6.111 Final Project Report *Due: May 13, 2004*

Paratroopers

Tushara C. Karunaratna, Chun-Chieh Lin, George Heming
Project TA: Jia Fu Cen

Contents

List of Figures

1 Introduction

We present a hardware implementation of the classic Paratroopers video game, in which we have replaced the analog joystick with the motion and gestures of a human player in a playing area. Our design uses video input from a camera to detect the motion of the player and translates this motion into positional data to control the player's gun position. A player uses arm gestures to trigger the gun by raising and then lowering his arm. This game play concept was inspired by the recent use of video games as exercising aids. Our objective was to give the player a workout while enjoying a classic video game.

The Paratroopers video game has a simple objective, which is to destroy enemy helicopters, bombs and paratroopers as they appear on the screen. During the game the helicopters release bombs and paratroopers at random positions while moving across the screen. The player, represented by a gun sprite at the bottom of the screen, has the objective of shooting these enemy objects before they hit the ground or move out of the screen. The player is allowed to move left or right, in addition to shooting.

2 Design Overview

The system is partitioned into 3 functionally distinct parts, a Video Capture Unit (VCU), a Game Control Unit (GCU) and a Video Display Unit (VDU). The Video Capture Unit captures the motion of the player and processes this video to extract positional information, which is fed as input to the GCU to update the state of the game. The GCU is the game engine, which simulates the game play. It manages the tasks of creating and destroying objects, updating the position of objects, detecting collisions and maintaining score and player health information. The VDU displays the current state of the game as simulated by the GCU on a video monitor.

Game play begins with a calibration stage in which the video capture unit detects the height of the player and establishes a height threshold for indicating a shoot gesture. The calibration stage consists of 2 prompt screens on the Video Display Unit. The first prompts the user to stand in the playing area with arms lowered, while the second screen prompts for a raised hand. Each screen lasts five 5 seconds during which time the Video Capture Unit determines the appropriate values for the shooting threshold. The Video Capture Unit does this calibration by examining each frame of video to find the dark pixels corresponding to objects in the playing area and averaging over all frames the first dark pixels encountered. The calibration data is transmitted to the Video Display Unit.

Once the playing parameters are configured, the Game Controller proceeds to the level selection screen where the player can select the difficulty of the game play by moving left or right to respectively increase or decrease the difficulty of the game. The position of the player in the playing area indicates the level of difficulty. The player confirms his selection of a difficulty level with a shoot gesture. The confirmation causes actual game play to begin and it continues until the players health points decrease to zero. At this point, a game over screen is displayed on the Video Output. The player may restart the game by walking out and back into the playing area.

3 Camera input subsystem

This section discusses the implementation and design of input motion input part of the system.

3.1 Overview

The Video Capture Unit is responsible for translating the left and right motion and hand gestures of the player into positional information and a shoot signal, which are fed as inputs to the Game Controller. Capturing a monochrome NTSC video signal of the playing area and processing it to extract the relevant information achieves this function. In addition the VCU does the initial calibration to establish a height threshold to correspond to a shoot gesture. This calibration occurs at the beginning of each game and is necessitated by the need to adjust for different player heights.

The Video Capture Unit consists of hardware for sampling and storing the Video Signal and control and processing logic for processing the sampled video data. The hardware consists of an NTSC camera, an Analog-to-Digital converter, a monolithic sync separator and an SRAM to store the sampled video. The control logic is partitioned into four modules: a Controller, Digitizer, Calibrator and Processor. The Digitizer captures the digitized video from the analog-to-digital converter and stores it in the Line Buffer. The calibrator processes the data in the line buffer during the calibration stage to determine the shoot threshold and communicates with the Video Display Unit to display the computed threshold. The Processor uses the Line Buffer data to determine position and shoot gestures during game play and communicates this information to the Game Controller. The controller coordinates the operation of the Calibrator, Digitizer and Processor using sync information from the monolithic sync separator. Figure 1 shows the overall system organization of the Video Capture Unit.

