Jin Hock Ong
Christopher L. Falling

Final Project Report: Motion Sensor Baseball Game

Abstract

For the final project, our group implemented a baseball batting practice game utilizing a motion sensor input. This sensor was implemented with a video camera that captures the movement of LEDs attached to a rod that represent a baseball bat. This project was divided into two main parts namely the motion sensor interface and the video and game interface.

The motion sensor interface converts the video input into digital signals using the internal AD7185 chip in the lab kit. The interface then decodes this information into meaningful signals that represent the position and movement of the baseball bat. It is the responsibility of the video and game interface to display these signals onto a VGA monitor and convert them to graphic representation of the baseball bat. Additionally video and game interface was responsible for converting the pitch control inputs to a series of images representing the baseball in flight as well as providing information to the scoring engine if a hit was scored.

While this baseball game was not fully implemented by the close of the project, a number of milestones were accomplished including the successful tracking of the rod and the synthesis of the video objects. The major milestone that we failed to accomplish was the capability to detect a hit and full configuration of the bat movement that would have lead to a playable game. Regardless we were able to draw a number of conclusions from the experience of implementing and debugging this project, including the need to provide for dynamic calibration between the sensor and the video interface.

1. Introduction/ Overview

Motion Sensor Interface

The purpose of utilizing motion sensing using a camera as an input to a baseball game is to allow users to simulate a hit at an angle that will ensure the ball to travel the furthest distance. Moreover, using a regular game controller will not provide the most realistic experience of swinging a baseball bat. While we initially decided to use accelerometers, the scale and packaging of the components we ordered proved to be too difficult to assemble onto a baseball bat representation. Thus, we conceived the idea of using a camera to capture the movement of a simulated bat

To ensure that we can recognize the movement of the rod, and not other moving objects, three LEDs are attached to the two ends. The lower end will be connected with one LED while the upper end will be connected with two LEDs. The upper end is connected with two LEDs so that we can observe whether the rod is being swung forward or backward, since the length between them will increase if they are swung towards the camera and vice versa.

The motion sensor interface will send the coordinates of the LEDs, and the speed of the swing to the 'Video and Game interface' so that user can interact with the baseball game by swinging a simulated baseball bat apart from generating a score if a hit is detected.

Video and Game Interface

The video for the game was synthesized for each frame based on the pixel being rendered. The eight bit VGA outputs for each color (Red, Blue, Green) and a visibility bit were generated for each game element at each pixel location. Based on this visibility information and the priority of the element, the appropriate color value was routed to the VGA encoder on the labkit.

The video and game interface will make use the coordinates sent from the motion sensor interface and translate the movement of the input rod onto the video screen. The game interface should also detect a hit if the user is able to swing the bat towards the ball at the appropriate time.

These two interfaces are then connected in the top module called labkit.v.

