register worked fine. We could not find anything wrong with it. However, even when we implemented a simple data command to go through SPI and have it sent back to the PIC, we did not receive anything in the PIC. So, we decided to test whether or not our PORTC/P82 was working. The PIC was sending the correct value to the SSPBUF, and was holding a value for several clock cycles, which was good. This led us to believe that our PORTC might not have worked. We tested PORTC with the logic analyzer and it seemed to be generating the signal that showed in the SSPBUF, so we narrowed our problems down to the FPGA. Yet, when we tested whether or not pins 37-40 were sending any signals to PORTA of the PIC, we got nothing. We tested this by setting up LED's across those pins in order to see if they would light up. We went back and tested our simulations a few more times, and tested simpler cases which should have generated a signal, but to no success. Thus, we could not get the FPGA to communicate with the PIC within the given amount of time allotted for the project.

LCD Problems:

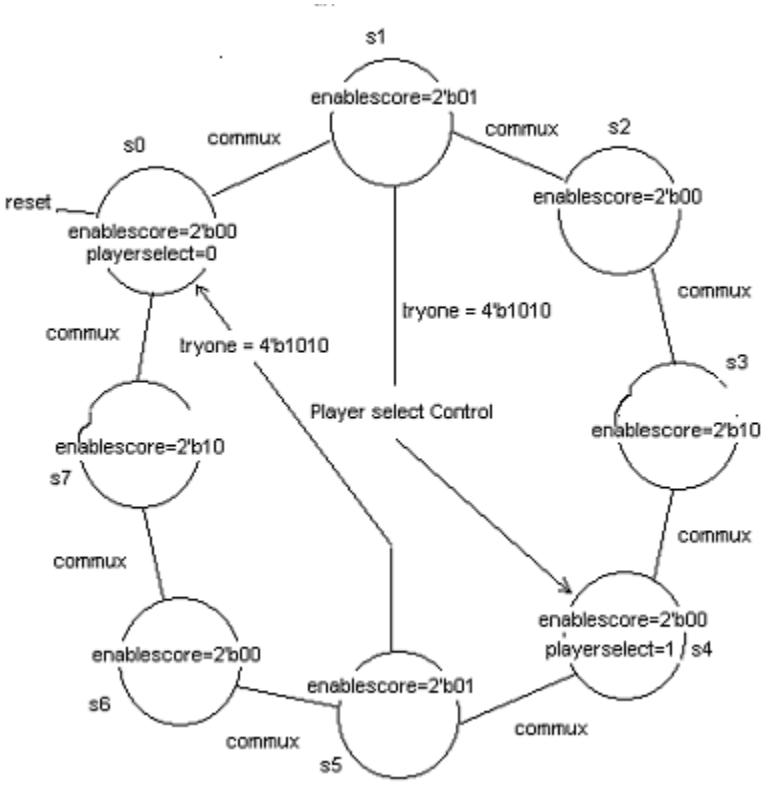
The LCD also had a problem. Although it displayed the pins, and both indicators, and also worked pretty well when we pressed the button to stop both indicators, we could not get a clear picture to appear. While the speed indicator moved, it would trigger a clear line to appear in the opposite half of the screen, even when we were not controlling that

side at all (CS1 would be low). We did not understand how it was happening, except for the fact that it might have been going into the wrong loops or calling the wrong portions of the program. However, when we ran it step by step, the program ran exactly how we wanted it to, but the line would still form. We checked the wiring to make sure that everything was correct, and nothing was wrong. We ended up suspecting that this glitch was caused by the fact that our interrupt would fail to work every so often (did not know why), and so because the program would not enter the interrupt sometimes, it would write unnecessary data to the screen. Furthermore, because we could not make the button-press into a perfect pulse because our FPGA would not function correctly, we believe that mechanical "spikes" that occur within the button press may have also caused these lines and random pixel glitches to appear.

Conclusion:

In conclusion, we attempted to design a bowling game that was fully-functional, and could be played by two players on an LCD screen. However, to our disappointment, we could not get communications between the PIC and FPGA to work, as well as a few minor screen glitches on the LCD, and thus could not get a working game. In the future, if any groups want to pursue this project in hopes of finding success, we would suggest working on getting the LCD to work properly first, then testing simple programs to check whether or not the communication between your hardware works, and then finally approaching the game itself. The reason for this is that both of those first two are the most time-consuming and most important of the project. The rest of the project does not matter too much because for that part, you actually control how difficult you make it. In this case, we made it too difficult, and so spent way too much time on it, rather than the hardware part, which needed a lot more attention than was given.

