Final Project Report

Virtual Basketball: How Well Do You Shoot?

Group #3: Chun Li & Jingwen Ouyang
May 17, 2007

6.111 – Introductory Digital Systems Laboratory
Primary TA: Javier Castro

ABSTRACT:

Inspired by our team's athleticism, Virtual Basketball allows the player to practice shooting anywhere and anytime, without the use of a ball. The goal is to create a final project that will take inputs from the accelerometer attached to the user's hands and display the projection of the ball after calculating release velocities. In addition, visual aesthetics like the ball's shadow, a score box, and bouncing would be great. The system is made of two main components: the accelerometer to calculate initial velocities and the cartoon image output simulating the shot. Using an ADXL330 3-axis accelerometer, the initial velocities of the ball can be calculated. We were able to calculate velocities from the accelerometer and display a screenshot with the court, ball, score box, and beaver "player" by reading the images from the ROM. A test screen composed of only the ball was able to take input velocities calculated from the accelerometer and display movement of the ball given those initial velocities. Unfortunately, given the time constraint and technical difficulties, the final system was not completely integrated. Nonetheless, it was an amazing educational experience.

TABLE OF CONTENTS

LIST OF FIGURES 2

LIST OF TABLES 3

I. INTRODUCTION 4

II. VIRTUAL BASKETBALL GAME OVERVIEW 4

III. ACCELEROMETER INPUT 5
3.1 Analog-to-Digital Interface 5
3.1.1 Parallel Interface 5
3.1.2 Serial Interface 6
3.2 Time Dividers 6
3.3 Calibration 7
3.4 Testing, Errors, and Debugging 7

IV. VELOCITY CALCULATION 7
4.1 Accumulator 7
4.2 Velocity 8
4.3 Toss Finite State Machine 8
4.4 Conversion 9
4.5 Testing, Errors, and Debugging 9

V. GAME DISPLAY 9
5.1 Game Logic 10
5.2 Display Field 13
5.3 Testing 15
5.4 Results 15

VI. CONCLUSION 16

LIST OF FIGURES

Figure **Page**

1. The Nintendo Wii 4
2. Big Picture Block Diagram of the Overall System 4
3. Picture of Device Used to Attach the Accelerometer to the Hands 5
4. Picture of the Fully-wired Circuit 5
5. State Transition Diagrams for the Parallel Interface Module. 6
6. Accumulator Shift Registers Used to Take an Average of Four Acceleration Points 7
7. Oscilloscope Waveform of Hand Toss Starting with the Hands Backwards . . 8
8. Proposed ideal *Game Display* screen shoot 10
9. Overview of *Game Display* block. 11
10. Overview of *Game Logic*. 11
11. Graphic example of calculating the intersection of a line and plane 11
12. Relationship between the center of the rim and the center of the ball . . . 12
13. 3D and 2D coordinate set up 13
14. Overview of the *Display Field* Block. 13
15. Score box without number (a) and score box with scores (b) 14
16. Screen shots of the VGA display 16

LIST OF TABLES

Table **Page**

1. Conversions: mm/sec to pixels/ frame 9

VIRTUAL BASKETBALL: HOW WELL DO YOU SHOOT?

I. INTRODUCTION

The introduction of the Nintendo Wii revolutionized the gaming industry by being the first console to use accelerometers, gyroscopes, and infrared sensors to communicate between the player and game. Accelerometers measure the acceleration of the player's hand that holds the stick. Gyroscopes read the tilt and rotation of the motion. And the infrared transmitter and receiver communicate the data and locate the player's relative position.

Figure 1: The Nintendo Wii

The Wii allows players of all ages to play games ranging from virtual golf, tennis, to Zelda and Spiderman. Its wireless controller can function as a pointing device such as gun or a tennis racket. It physically involves the user, such that researchers even claim that it is a good, fun weight-loss mechanism. It has truly changed the future of video games and opened new doors with its innovative use of accelerometers and gyroscopes.

II. VIRTUAL BASKETBALL GAME OVERVIEW

The idea behind our final project is to build a virtual basketball system that simulates a player shooting free throw shots. The Virtual Basketball system has two main components shown in Figure 2, one that calculates the velocity given inputs from an accelerometer and another that displays the projectile motion of the ball via VGA given initial velocity inputs.

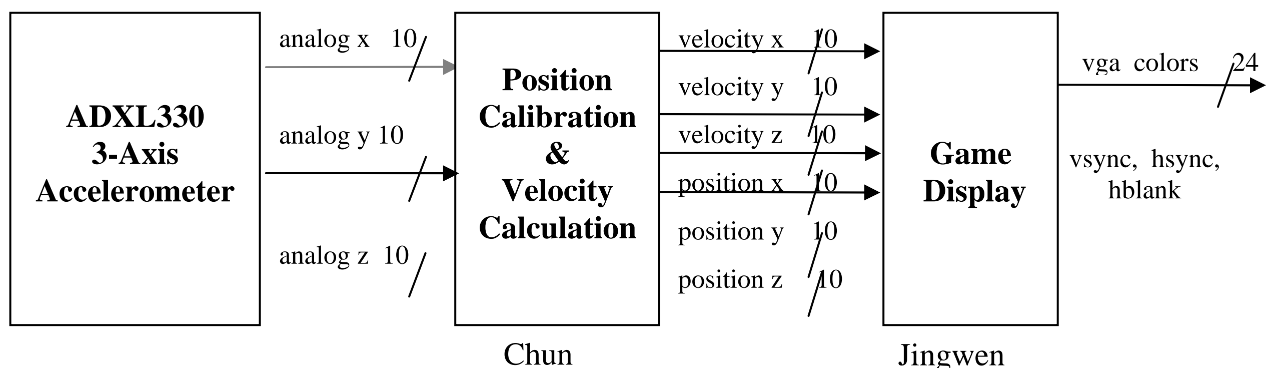


Figure 2: Big Picture Block Diagram of the Overall System

Ideally, the basketball game would involve the player starting in set position. As he/she tosses the imaginary ball, an almost-real-time display of the ball's projected path would show up on a basketball court angled at a perspective based on readings from the accelerometer attached at the hand. If the player makes the basket, then a celebration picture shows. Should the player miss, the ball will bounce off some surfaces and come to a rest. Refer to the Game Display for more details.