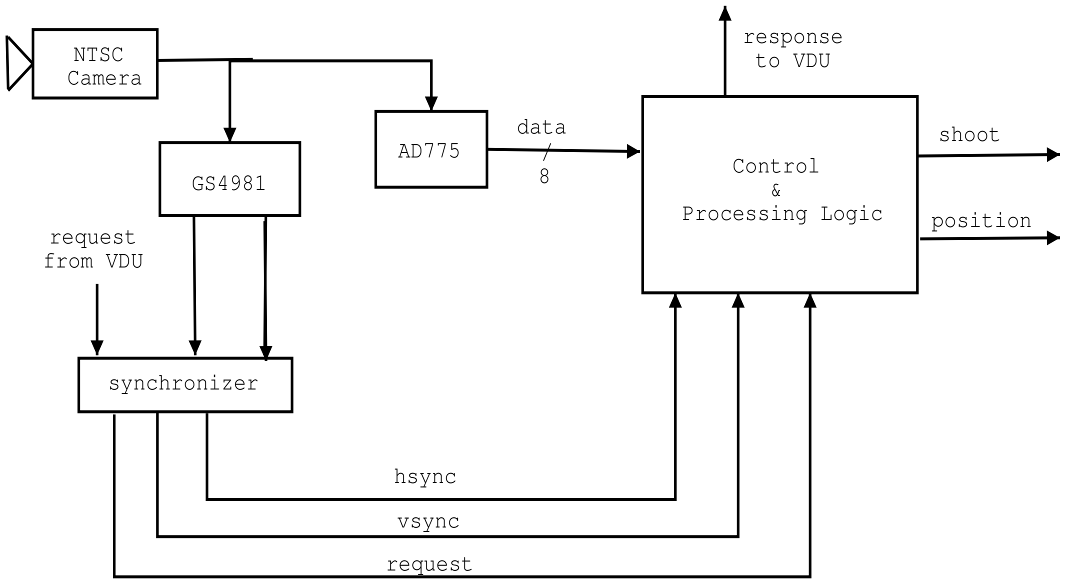


Figure 1: Block diagram of the Video Capture Unit.

3.2 Description of hardware used for Video Capture

GS4981 (Monolithic Sync Separator) The sync separator is responsible for extracting the vertical and horizontal sync signals from the NTSC signal from the video camera.

AD775 (Analog-to-Digital Converter) The AD775 is a flash based converter capable of sampling rates as high as 35Mhz at a resolution of eight bits. In this project, it was operated at the system clock frequency of 10Mhz, making it possible to take at least two samples of the video signal in determining the value of each pixel.

SRAM Line Buffer The Line Buffer is a 16-byte memory element used to hold pixel data corresponding to one horizontal line of the NTSC video signal. The choice of a 16-byte depth for the SRAM was dictate by our initial design decision to display the output of the game at a resolution of 128 x 96 pixels. Storing one bit of information per pixel, 0 corresponding to a value below the threshold and 1 for values above the threshold, it was sufficient to have 128 bits or 16-bytes of data per horizontal line. The 96 pixel vertical resolution also meant that it was possible to process a line of video between digitizing intervals. This eliminated the need to buffer an entire frame of video (approximately 500 lines), which would have been processed during the vertical blanking interval of the NTSC signal (approximately 22 lines) sample the video rather coarsely (roughly every fifth line)

3.3 Control and Processing Logic

Controller FSM The controller is a finite state machine (FSM), which takes as inputs the horizontal (h-sync) and vertical sync (v-sync) signals from the sync separator. Based on this sync information, the controller instructs the digitizer to start sampling the video.

Each frame of the NTSC signal consists of 525 horizontal lines and thus to achieve a vertical resolution of 96 pixels on the output display, it was sufficient to sample approximately every fifth line of the video signal.

The Controller FSM has 8 states and transitions between them based on the h-sync and v-sync input provided by the GS4981. On a reset, the controller transitions into the INITIALIZE state for one clock cycle where various counters are set to their appropriate initial values. After initialization, the FSM transitions into the WAIT_VSYNC_LOW stage, where it waits to detect the active low vertical sync signal, which indicates the end of a video frame. Once the end of a frame is detected in the WAIT_VSYNC_HIGH state by the v-sync signal going high, the Controller FSM waits in the DELAY_VBLANK state to account for the vertical blanking period of the NTSC. After the blanking period, the controller instructs the Digitizer FSM, based on the h-sync signal and the counter variable, to sample the active video on every fifth line after a horizontal blanking period of roughly 7us which is accounted for by the DELAY_HBLANK state. The start signal to the Digitizer is asserted in the START_SAMPLE state. The Controller remains in the WAIT sample state until it gets a done signal from the digitizer. Figure 2 depicts the state transition diagram of the Controller FSM.