2. Motion Sensor Interface

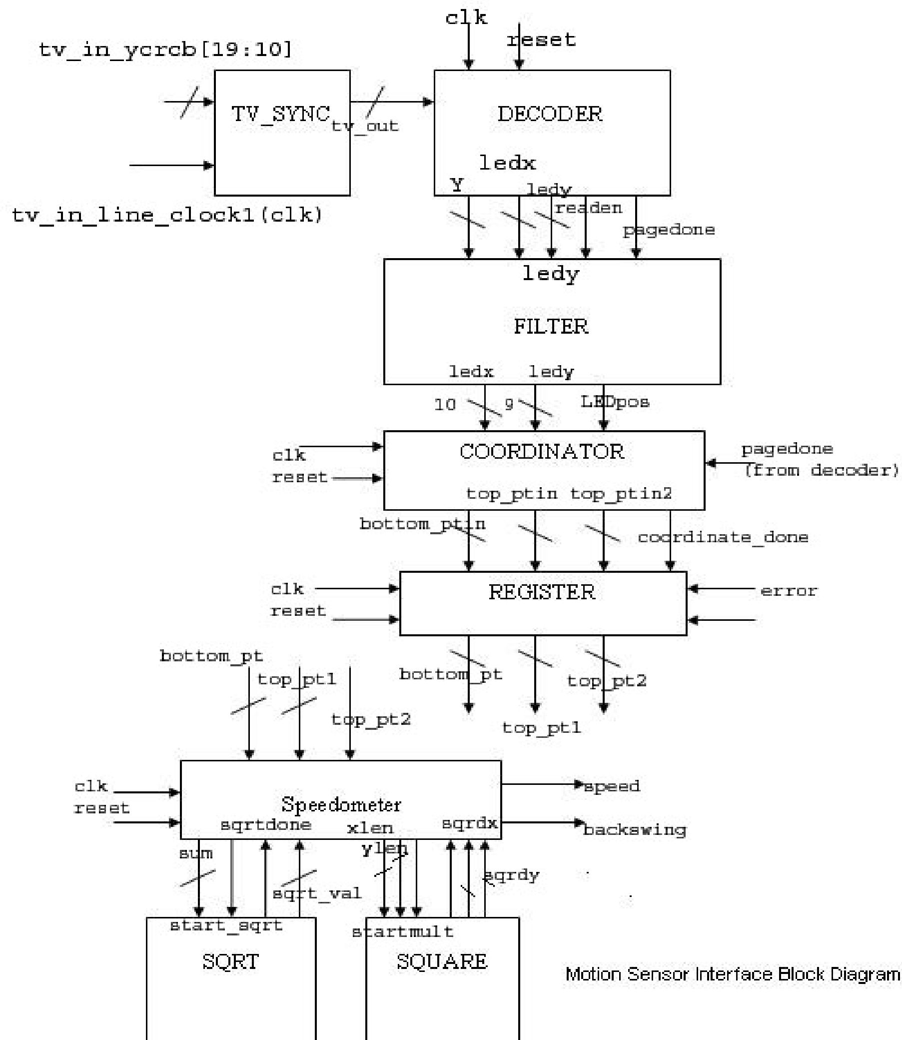
2.1 Motion Sensor Architecture

The motion sensor interface is consisted of seven modules: tv_sync, decoder module, filter, coordinator module, register, speedometer, square and sqrt.

Data is updated every time the AD7185 chip finishes decoding a page. A page refers to 525 horizontal lines of video data including blanking lines, while each line comprises 858 samples. It takes two clock cycles to decode one sample data.

Moreover, the motion sensor can be further separated into two parts, **Coordinator** and **Speedometer.** The **Coordinator** computes the coordinate of each LED while the **Speedometer** utilizes data from **Coordinator** to compute the speed of the swing.

Motion Sensor Interface Block Diagram



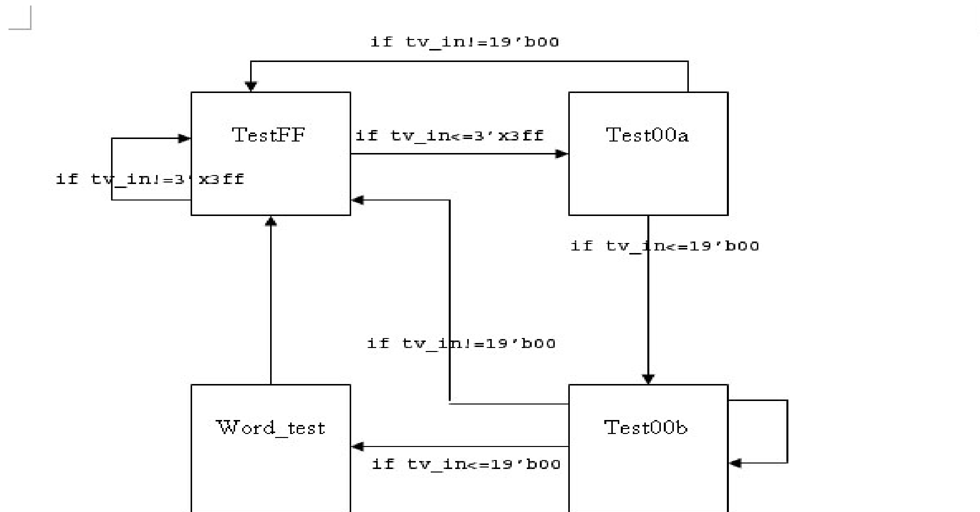
2.2 Motion Sensor Module Description

TV_SYNC

This module serves the purpose as a synchronizer due to the delay of the output data relative to the positive edge of the clock. Moreover, the 27MHz clock connected to this module is from the AD7185 chip, instead of the default 27MHz clock in the lab kit. This again ensures that the output is not shifted in anyway relative to the clock. The other input to the TV_SYNC module is the video data from the AD7185 chip.

DECODER

This module is implemented as a finite state machine. Moreover, this module also serves the purpose of a counter but the counter is reset and triggered based on TRS (time referencing signals) preamble of the video data. Below is the FSM diagram that explains the workings of this module.



Decoder FSM

At state *TestFF,* the Decoder module compares if the incoming video data is equivalent to x3FF. If it is, the FSM transitions to the next state, *Test00a.* If not, the FSM stays in the same state until the incoming video data is equivalent to x3FF.