It turns out that calculating the velocity from the accelerometer outputs is not easy. The difficult lies in adjusting for noise in the hands as they move. Also, the rotation of the hand presents a much more complicated mathematical problem than originally anticipated. As a result, our actual

working system is a simple game where the player pushes his/her hands forward to move a ball according to the force of the push.

The accelerometer is sewn onto the fingers of the Velcro-made device show in Figure 3**.** It is guarded as much as possible with electrical tape to avoid static shock. Its location on the fingers allows the most stable measurement of acceleration without excessive noise.

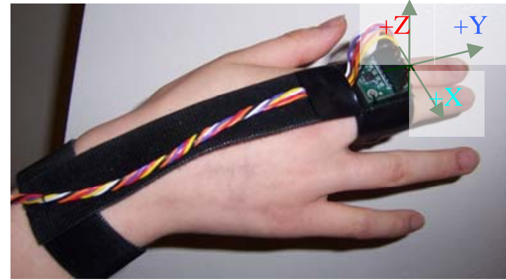


Figure 3: Picture of Device Used to Attach the Accelerometer to the Hands. The positive axes are indicated as above.

III. ACCELEROMETER INPUT

The main device used in this project is the ADXL330 3-axis accelerometer (on an evaluation board), the same accelerometer used in the Nintendo Wii. It is powered with the 3.3V voltage source on the lab kit such that the output voltage ranges from 0 to 3.3V. The ADXL330 has a measurement range of ±3 g minimum and a sensitivity of approximately 330 mV/g (for a 3.3V operating voltage). In order to actually calculate velocity using the FPGA, the voltage outputs from the accelerometer needs to be passed through analog-to-digital converters. For a full scale block diagram, please refer to Velocity Block Diagram in Appendix A. The lab kit 27Mhz clock supplies all the modules with the *clk* signal, and *reset_sync* also goes to all modules.

3.1 Analog-to-Digital Interface

ADCs are made with serial or parallel output ports, each with advantages and disadvantages. The advantages of serial ports are that it requires minimal wiring. However, it also requires precise timing guidelines such that the correct outputs bits are registered out from the ADC. Parallel ports output all the bits at once so that the data can be read all at once as soon as it's ready. However, lots of wires are required to represent each bit.

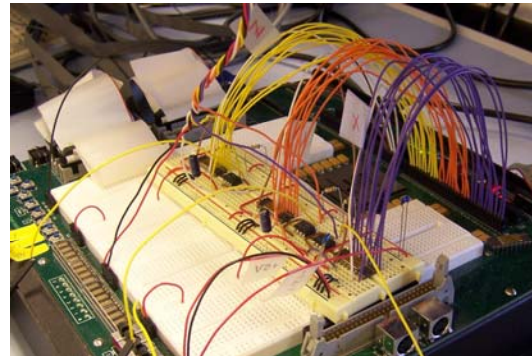


Figure 4: Picture of the Fully-wired Circuit

At first, the 10-bit AD7810 ADC was chosen for its serial output ports, optimal analog input range, and ease of usage (dip pins). However, due to ordering issues, the chips did not arrive on time. Instead, parallel 10-bit AD571 ADC chips were used in our circuit. Each digital bit represents 10 mV (important for conversion later). Each of the 3 axes to the ADCs first connects to an operational amplifier before being fed into the ADC. This is because the accelerometer cannot drive such a high load as the ADC, so the op-amp provides a high impedance input resulting in near zero input current. This way, the outputs of the ADC are more representative of the actual voltage outputs from the accelerometer. The final circuit can be seen in Figure 4.

3.1.1 Parallel Interface

For reference to the pin connections, please see the data sheet for the AD571. Pin 14, 15, 16 are connected to ground. *V-* (pin 12) is connected to +5V and *V+* (pin 10) is connected to -12V, both from the lab kit. The control signals DATA_READY_bar (pin 17) and BLK/CONV_bar (pin 11) are controlled by the parallel interface module connected through the lab kit.

The module has the inputs *clk* and *reset* (as with all other modules), and *div_convst* (the sampling clock), *data* (10-bit data from the ADC), and *data_ready_bar* (on active low, signals data ready to be read). The outputs are *convert_bar* (goes low to signal conversion start) and a 10-bit acceleration output, *acc.*

Using the timing and control sequences diagram from the specifications of the AD571, the transition state diagram in Figure 6 shows the signals used in the parallel interface to control the ADC. There are many waiting periods in this modules, thus an internal counter is used to count all the waits because the times do not overlap.

On reset, the module initializes in the BLANK state. In BLANK, it waits for the counter to reach BLANK_TIME (2 us). Once the counter equals BLANK_TIME, the *convert_bar* signal goes low to signal the ADC to start a conversion and the state transitions to CONVERTING. In this state, the module wait for the AD571 to tell it that data is ready by making *data_ready_bar* low. Once it gets this signal, the next state is WAIT_ACTIVE, where the module waits DATAREADY_TIME (500 ns) before the data is active for reading. Once available, the digital *data* gets stored into the output *acc*. Finally, in the DONE state, the module waits for a sampling clock high (described in Time Dividers) to begin another blanking and conversion.