Digitizer FSM The Digitizer FSM after initialization, triggered by a reset signal, in the INITIALIZE state waits in the IDLE state for a start signal from the Controller FSM. When the start signal is received, the Digitizer transitions into the SAMPLE1 followed by SAMPLE2 states where it reads in a digitized sample from the analog-to-digital converter. The average value of the two samples is compared with the external threshold input to determine the value of a corresponding pixel in the STORE_BITS state. In the STORE_BITS stage, the Digitizer FSM keeps track of the pixel positions in the video and assembles them into 16-bytes, with the LSB of the nth byte representing the 2n horizontal pixel of each line. The Digitizer transitions between the STORE_BITS

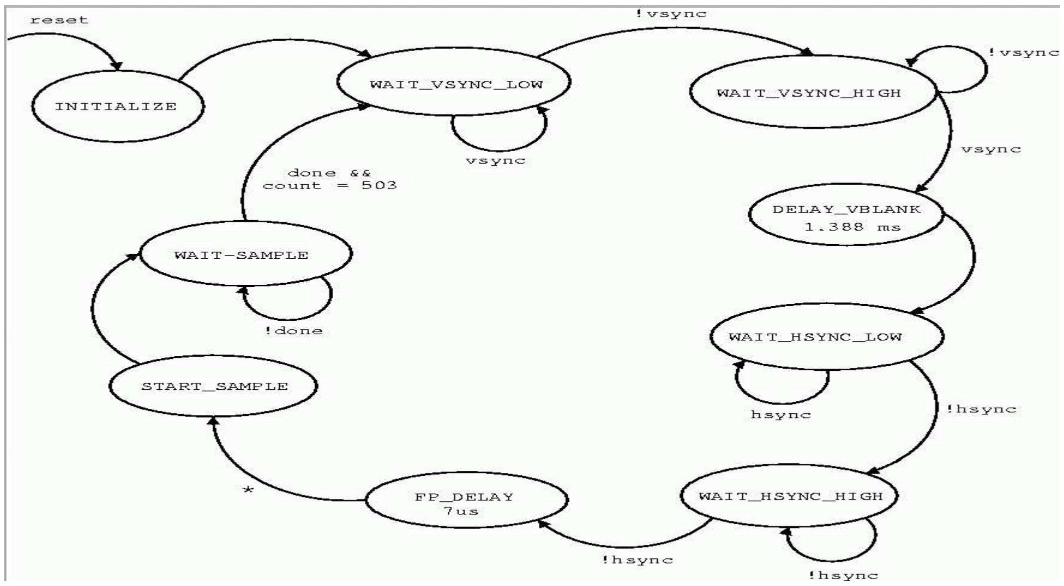


Figure 2: Controller of the Video Capture Unit.

and SAMPLE states until all sixteen bytes are filled, at which point it transitions into the WRITE1 state. Here the done signal is asserted to inform the controller that sampling is complete. The time constraints imposed by the interval between successive h-syncs is met by asserting the done signal as soon as sampling is complete. This ensures that the line count maintained by the Controller FSM is accurate. The WRITE1, WRITE2, WRITE3 and INCR_ADDR states are used to write the bytes into the Line Buffer. At the end of sixteen writes, a start_proc signal is asserted to inform the calibrator and processor to start processing stored line of video. The addr_sel signal is also switched to give the calibrator or processor access to the Line buffer.

Calibrator FSM The Calibrator FSM establishes a shoot threshold value for a player during the calibration phase of game play. After initialization in the INITIALIZE state on a reset, the Calibrator FSM waits in the IDLE state until it the start signal, provided by the digitizer FSM, is asserted. When the start signal is asserted, the Calibrator transitions into the LOAD_LINE state where it reads in sixteen bytes of data from the Line Buffer into an internal 128-bit register. It then transitions into the CALIBRATE state where it determines values for the shoot thresholds. Each of the two thresholds is initialized to 96 (corresponding to the bottom of the screen). The Calibrator FSM keeps track of the line it is currently processing and sets the threshold value to the first line number it encounters for which the 128-bit internal register is greater than 0, which indicates a blank line in the video. This process is averaged over roughly 1900 frames corresponding to 10 seconds of video to establish the final value, which is fed as input to the Processor FSM. To display the output of the calibration process, the Calibrator transitions into the WAIT_VID_REQ state where it waits for a video request signal to be asserted by the Video Display Unit. Once the request is received, the Calibrator transitions into the SETUP_DATA state where the value of the internal register is either maintained or changed to all 1s if the line number corresponds with the threshold value. The nth to (n+8)th bits of the register are setup as the data to be transmitted to the Video Display Unit, where n is the transmission count. Once the data is setup, the Calibrator asserts it's