At *Test00a* state, the FSM checks if the incoming vide data is equivalent to 0. If it is, the FSM transitions to the next state, *Test00b.* If not, the FSM will go back to the first state, *TestFF.*

The same thing happens in state *Test00b.* However, if the incoming video data is equivalent to 0, then the FSM will transition to the next state, *word_test.* At this point, we are certain that the video data submitted is the TRS preamble, since a sequence of x3FF, x000 and x000 is found.

Depending on the value of the incoming video data at this point, the signal *at_even_field,* and *blank_signal* can either be set to high or low. At this state also, *counterx* is reset to 0.

Countery may be incremented by two or reset to one or two depending on the values of the incoming video data and the value *at_even_field,* and *blank_signal* at that point.

Also, counterx will always increment by one every clock cycle regardless of the state the FSM is in. The value of counterx divided by two is equivalent to the data of sample n^{th} being sent, while countery represents the line n^{th} the AD7185 chip is decoding.

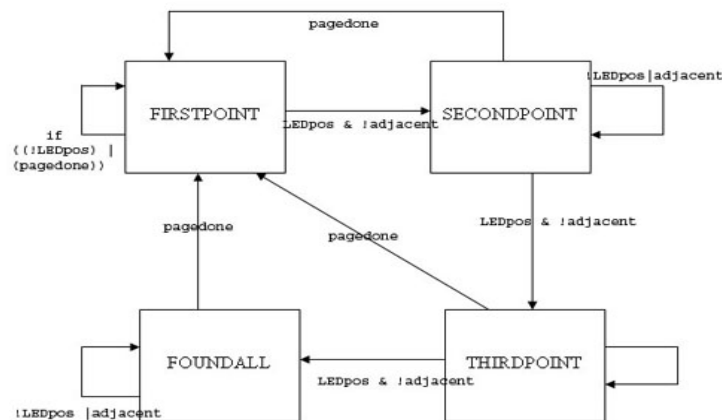
The main outputs of this module is the Luminance value of the video data, and the corresponding coordinates of the luminance values being sent.

FILTER

The filter module decides whether the luminance of the video pixel being received is high enough to be passed off as a pixel that represents the LED (since the LED's luminance should be higher than all other surrounding objects) If the pixel data passes the filter test, then one bit LEDpos signal will be triggered high or otherwise. All outputs of the filter are connected to the COORDINATOR module.

COORDINATOR

The COORDINATOR module decides which LED the filtered pixel belongs to. This module is also implemented as a finite state machine. The FSM diagram below will explain the workings of this module.

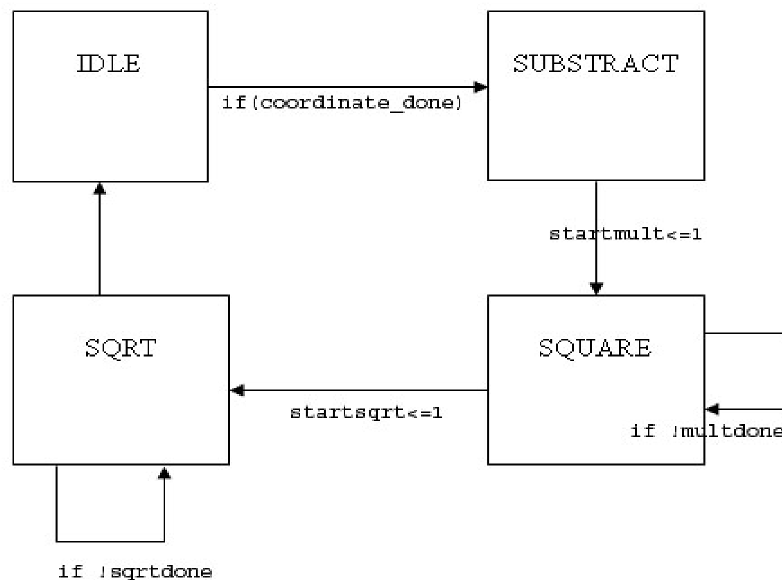


At the first state, FIRSTPOINT, the module will wait for a LEDpos signal. It will then randomly assign the register *bottom_point* the coordinate {ledy, ledx} sent from the FILTER module. It will then transition to the next state SECONDPOINT.

At both SECONDPOINT and THIRDPOINT, the states will first check if the next coordinate of the pixel representing the LED is adjacent to the already assigned point. If not, these coordinates will be assigned to *top_point1* or *top_point2* depending on their coordinates. This is an important process because we need to distinguish the points such that the speed of the swing can be calculated since we are always checking the difference of *top_point1* and *bottom_point* to determine the speed. All outputs of the COORDINATOR module are sent to REGISTER module and SPEEDOMETER module.