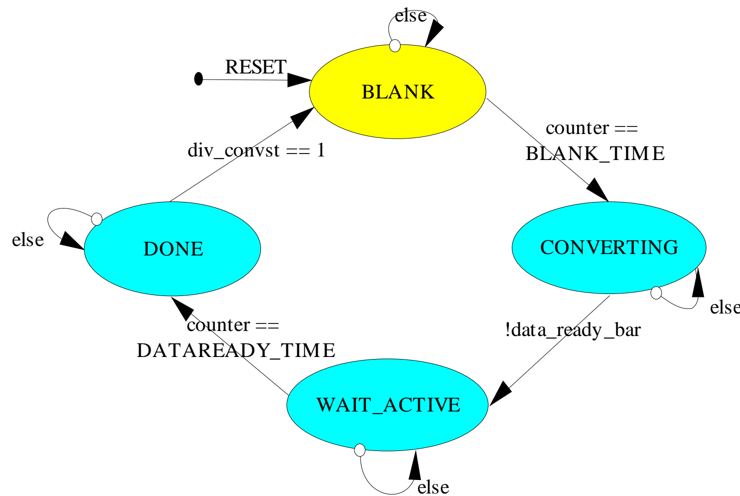


Figure 5: State Transition Diagrams for the Parallel Interface Module. The module outputs data conversion signals to the ADC and reads data when the ADC signals *data_ready_bar* low.

3.1.2 Serial Interface

As said earlier, the serial interface was built in preparation for use with the AD7810 ADC. It works differently from the parallel interface described above, but the timing concepts are the same. We will not go into detail about the timing and state transitions, but the module transition state diagram can be found under Serial Interface State Transition Diagram in Appendix B, as well as the Verilog module.

3.2 Time Dividers

According to specifications, the X- and Y-axis of ADXL330 have a maximum sampling frequency of 1.6 kHz, and the Z-axis has a maximum operating frequency of 550 Hz. To meet this requirement and still sample at an appreciable resolution, two clock dividers were created to generate a 1 kHz and a 500 Hz signal. The outputs of these two time dividers *div_1khz* and *div_500hz* are used to initiate analog-to-digital conversions in the parallel interface and to time sampling rates in the accumulator and velocity modules.

3.3 Calibration

Because each device has different zero gravity bias and sensitivity levels (ranging from 1.2 to 1.8 V), it is necessary to calibrate those values specific for each device each time the bit file is programmed onto the FPGA. The stored values can only be reset with *calib_reset* (button1) and the values of zero-g bias can only be changed if the user enters calibration mode (*switch[0]* = 1). Once in the calibration mode, the user must perform a sequence of events to calibrate zero-g values, from which a sensitivity can also be calculated. The typical values of 1.66V for zero-g bias (every axis) and 33mV sensitivity are set as default on calibration reset.

After the user enters calibration mode, to calibrate the zero-g bias in the X- and Y-axis, lay the chip flat on a surface such that its –Z axis points down (see Figure 3 for axis). This ensures that the gravity does not act on either the X or Y-axes so a zero-g bias value can be measured. Pressing button2 (*storeXY_sync*) stores whatever *accX* and *accY* was inputted at the time of the button press into the registers *zero_gX* and *zero_gY* respectively. At the same time, *accZ* is stored in a temporary variable to be used to calculate sensitivity. Next, position the chip vertically such that the +Y axis points towards gravity. In this position, the X and Z-axis display zero-g bias. Pressing button3 (*storeZ_sync*) places the Z-axis bias into *zero_gZ*. Lastly, by pressing the *enter* button, the user tells the module to calculate sensitivity, given two different Z-axis values. Note that sensitivity can only be changed if both store buttons are pressed, or else it refers to default. Thus, the final 10-bit outputs *zero_gX*, *zero_gY, zero_gZ* and *sensitivity* are calibrated according to the specificities of the chip.

3.4 Testing, Errors, and Debugging

The main tools for testing used were ModelSim and the logic analyzer. It was very important to debug each component individually and test for all scenarios before implementing together. This saves a great amount of time, especially when the code became long. Some sources of error included connecting the user I/O output as an input. This resulted in zeros being read in as data. Device errors were another major issue. One of the AD571 chips was converting incorrectly. This mistake was found using the logic analyzer. The data had seemingly random 512 values inserted in between the 200s, which signaled that either the bits were connected wrong or that the device was faulty. Also, throughout the project, a total of 3 accelerometers were used, because two mysteriously stopped functioning, most likely due to static shock.

For all ModelSim waveform simulations, please refer to Appendix C.

IV. VELOCITY CALCULATION

4.1 Accumulator

Due to possible noise from the accelerometer readings, the accumulator module takes in raw acceleration inputs from the ADC and calculates an average of four acceleration inputs. The key inputs include the sampling clock (*div_1khz* or *div_500hz*) and an acceleration. This is done with four 10-bit registers. Each time the

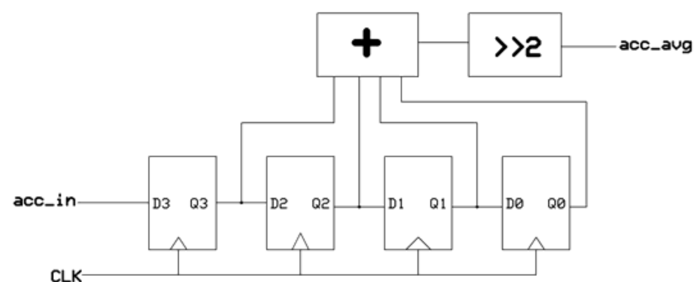


Figure 6: Accumulator Shift Registers Used to Take an Average of Four Acceleration Points

sampling clock goes high, it indicates that data is available and the module shifts all the values to the next register. The beauty of choosing only four registers is that to take an average, all that needs to be done is to shift 2 bits to the right and sign extend the most significant bit. Thus, at any time, the 10-bit output *acc_avg* is used in the other modules. This module is instantiated three times, one for each of the axes.