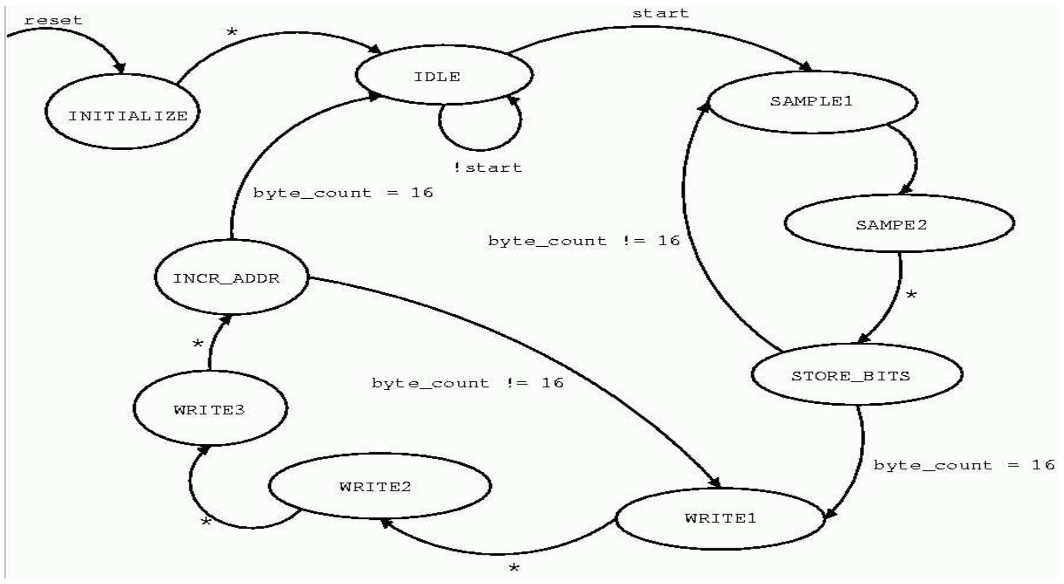


Figure 3: Digitizer FSM of the Video Capture Unit.

response/ready signal and transitions into the VERIFY_TRANSMIT state where it waits for the request signal to go low, confirming receipt of the data, at which point it sets the response output low and transitions back into the WAIT_VID_REQ state. This asynchronous transmission process is continued until all sixteen bytes are transmitted and then the Calibrator transitions back to the IDLE state to wait for the next start signal from the Digitizer. The digitizer remains in the IDLE state indefinitely once the maximum frame count is reached, indicating the end of the calibration stage. At this point memory access is handed over to the processor unit by asserting the addr_sel output. Figure 4 represents the state transition diagram of the Calibrator.

Processor FSM The processor FSM computes the position of the player in the playing area and determines if the shoot gesture has been asserted. The Processor FSM transitions into the INITIALIZE state on a reset where various counters are set to their initial values and transitions into the IDLE after one clock cycle where it waits for the start signal from the Digitizer FSM. Upon receiving the start signal in the IDLE state, the Processor FSM transitions into the INIT_HIGH_BYTE state where it reads the first byte of data corresponding to address 0x0 from the Line Buffer after which it transitions into the SETUP_LOW_BYTE state where it reads the second byte corresponding to address 0x1 from the Line Buffer. Once the two bytes have been setup, the Processor computes the left and right bounds of any object using a windowing algorithm in the COMPUTE_BOUNDS state. The algorithm relies on the assumption that the player occupied at least eight pixels and hence moves an 8-bit wide sliding window over the two bytes read from memory. If the bits in the window have the value 0xff, then the left bound is set to the value of the pointer to the beginning of the window. If not such sequence is found, the pointer is incremented until it reaches address 0xf. After computing the left bound, the bytes are reversed and the same computation is done to find the right bound of the player. Once the bounds are computed for the first two bytes, the second byte read in is shifted into the high byte position and the next byte in memory is read. This process continues until all sixteen bytes are read from