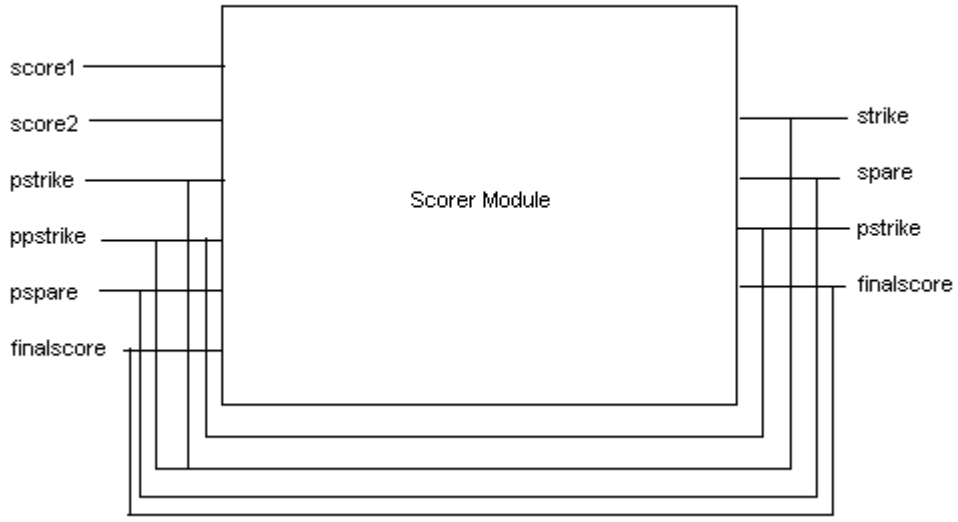
Appendix A



Here are the finite state machine for the player control.