4.2 Velocity Calculation

By playing with the accelerometer and looking at the outputted oscilloscope waveforms, it can be assumed that at the time of release, the velocities should be traveling at it peak. This is equivalent to taking the area under the acceleration curve until a maximum or minimum velocity is reached. For example in Figure 7, this corresponds to the time the blue curve falls below its starting baseline to the time it returns back to the baseline. The remaining waveform is simply the accelerometer decelerating and can be ignored.

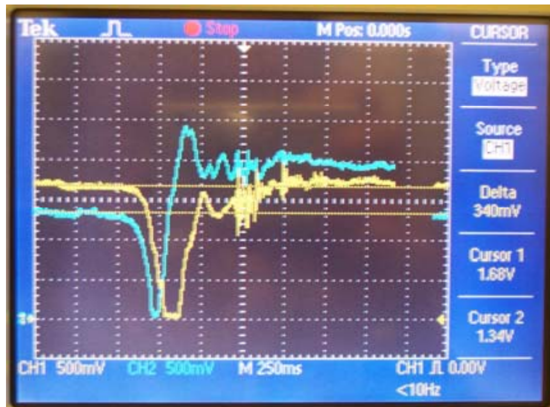


Figure 7: Oscilloscope Waveform of Hand Toss Starting with the Hands Backwards. The Z-axis is blue, and the Y-axis is yellow

For the velocity module takes as inputs the sampling clock, a delta time (in milliseconds), the averaged acceleration, a reference velocity, sensitivity, and a start signal. It outputs a final 26-bit signed velocity in mm/sec. This module is equivalent to an integration module.

The module holds the output velocity at zero until it receives a *start* signal from the Toss module. This signals that the player is ready to shoot, so the module can start computing the velocity. It looks at the input *acc_avg* and compares the value to *acc_ref*. If the average acceleration is above the reference, then add the difference to the final velocity, else it subtracts the difference (equivalent to negative velocity). This output has units of mV. In order to calculate the velocity, CoreGen multipliers and dividers were generated to divide the velocity in mV by the sensitivity (mV/g) and to multiply by time and gravity constants to convert velocity in understandable units.

4.3 Toss Finite State Machine

This module is used to start the game. Based on the input acceleration, it determines which starting position the player's hands are in by comparing the input acceleration to a reference velocity. If the player holds his/her hands steady for one second at the reference acceleration (plus or minus some offset to account for minor hand movements), then an led light comes on to tell the player that he/she can begin the toss and a start signal gets sent to the velocity module to begin integrating velocity. There are currently three possible starting positions. The player can start with 1) hands parallel to the ground, 2) vertical to the ground (as if pushing a wall), or 3) flipped back as if about to throw. These are simplified start modes so we can test the velocity component.

Due to mismatching accelerometer axis to real axis, it is necessary to hard code the different scenarios. If the player's hands are in start position (1), then the velocity reading from the Y-axis should be assigned the *release_velY*. If in position (2), then the *release_velY* is now the Z-axis

reading from the accelerometer. Finally, the last position tries to integrate both a *release_velY* and *release_velZ* by using velocity readings from the Z-axis alone. Unfortunately, this mode never quite worked as expected. The reason these axis must be adjusted is because once the accelerometer is attached the hand, so as the hand rotates, so do the accelerometer axis relative to the real axis that is used in the display module. It quickly became very confusing.

4.4 Conversion

The conversion module is simple. It takes as inputs *clk, reset*, and a 15-bit *input_vel* (measured in mm/s) and converts the input velocity to a 10-bit output velocity in pixels per frame clock. Doing some back-of-the-envelop conversions with the conversions in Table 1, the math simplifies to dividing the input by 800 to obtain an output in pixels per frame. Using CoreGen to generate a divider with a 15-bit dividend and a 10-bit divisor (all unsigned), a 15-bit quotient is produced of which the last 10 bits are used as the final output since the output velocity is on the order of 2 to 8 pixels per frame.

Table 1: Conversions: mm/sec to pixels/ frame

1 inch	25.4 = 25 + 2/5 mm
80 pixels	42 inches
1 second	60 frame clocks

4.5 Testing, Errors, and Debugging

The main method of debugging is with the logic analyzer. Some errors encountered included not correctly converting a signed two's complement number to a non-signed number. This resulted in velocity outputs many magnitudes larger than expected.

To test if the velocities make sense, the release velocity was approximated to be 7 m/s at an angle of 60 degrees. A break down of the velocity into x and y components show that Vx = 3.5 m/s and Vy = 6 m/s. Using this as a general guideline, I can look at my velocity outcome to determine if it makes sense or not. The outputs on the analyzer should display something in the order of 2000 to 7000 mm/s.

V. GAME DISPLAY

The goal of the *Game Display* block is to take the 3D initial positions and velocities of the virtual basketball as input, and display the cartoon version of a beaver (MIT's mascot) shooting a basketball with it's tail at the free throw line on VGA. A score box is also displayed to record the player's score (Figure 8).

Figure 8: proposed ideal *Game Display* screen shoot

Similar to the Pong Game that was implemented in lab 4, the *Game Display* block consists of the following internal modules (figure 9): *DCM; Debuncer; VGA controller;* a *Game Logic; Display Field*. The VGA display for this project is 640x480, 60 Hz. Because the *DCM*, *Debuncer,* and *VGA controller* modules are the same as in lab 4, the modules that are discussed in this paper are *Game Logic* and *Display Field.*