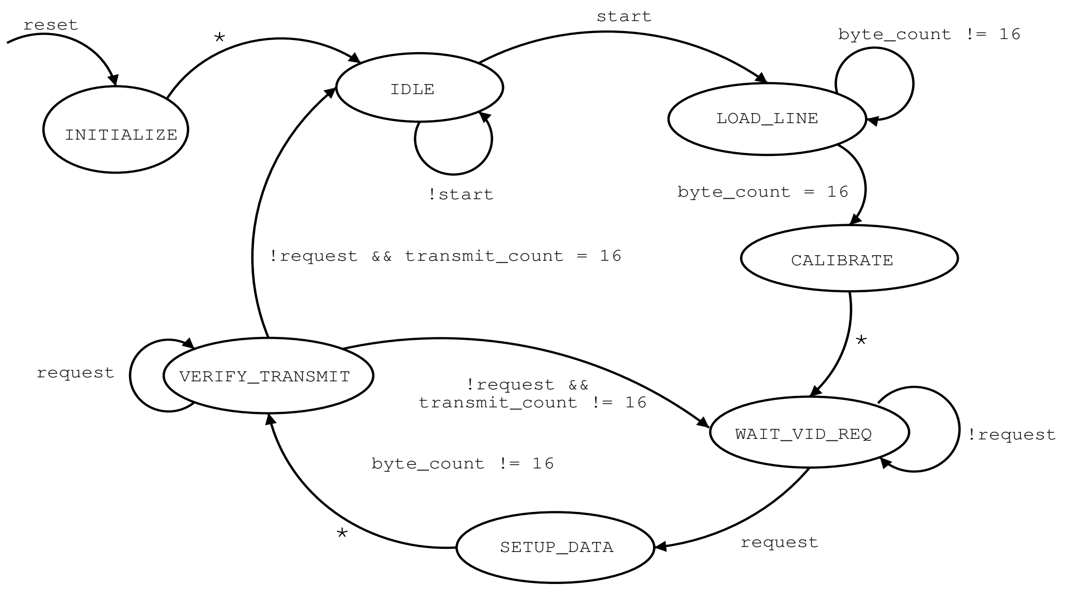


Figure 4: Calibrator FSM of the Video Capture Unit.

the Line Buffer. The shifting and loading operations are perfomed in the states SHIFT_BYTE, LOAD_ADDR_LOW_BYTE and LOAD_DATA_LOW_BYTE states.

The final value assigned to the left and right bounds are the minimum and maximum values respectively, computed over all lines in a fram. The COMPUTE_BOUNDS state also computes if a shoot signal has been asserted by comparing lines corresponding to the first black pixel it encounters with the its shoot_low and shoot_high inputs, which are provided by the Calibrator FSM. If the first black pixel is encountered in a line lower than the shoot_high input, the shoot output is set high and if it is lower than the shoot_low input, the output is set low. This in effect implements a level signal, which is converted to a pulse by the Game Controller Unit. This design choice for the shoot signal is to avoid automatic rapid fire, caused by a player keeping their arm up during game play. Once the entire frame has been read, that is 96 lines have been processed, the Processor transitions to the TRANSMIT state where it assigns it position and shoot outputs to the values in their corresponding internal values. These output values are held until the next position update at the next of the subsequent frame. After transmission, the Processor waits in the IDLE state for the next start signal from the processor. Figure 5 shows the state transition diagram for the Processor FSM.

3.4 Testing and Debugging

The entire project was simulated in the Max+Plus II development environment. Each module was individually tested by checking for correct state transitions and internal variable states on various inputs which had to be specified manually by forcing input values in the Max+Plus II simulation environment. The critical tests for the Controller FSM were with regards to certifying that delay states met their timing requirements and that other state transitions behaved according to the specifications of the h-sync and v-sync inputs. For the Digitizer FSM testing was focussed on the bit accumulation process. It was crucial that the sampled pixels were stored in the correct position