REGISTER
The register module is meant to store the value (coordinates) of the LEDs so that the game and video interface can translate the movement of the simulated to the video.

SPEEDOMETER
The speedometer is designed to calculate the speed of the swing of the simulated baseball bat. Since we are unable to compute the length between *bottom_point* and *top_point1,* this module is implanted as a finite state machine. This module is also connected to two helper modules that will compute the square root and the square of the values being sent from the SPEEDOMETER module. Below is a FSM diagram that explains the workings of this module.



The FSM starts at the IDLE state and wait for a start (*coordinate_done* signal from COORDINATOR) signal before moving on to the next state, SUBSTRACT. At this state, the differences of the x-coordinates and y-coordinates are calculated and *startmult* signal is set as high. The FSM will remain in the SQUARE state until the SQUARE module finish multiplying, where a high *multdone* signal will be sent from the SQUARE module. It will then transition to the next state, SQRT and remains there until a high *sqrtdone* signal is sent from the SQRT module.

SQUARE/ SQRT

The SQUARE and SQRT module is generated using the math functions found in CORE GEN. The SQUARE module performs multiplication operation on its inputs while the SQRT module computes the square root of the input.

2.3 Sensor Design Methodology/Decisions/Trade Offs

I originally planned to use accelerometers to track the moving rod. However, due to difficulties in soldering the actual accelerometers, we decided to use a camera to detect the movement of the simulated baseball bat. By doing so, the coordinates of the baseball bat will be more accurate. However, we will not be able to measure the actual speed of the swing. On the other hand, the speed of the swing is calculated by measuring the change of distance between the points of the baseball bat.

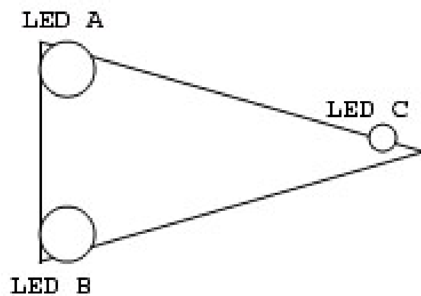


Figure 1

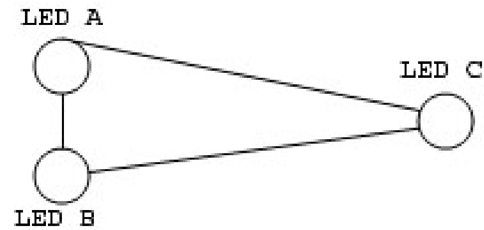


Figure 2

As we can see in figure 1, when the bat is being swung towards the camera, the distance between LED A and LED B is further compared to figure 2, when the bat is being swung away from the camera. On the other hand, the distance between LED A and LED C is further in figure 2 as compared to figure 1. From these facts, we can determine the direction of the swing, and the relative speed of the swing.

Moreover, the counting of the DECODER module is designed such that the counting will not be affected if the camera is suddenly turned off. This is done by keeping track of the TRS preamble at all times so that the counters will always be reset to the correct value after a few cycles (or after getting information through the TRS preamble).

While designing the COORDINATOR module, error handling is also taken into consideration to avoid infinite loops if no LEDs can be found on the screen. Not only will an *error* signal be triggered high, the old coordinates of the LED will be retained on the video output.

3. Video and Game Interface

3.1 Video block Diagram and descriptions

The video for the game was synthesized for each frame based on the pixel being rendered. The final resolution chosen was 800x600 @ 75Hz which required a 11 bit counter for the horizontal pixel count and a 10 bit counter for the vertical row count. These counts as well as the inverse of the vertical synchronization signal were passed to the picture generation module that determined the VGA color values for the pixel. Using the inverse of the vertical synchronization signal allowed for updates to occur outside the active video region. This constraint prevented any duplication or skipping of elements based on a changing value location during the rendering of a frame.