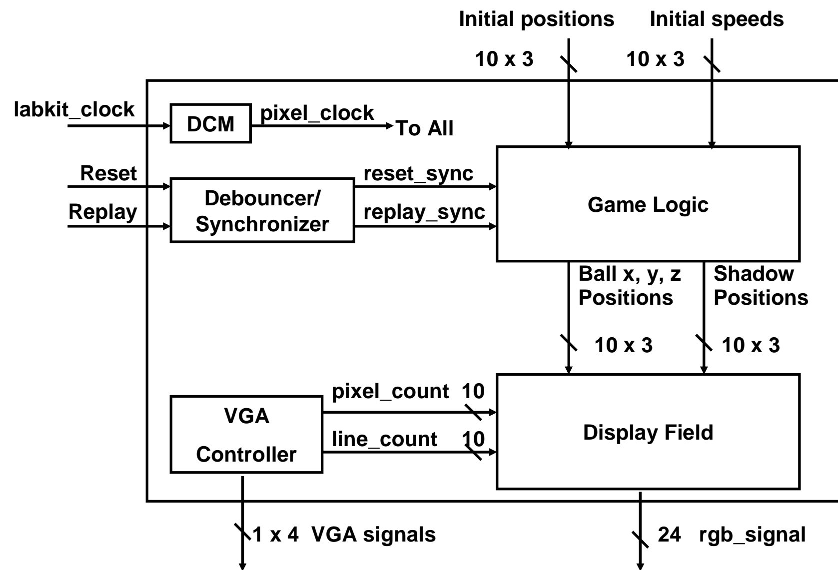


Figure 9: Overview of *Game Display* block

5.1 Game Logic

To display the basketball, *Game Logic* first computes where the ball is supposed to be in 3D at each frame. It then converts that 3D position into 2D. Figure 10 shows an overview of the *GameLogic*.

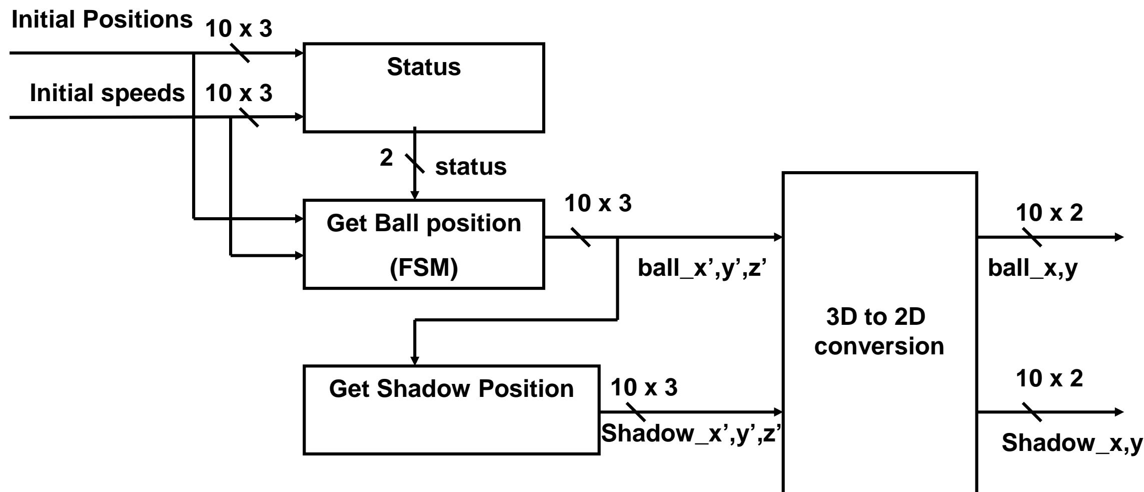


Figure 10: Overview of *Game Logic*

Once in the air, the ball may have a few different situations. For example, it may keep going; it may hit the rim, the backboard, the pole, or the floor; it may also go out of bounds; or it may go through the hoop and scores. A module *getStatus* is implemented using finite state machine (*FSM)* to check whether the ball hits any thing when transitions from the location in current frame to the one in next frame. This *FSM* always starts in the *idle* state (0) and stays there unless it gets a start signal, in which case it goes to the *getIntersectStatus* state (Appendix E.1). When in the *getIntersectStatus* state, the *FSM* checks to see if there is a potential of hitting anything and sets the control bits, such as *intersectWithRimPlane* to either high or low. Then the *FSM* transitions to the proper state according to those control bits. Take the expected path of the ball into consideration, the states from 2 to 5 (corresponds to the situations of hitting the rim, plane, pole, and floor) are treated with priority from high to low. In states 2 to 5, *getStatus* calls a minor FSM (Appendix E.2) *getIntersection* to check whether the ball indeed hit the desired object. Once the desired object is hit, the *FSM* sets the output *status* to the proper value and goes to the *getStatus1* (state 6); if the desired object is not hit, the *FSM* goes to a higher numbered state until it goes through all the rest states. The purpose of having the ostensibly extra states *setStatus1* and *ssetStatus2 are* adding delay to wait for the upper level module to turn off the *start* signal. Otherwise, the *FSM* continues to loop indefinitely.