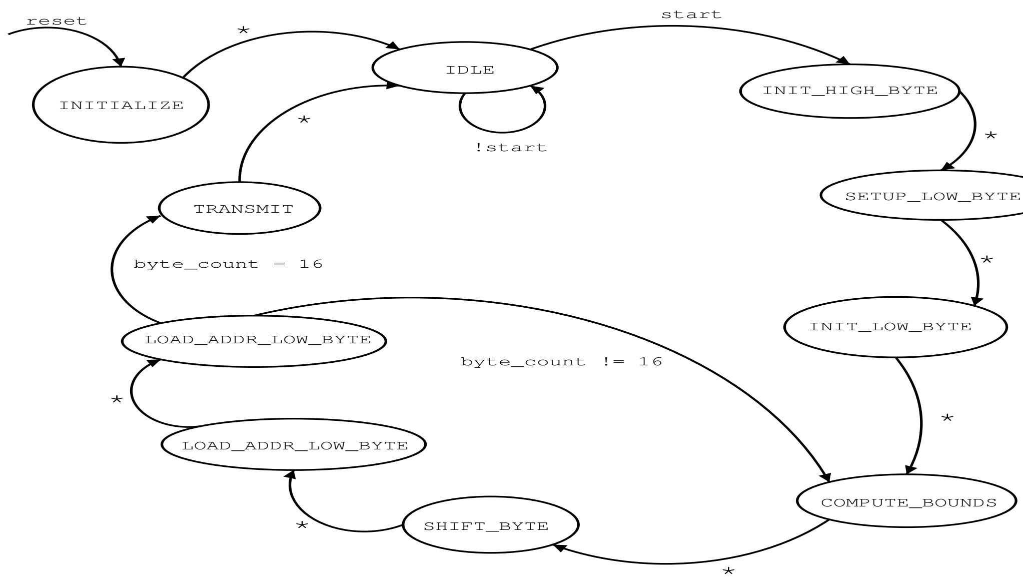


Figure 5: Processor FSM of the Video Capture Unit.

in each byte and that the bytes were written to the Line Buffer so as to represent the order in which the data was received from the AD775. Also it was crucial that the entire operation for the Digtizer finish within 63.1 us which corresponds to one horizontal line of NTSC video. Although I had initially estimated taking the average of four samples for each pixel, the simulation revealed that it number had to be revised to two samples per pixel. Even with the reduced samples per pixel rate, the done signal to the Controller FSM had to be asserted before the process of storing the pixels in the Line Buffer began. These tight timing constraints were the result of a design choice to guarantee at least 1us delay before the next h_sync signal. In retrospect, this contraint may have been overly cautious and restricting. The crucial parts of the Processor FSM was guaranteeing the level behavior of the shoot signal and the correct computation of the position as the mid-point of the leftmost and rightmost pixels in a frame. For the Calibrator FSM the tests were focussed on the asynchronous communication interface and the computation of the threshold. The latter was difficult to do in simulation, as it required manully simulating the NTSC signal over 96 lines, a rather tedious process. The alternative testing strategy was to use the Video Display Interface, but this was dependent on the hardware sampling process being functional.

The hardware testing and debugging was a frustrating experience which ultimately led to the demise of the project. After setting up the AD775 according to its specifications, I noticed that the values produced by the AD775 did not correlate with the NTSC video signal. In particular, the values seemed to have a fixed pattern even when I was certain that the NTSC signal wasn't periodic. I ensured the variability of the video signal by moving my hand in front of the camera lens, varying the aperture on the lens from fully open to closed and pointing the camera in different directions, observing the signal on an oscillioscope each time. My hypothesis for this behavior was that the reference voltages on the AD775 were not properly set. To this end, I enlisted the help of the lab assistants to confirm that I had correctly setup the AD775. Once the setup had been confirmed, I proceeded to write a test controller for a digital-to-analog converter to try to reproduce the ouput of the analog-to-digital converter. This setup revealed the same results. After several hours of trying to fine tune the reference voltages with potentiometers, it was suggest by one TA to amplify the video signal before feeding it to the AD775, as the signal range may be too small. Once I contructed the amplifying circuit and tested it with the video to make sure I had the correct gain and the video signal was indeed amplified, I proceeded to test the output through the Video Display Unit interface. The observed output did not resemble the camera view and showed little variability with moving objects in the scene. At this point, I had had run out of ideas and given up out of frustration. In retrospect I realize my folly in not attempting to substituting the AD775 with another one, but rather assuming the error was with the thresholds.