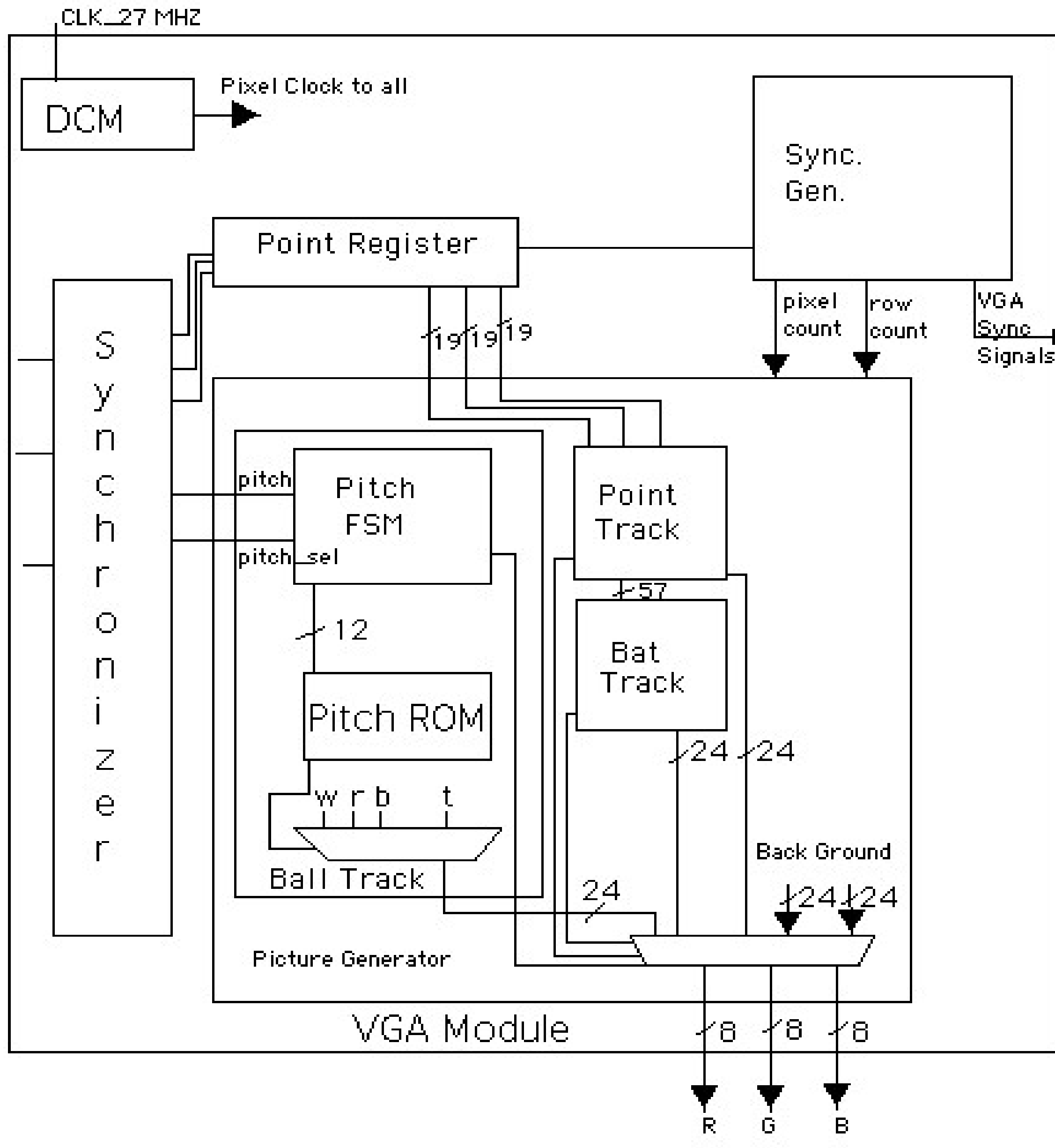


Figure 7 Block diagram for VGA Module

DCM

This is the digital clock manager provided by the xilinx FPGA used to implement the game. This functions as a clock multiplier which produces a 49.5 MHz pixel clock from the 27 MHz clock of the lab kit. Because the video is being synthesized on the fly this pixel clock is also used as the clocking signal for all of the VGA components.

SYNCHONIZER

This is a standard synchronizer used to bring signals from outside the pixel clock realm into synch with the pixel clock. In particular it is responsible for synchronizing the three 19 bit led coordinates as well as the pitch and pitch select game inputs from the lab kit in addition to the global reset signal.

POINT REGISTER

This module serves as a small register based memory that keeps track of the three LED points for the frame that is being generated. In particular the location of the points are only updated during the video vertical blanking period to keep one point from being drawn multiple times in a single frame. The picture generation component uses the data from this register to determine if the bat or point sub-elements are visible for a given point of the frame.

SYNC. GENERATOR

This module is responsible for generating the synchronization signals for VGA output. This is where the parameters for the horizontal and vertical blanking periods are defined as well as the logic to determine if these synch signals should be generated for a given pixel or row. While the counters needed to store and determine the actual line and pixel counts are actually defined in the overall VGA module, Figure 7. shows these counts being defined by the sync generator for diagram clarity.