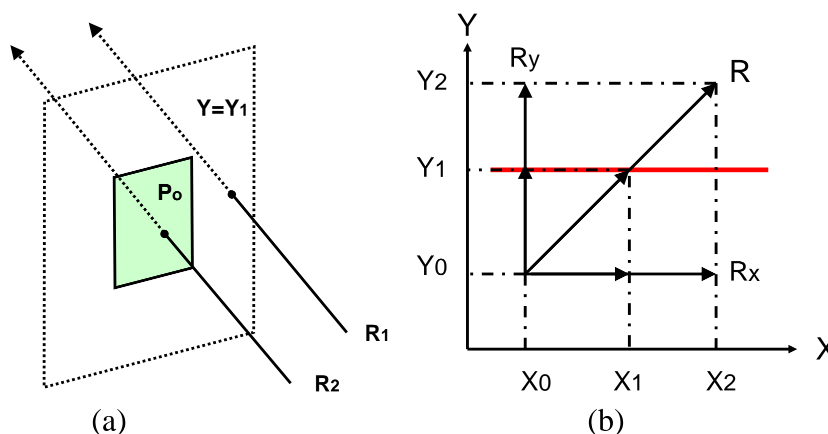


Figure 11: Graphic example of calculating the intersection of a line and plane

So, what does state 2 to 5 do? How does the *getIntersection* module determine if the ball is hitting any object? Take the situation of hitting the backboard as anexample. Suppose the backboard is in the plane $Y = Y_1$, as shown in figure 11 (a). Because the board is not infinitely large, simply checking the Y-axis values of the ball is insufficient. The x and z-axis values also matter. A solution to this problem is to simplify this 3D problem into a 2D problem. First, *getStatus* finds where the ball will intersect with the plane that the board is in, which is $Y=Y_1$ because the board is parallel to the plane Y=0. With known Y_0, Y_1, Y_2, X_0, X_2, Z_0, Z_2, the X, Z-axis intersection coordinate values X_1 and Z_2 can be easily computed due to the geometry shown in figure 4 (b). Last, *getStatus* checks whether the X, Z-intersections are within the range of the board by checking whether the X-intersection value is in between the X values of the board and whether Z-intersection value is in between the Z values of the board. Other obstructions in the court, other than the rim, are checked using the same method. However, because the rim is round rather than square, the algorithm for the rim is different from the others, if the vector length of the rim to the center of the ball is smaller than the radius of the rim minus the radius of the ball, the ball is inside of the rim; if the vector length of the rim to the center of the ball is bigger than the radius of the rim plus the radius of the ball, the ball is outside of the rim; otherwise, the ball hits the rim. (Fig. 12).

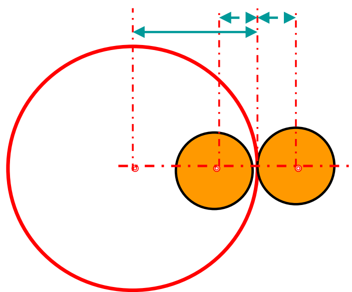


Figure 12: Relationship between the center of the rim and the center of the ball

With the information for the ball position, the top level *FSM, getBallPosition,* computes the position of the ball in the next frame. The logic for the *getBallPosition* is not complicated(Appendix E.3). The *getBallPosition* stays in *idle* state until *start* signal goes high. According to the information from the *getStatus* module, *getBallPosition* either goes to the *keepGoing* state directly, or go through a collision state to update the velocity. For the basic version, a few assumptions about collision are made to simplify the calculation. The first assumption is that energy is conserved at all times. Other assumptions are the ball is not spinning and there is no friction. With this simplification, in order to change the velocity of the ball, the only thing needs be changed is the direction in proper axis. .Ideally, it would be more realistic if there is less assumptions; unfortunately, there was not enough time to do such extensive calculations.

In addition, a simple module *getShadowPosition* is created to get the positions of the shadow. Again, it is assumed that the shadow has the same X, Y-axis values as the ball, but the Z-axis values is 0, which means the ball is vertically projected onto the floor.

So far, the positions are calculated in 3D. These positions need to be converted to 2D in order to display onto the VGA. From the 3D and 2D coordinate's relationship diagram (figure 13), the following converting equations holds:

$$X = X'\cos(a) - Y'\cos(b) + 235 \quad \text{and} \quad Y = Z' - X'\sin(a) - Y'\cos(b)$$

Based on those questions, *calculateFor2D* is created. The sine and cosine values are stored as fixed parameters in this module base on the angle of my court image. This module is implemented by instantiating the Xilinx built-in IP *divider* and *multiplier*. Because the *divider* takes M (about the order of the size of the dividend, which is 20 bit here) clock cycles to output the quotient, a *FSM*

with a counter is created to make sure that the module waits long enough to output the signal. The *FSM* transitions in a fairly simple manner; therefore, it is not explained.