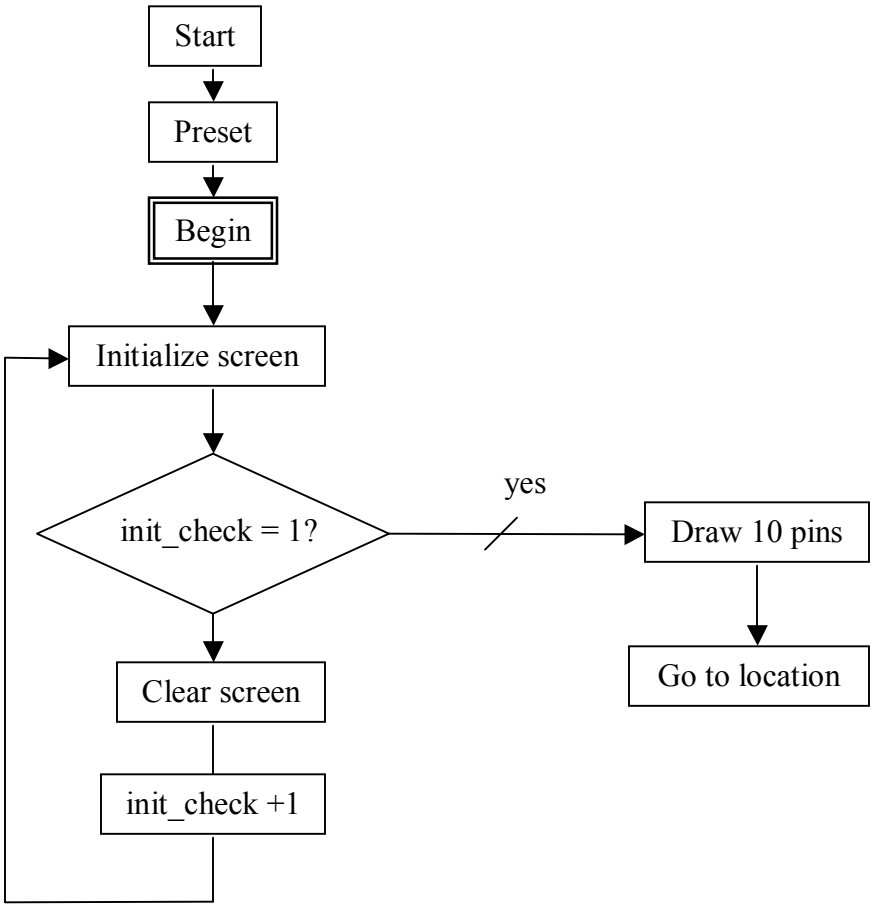
Appendix B

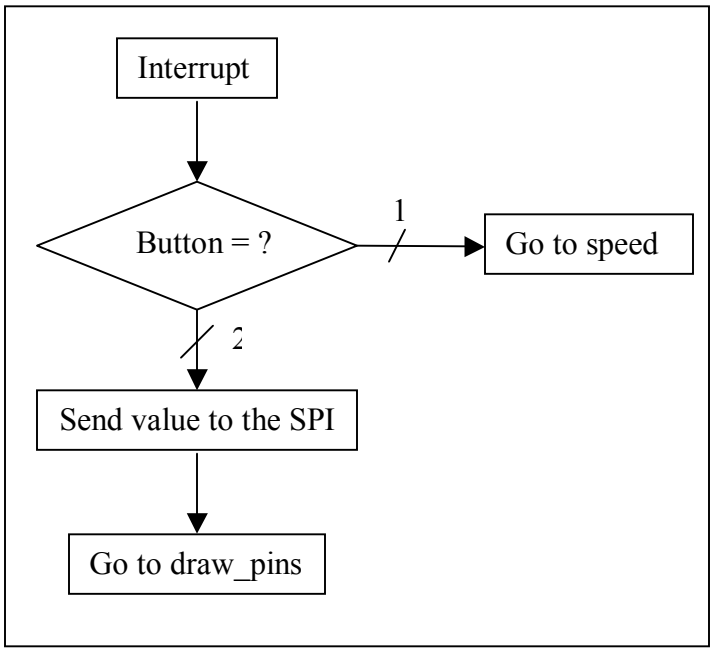
Here is the scoring “black box” representation for this module.