PICTURE GENERATOR

This module is composed of a number of elements that are each responsible for generating the video output values for a given pixel on a line. Each sub-module takes the line and pixel counts as inputs and uses this to determine if a pixel should have a defined output, or be transparent. These visible flags are combined into the selection input of a multiplexer with priority logic incase multiple element overlays are visible. The output of this multiplexer is then used to drive the outputs of the VGA monitor.

POINT TRACK

This module was designed as a debug component, which would render the detected points of the bat on the screen as points. The visible range is computed by taking a three-pixel border from the center defined by each of the three points from the motion sensor. The bat track module was to use these points and potentially computed values such as the difference between frames to render the bat in a reasonable orientation.

BAT TRACK

This module was originally designed to take the points detected by the motion sensor and then render the bats current position. However due to time constraints a simple threshold based movement protocol was designed where the bat would move left, right, up or down based on if one of the LEDs fell within certain boundaries. However the only method of changing these values was in Verilog at compile time and this led to sporadic movement of the bat. A better solution would have been to include a calibration routine where the thresholds could have been set at "run" time by reading the actual measurements. As the points defining the bat are held constant from the frame, the bat itself is also only updated each frame.

BALL TRACK

This module is also composed of a number of sub modules to determine the image that makes up a baseball based on the current state of the game. It should be noted that the image of the baseball is defined to be off screen when the PITCH_FSM is in an idle state.

PITCH_FSM

This module is the finite state machine that controls the location of the pitch. Each type of pitch is defined by 10 states. With the four different types of pitches, this leads to a total of 41 states for pitches and idle. While not implemented, this FSM would have been extended to provide transitions based on if a hit was scored. The state transition diagram is provided in Figure 8. It should be noted that the transitions from the idle state occurs when the pitch input is high and this is the time that the values from the pitch selection switches are read. The remainder transitions are control by a timing module not shown on the block diagram. This module takes a update signal indicating that a frame is complete (the inverse of the vertical sync signal) and a number of frames based also on the type of pitch have been rendering before updating the image offset for the ROM address. The state transition diagram has been simplified to not show the output or the complete number of states. In particular it only shows the first set of transitions for the fast ball pitch and skips the intermediate transitions until the end of the pitches, FB_9 and FB 10 which returns the state machine back to IDLE after the update size signal is received. The state machine is defined in such a way that different timers controlled the size and update signals so that the ball could rotate at one speed and travel at a second. Each pitch contained a total of 10 images and thus states which defined the image selection and ROM address offset.