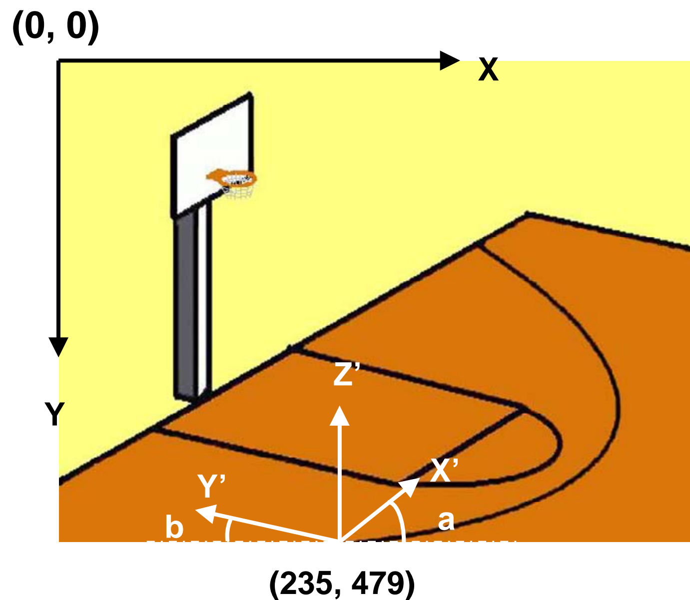


Figure 13: 3D and 2D coordinate set up

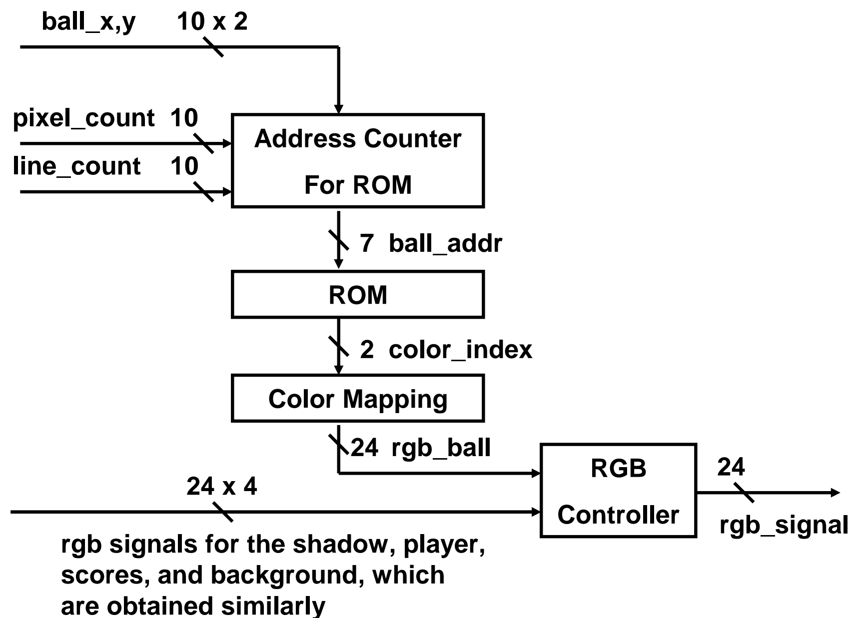


Figure 14: Overview of the *Display Field* Block

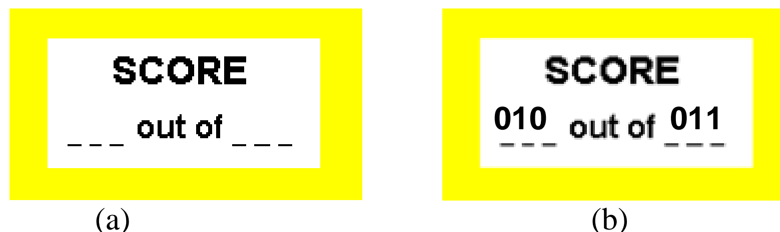
5.2 Display Field

With the computed positions of the moving objects (basketball and its shadow), the *Display Field* draws the images onto VGA screen. The process of displaying each image is about the same. An overview of the *display field* with draw ball as an example is shown in figure 14.In the beginning image information needs to be stored into a BROM. The *address counter for ROM* then computes from which memory address to read at each pixel that is being drawn. With the correct

color address, the BROM gives a 4-bit color index, which then goes through a *color mapping* module to be converted into a 24-bit RGB signal to specify the true color.

There are various ways to store an image onto a BROM. The best image format is BMP, which consists of header and color information. A 24-bit BMP file stores the exact RBG value for that pixel, which is exactly what is needed for this project. However, there are two problems are associated with this. First, the BROM is not big enough to store a 24-bit 640x480 image as the background. This 24-bit color image needs to be converted into a 4-bit 640x480 image， which takes 75 out of 144 available ROM blocks to store. Instead of the reading the color signal directly from the ROM, a 4-bit color index is read from the ROM. This is why the *color mapping* module is needed. Second, the color information is stored starting from the bottom right corner, which means the image is stored up side down. After researching the different methods, I determined that the best way is to use Matlab to read the convert the image into numbers; Matlab already has a built-in function to take out the header and store the image color information from top to the bottom into a big matrix. For this project, a solid model of the court is created using SolidWorks. A solid model of the court is needed for accuracy, can rotate freely, meaning that it allows viewing the court from another angle without further extensive math calculations. Then a JPEQ image of the solide model at a certain angle is created and converted into a 16-color (4bit) BMP image. For a better display quality, this image is edited in Photoshop and Paint. Finally, this edited image is read into Matlab as a big matrix. A script in Matlab is written and executed to read each element of the matrix and to convert that element into the correct format for the .coe file with the color index corresponding to each pixel on the screen.

After the image is successfully read into the BROM, the next task is to read the information out of the BROM, which means the BROM address needs to be computed for each pixel. For most of the images that are displayed in this project, the BROM address value is incremented if the pixel location on the screen is in range of that object. Otherwise the address stays the same until it gets reset at the end of each frame. The only module that uses a different algorism is for displaying the score. A screen shoot is shown in figure 15, where 010 is the total shoots made and 011 is the total shoots attempt. *Draw numbers* is interconnected with *getStatus* discussed earlier in the previous section. If the *status* is *score*, then the total shoots made and attempt are incremented; if the *status* is *miss,* only the total shoots attempt is incremented; other wise both total shoots made and attempt stays the same. In order to save BROM space, a single BROM is created with information for numbers from 0 to 9. To display 6 digits onto the score box, 6 address pointers are generated. For each number, 3 digits are controlled separately. Once the less significant digit transitions from 9 to 0, which is indicated by the BROM address pointer for that digit reaches the end and resets to 0, the next less significant digit increments, which is indicated by adding 96 (each digit is a 8x12 image) to the BROM address pointer for that digit. This method avoids complicated binary to decimal conversion.



(a) (b)

Figure 15: Score box without number (a) and score box with scores (b)

Because there are multiple images to be drawn on the VGA, along with the RGB signal for the basketball, there is also RGB signals for the court, the beaver, and the scoring box. How does the

VGA know which signals to draw? How does the VGA know that whether it is drawing a ball or a court, at a particular pixel on the VGA screen? The module *rgbController* is created to over lay the images properly. When an image is created as BMP file, it is saved as a square image. However, not all the objects are in rectangle shape; the ball, for example, is round. The *rgbController* needs a way to recognize the transparent area and soa color that is treated as "transparent" (pink for this project) is set aside. Transparent color is used to indicate that area should not be drawn. The module *rgbController* has priority over the ball, the beaver, and then finally the court. First rgbController checks to see at the pixel is currently being drawn on the screen (specifies by the signals *pixel_count* and *line_count* gives the location of the pixel that), whether the ball's color is "transparent". If it is not, *the RGB* color is draw for the current frame; if its color is "transparent", meaning a ball is not presented at that pixel, *rgbController* goes on to check whether the color of the beaver is transparent for the current pixel; lastly, if nothing else is present at the pixel location, the color of the court is drawn.