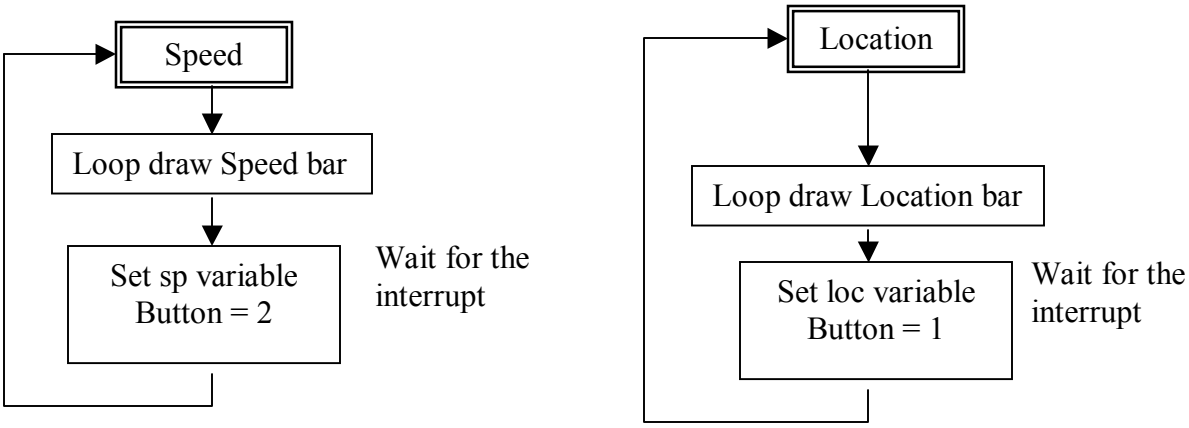


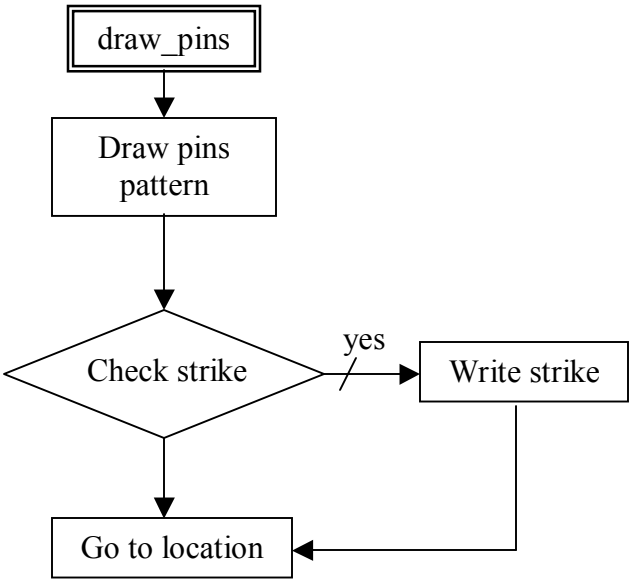
Appendix C

Here is the assembly language coding for the PIC:

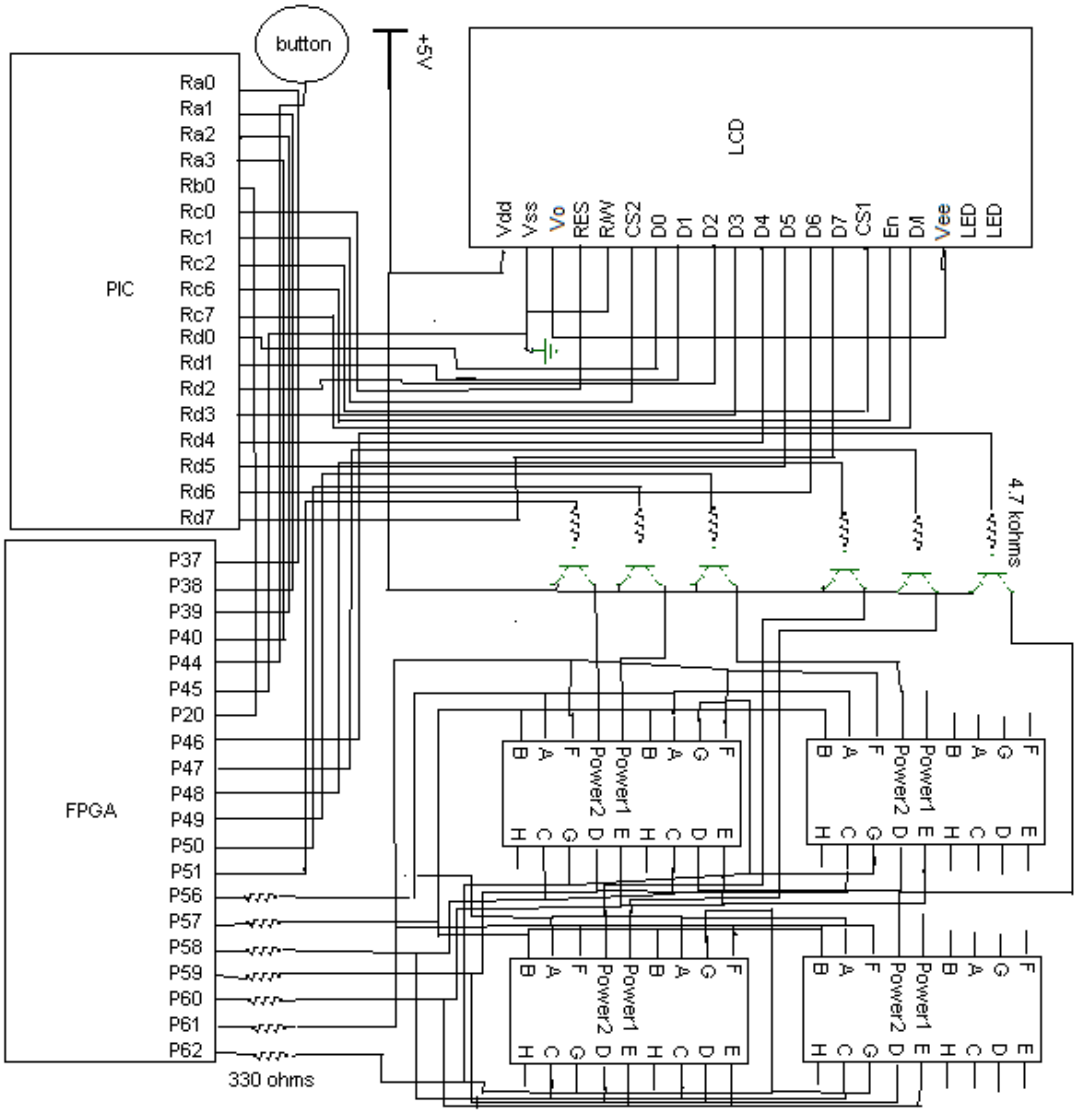





Speed
Loop draw Speed bar
Set sp variable
Button = 2
Wait for the
interrupt
Location
Loop draw Location bar
Set loc variable
Button = 1
Wait for the
interrupt


draw_pins
Draw pins
pattern
Check strike
yes
Write strike
Go to location

Appendix D



Here is a schematic of the overall hardware implementation of our design. The four boxes at the left represent four hexadecimal displays for scoring. The button at the top is for the player control of the game. The Vee to Vo is –9.4V, which controls LCD contrast.

Appendix E

Here is the FPGA code and PIC code:

```
module bowlinggame2(clk,reset,button,commux,spidata,segm,control,pinoutcome);
input clk, reset, button;
input spidata;
output commux;
output [3:0] pinoutcome;
output [6:0] segm;
output [5:0] control;

wire [3:0] pad, s1, s2, s3, s4, s5, s6,
  mux1, mux2, mux3, mux4, mux5, mux6;
wire slwclk;
wire commux,spidata;
wire [3:0] tryone;
wire [3:0] trytwo,hundreds1,tens1,ones1,hundreds2,tens2,ones2,framecount;
wire playerselect;
wire [1:0] enablescore;
wire [3:0] score1,score2;
wire [5:0] location;
wire [2:0] speed;
wire [8:0] finalscore0,finalscore1;
wire [7:0] spioutput;

clkdiv clk1(clk, reset, slwclk); //instantiation of the clock divider

pulse enable(clk,reset,button,commux); // mux signal enable

multiplex m1(hundreds1, s1, commux, mux1); //used to mutiplex the hex display

FF reg1(slwclk,reset,mux1, s1);

multiplex m2(tens1, s2, commux, mux2);
FF reg2(slwclk,reset,mux2, s2);

multiplex m3(ones1, s3, commux, mux3);
FF reg3(slwclk,reset,mux3, s3);

multiplex m4(hundreds2, s4, commux, mux4);
FF reg4(slwclk,reset,mux4, s4);

multiplex m5(tens2, s5, commux, mux5);
FF reg5(slwclk,reset,mux5, s5);
```

```
multiplex m6(ones2, s6, commux, mux6);
FF reg6(slwclk,reset,mux6, s6);

hexdisplay seg(slwclk,reset,s1,s2,s3,s4,s5,s6,segm,control); //used to designate hex display

shiftregister shiftregister(clk,spidata,spioutput); //shifts input serial data

//takes the spioutput data and makes into one-hot for the location and speed
onehotdecoder onehotdecoder(spioutput,location,speed);

firstturn firstturn(clk,reset,enablescore,speed,location,tryone,score1); //first attempt

secondturn secondturn(clk,reset,enablescore,tryone,speed,location,trytwo,score2); //2nd
attempt

//selects player and assigns enablescore to value 0,1,2
playerselect playerselect1(clk,reset,commux,location,speed,
framecount,playerselect,enablescore);
//scores
bowlscore bowlscore(clk,reset,playerselect,enablescore,score1,score2,
finalscore0,finalscore1);

//player 1
conv conv1(clk,reset,finalscore0,hundreds1,tens1,ones1); //converts to decimal

//player 2
conv conv2(clk,reset,finalscore1,hundreds2,tens2,ones2); //converts to decimal
                                                                    //outputs pin combo
to the PIC

pins pins(clk,reset,enablescore,tryone,trytwo,pinoutcome);

endmodule


/*
Verilog Code for the divided Clock
*/
module clkdiv(clk, reset, slwclk);
   input clk;
   input reset;
   output slwclk;

 reg [7:0]count;
```

```
always @ (posedge clk or posedge reset)
if (reset) count <= 0;
else count <= count + 1;

assign slwclk = count[7];
endmodule
```