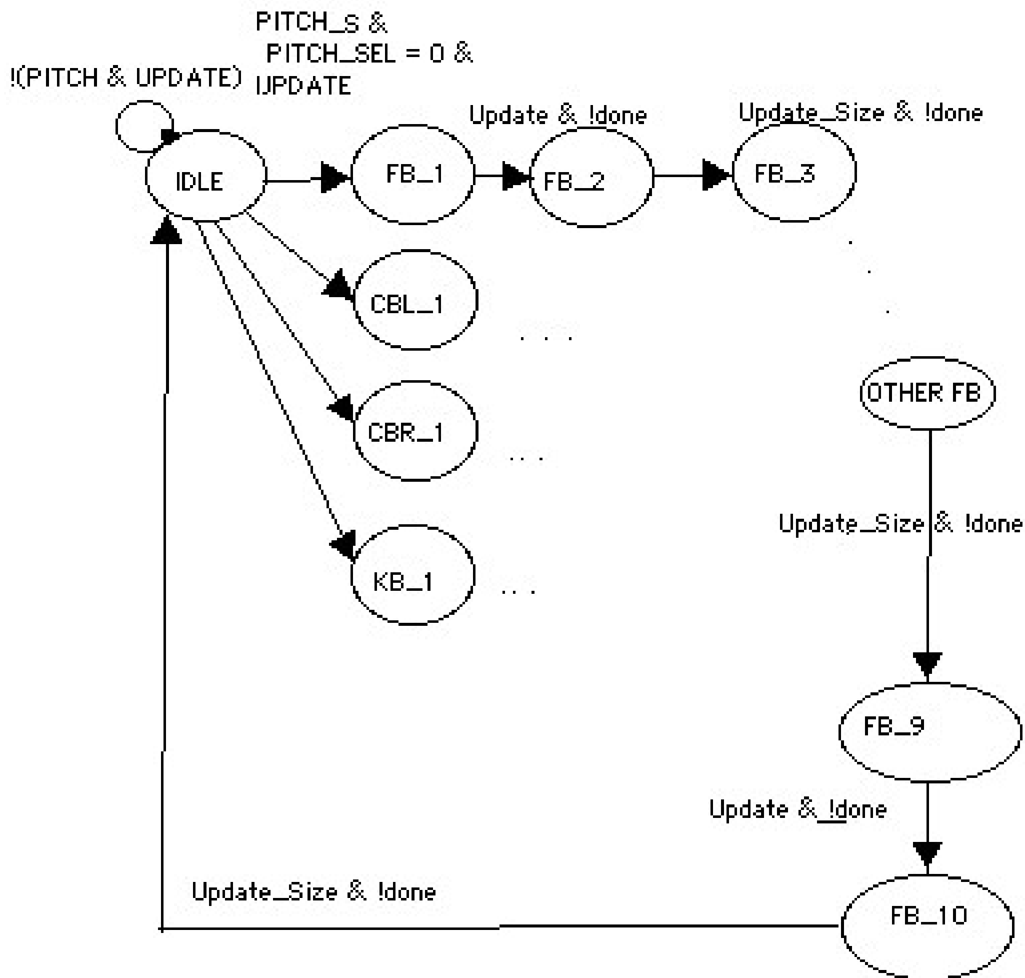


Figure 8 Simplified State transition diagram for Pitch FSM

PITCH_ROM

This module is a simple single port read only memory that was generated using the xilinix CORE GEN program. This initialization file defined the point that represents the colors of the baseball sprite at different sizes and orientations (based on stitch) pattern. Each baseball image is 24 x 24 pixels and is stored as a 2 bit value with the following interpretation.

ROM Value	Description
00	Transparent, override visible to false.
01	Ball mapped to white.
10	Edge mapped to black.
11	Stitch mapped to red.

Table 1 PITCH ROM values

Note that a value of 00 is transparent which overrides the visible bit and sets it to false, even though the current counts may indicate that the ball is in view. This design allows the memory to be read consistently for any size of ball, and for the background to potentially be presented as a complex image as opposed to a simple blue green separation. The values from the ROM are used to drive a palette selector and set the VGA outputs.

3.2 Video Design Methodology/Decisions/Trade Offs

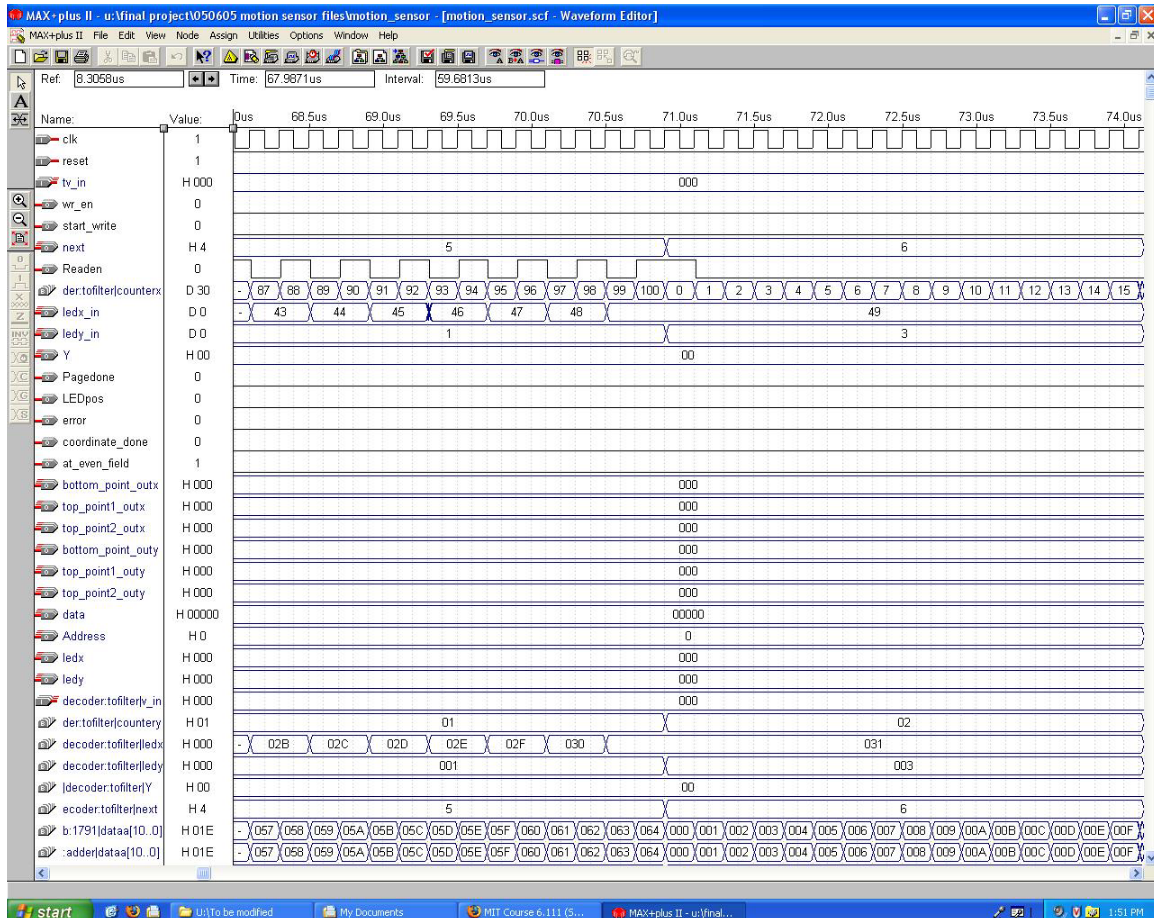
The synchronization of the VGA output is a well-understood process and the generation of the synchronization signals is not very exciting. However there existed a number of choices for rendering the active video region. The mechanism I used to synthesize the video on the fly worked given the small number of elements that I was working with but could have soon grown unwieldy. The complexity of this mechanism is further constrained by the speed of the pixel clock, which limits the complexity and routing of the logic generating the video images.