5.3 Testing

The modules in *game logic* are mainly calculations. The way I try to debug those modules were do simulations in ModelSim with sets of representative inputs. The outputs according to the inputs then were checked against the outputting wave form (Appendix D) from ModelSim. The resulting wave forms are attached in the appendix. Most of the results match my expect values. The part that needs further debugging for logic is the *getStatus* module. There are a lot more conditions to check, especially when the ball is around the rim area.

VGA take a lot longer to test, because it has to go through generating the lab kit every time. I also found that displaying the background court takes extra 15 minutes. To speed up the generation speed, I turned down *drawBackground* module, so that it doesn't have to go through the big BROM. There are still some error consists in the VGA display. A thing I noticed but never get the time to fix was that the VGA *delay* unit needs to be adjusted according to the speed of the other modules, especially the ones instantiates a *divider*.

To simulate at the top level, each block was instantiated in the labkit.v file. I found errors from the modules that seem to work individually. Unfortunately there wasn't enough time to debug all the errors.

5.4 Results

An ideal fully integrated system should be able to respond to the player's hand motion, detect, and compute the initial positions and velocities. With those initial values, the system then should compute the 2D displaying trajectory of the basketball and display the ball travel along the calculated path. Unfortunately, with the limited time, the integration of the *game display* unit wasn't quite done yet.

So far, most of the individual parts in *game logic* were working in ModelSim, and gives the right results. Figure 16 is a series of VGA display screen shoots of the *game display* documents the progress of the project. As you can see, figure 9 (b) shows that the VGA display was able to draw an object with a given coordinates. Figure 9 (a) through (b) shows that the *Display Fields* successfully overlaid the different images together with pre-defined "transparent" color. Figure 9 (f) is generated with a different *color mapping* definition, which includes color that is not defined in a 16 colored BMP file that was created by Paint. The file with new *color mapping definition* looks better than the 16 colored version. We are confident that with more time, we will be able to implement a fully integrated system.

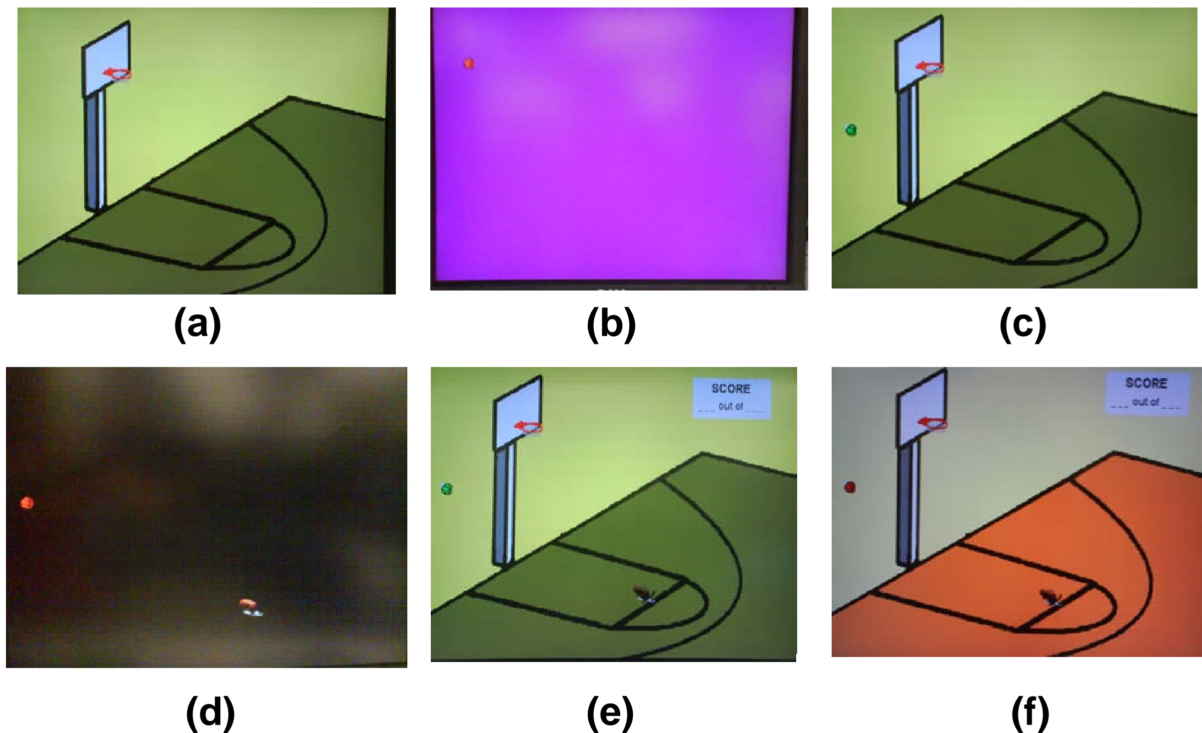


Figure 16: Screen shoots of the VGA display

CONCLUSION

Virtual Basketball aimed to produce a new, simple game similar to ones on the Nintendo Wii gaming console. The modular system involved designing and implementing two very different components. The system involved integrating acceleration readings from an accelerometer to get release velocities from which a ball's projectile motion can be calculated.

The end system had a working interface to obtain digital signals from analog acceleration inputs from the accelerometer and a functional velocity calculator. Given time, the velocity calculator could be improved to obtain more precise measurements of the actual velocity. The system could also successfully display all components of the game display like the background, ball, score box and a beaver player by reading the stored images from the ROM. Unfortunately, the final integrated system could not come together at the end.

For the future, it would be better to tackle the project in smaller, more manageable chunks such that there is a simple functioning model at the end. However, we still have many great ideas for how we can expand our project, such as adding gyroscopes to better measure the rotation of the hand or a wireless transmit/receiver system.

We would like to thank Gim, Javier, and the rest of the staff for their encouragement and technical help. Special thanks go to Howard Samuels of Analog Devices for helping us obtain the ADXL330 accelerometers. We learned a great deal about design, timing, time-budgeting, hardware interfacing, integrating, video display, and more, and greatly enjoyed working on the project!