Another requirement was when to update the state for the video generation. In particular, it was necessary to prevent a single point from being skipped or duplicated in a single image. This could have occurred if the location of the point moved across the current pixel or row count during the rendering of the video screen. As such, the vertical synchronization signal was used as an update load enable for the state registers and to coordinate the interpretation of the points when moving the bat.

Other designs that were considered included using a multiple memory interface where a image would have been generated in a main memory and loaded into a video frame buffer cache during the blanking period. Another method would have been to use the external memory and a shared bus / major / minor FSM paradigm. While the video component would have still used the bus at 49.5 MHz, the game logic and state could have still been written using the 27 MHz clock of the FPGA. It would have been necessary to intelligently update the memory by writing only the changed pixels given the number of cycles for a given blanking period. This design could have been extended to include multiple frame buffers and switching between them if the vertical blanking period was insufficient to update the image. Another option would have been to store the image in an encoded format such as vertices and colors and have designed a video subsystem which would render these on the fly, yet this methodology was specified as a project in its own right by another group.

4. Testing/Simulations

The modules were tested separately with MaxPlus II simulation function and ModelSim simulation program before being connected with a top module. This top module was also simulated before being tested. Below is a screen capture of the simulation of the top module of the motion sensor interface.



The screen capture above shows the change of states apart from the increment of the counters of the motion sensor top module.

After programming the codes to the FPGA, the output of the important signals such as coordinates of the bat, an *error* signal was connected to logic analyzer to determine the possible bugs in the codes.

For system testing, the video input of the camera was passed though an analog to DV converter, which allowed the video signal to be viewed on an auxiliary computer before being passed onto the lab kit. This allowed us to see what the camera was seeing and view the results of the interpretation on the logic analyzer and VGA screen at the same time.

5. Conclusion

Motion Sensor Interface

While previous labs exercises are not similar to the final project implemented, the experience gained from the design and debugging process aid the implementation of the final project. By debugging the codes systematically, I was able to save a lot of time instead of random trial and error method.

Moreover, it is also essential to take noise into consideration when designing a project. A good project should also have some form of error handling to avoid the system from crashing or going into infinite loops. In this final project, the noise (bright objects) from the environment caused the LEDs to be poorly detected by the camera. This constrains the bat into a box in order to ensure the integrity of the signal being read.

Video and Game Interface

The use of synthesized video made it more difficult to provide a pretty image for the VGA screen however the main conclusion from the video subsystem is the need to provide run time configuration of the bat movement. For example the detected location for the bat depends not only on the position of the bat, but also the position and angle of the camera, thus static thresholds where insufficient for determining desired movement. Also some form of runtime configuration would have been needed if a more sophisticated bat rendering algorithm was being used due to the resolution differences between the video and VGA resolutions and the offsets introduced by the camera angles.

Acknowledgements/References

Nathan Ickes.
Initialization files for the AD7185 chip. Help in debugging the video decoding parts, and initial file for generating a VGA image.

Keith Kowal, Chris Forker and Charlie Kehoe:
Significant help in debugging the video decoding and encoding.