3.5 Conclusions and Reflections

The design presented in this paper is in theory a fundamentally sound one, which is well partitioned into functionally distinct units that reflect the operation of the Video Control Unit. The one crucial limiting factor in this design, which was overlooked in my implementation, however is that the success of the entire unit depends on successfully implementing the analog-to-digital conversion circuitry. Having recently completed Lab exercise 3 which dealt with similar issues, I was confindent that I would be able to implement this phase of the project when needed, and thus focussed my efforts on design memory efficient data structures and relatively fast algorithms for processing the data, assuming it was available. The algorithm chosen in retrospect seems very susceptible to noise, but this limitation may be mitigated by adjusting the 0-1 pixel threshold by trial-and-error and ensuring a certain level of lighting and contrast in the playing area. More precise algorithms may be difficult to implement, as they will involve some form of pre-processing of the digitized video, say median filtering, to reduce the noise. Alternatively, the video may be sampled at at finer level than is needed , but this will require substantially more storage and perhaps a faster system clock rate.

In conclusion, to successfully implement such an ambitious project, it is essential for any digital designer to be as familiar with analog circuitry and interfaces as he is with digital. Such knowledge will minimize the amount of time spent on debugging analog circuits and increase the time available for testing the entire system design outside of simulation, and fine tuning as necessary.

4 Game Controller

4.1 Introduction

The game controller waits for the video input to calibrate, specifies a difficulty level, and then starts the game. It takes the output of the video camera controller as the input to the game, and outputs the positions of the sprites to the video output controller. It also uses a random number generated with a random number generator to ensure that the game is different each time it is played.

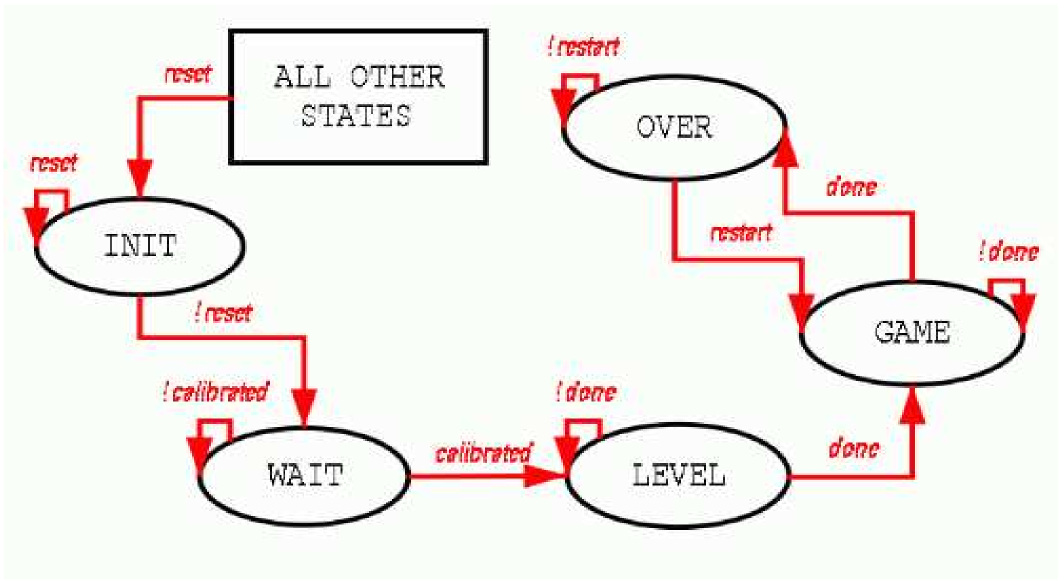


Figure 6: Major FSM of the game controller.

The game FSM is the FSM that actually controls the game play. It specifies the general sequence of operation for the game, and it uses several minor FSMs to perform the computation.

The game FSM waits for the frame request from the output, then it outputs the current state of the game in the form of the locations of the sprites. Then the game updates the state of the game by creating, moving, and destroying the sprites. Then it checks for the collisions, like bullets hitting the paratroopers, or the bombs hitting the ground, and determines which objects have to be destroyed. The checking also calculates the damage in health points the player sustains during the frame, and if the player's health reaches zero, the game ends.

4.2 Random Number Generator

To make sure that the game is different each time it is played, a random number generator is used to give randomness to the game. The best way to do it would be to use the video input, but since the video input was taking too long to finish, an alternative one was created.

The random number generator is run on a ten-megahertz clock, and it takes in a signal from a 1.84 megahertz clock as its input. It contains an eight-bit register, which is updated every clock cycle. Every clock cycle of the ten-megahertz clock, the input is sampled, and one of the bits of the eight-bit register is replaced by the exclusive-or of the input with the previous value.

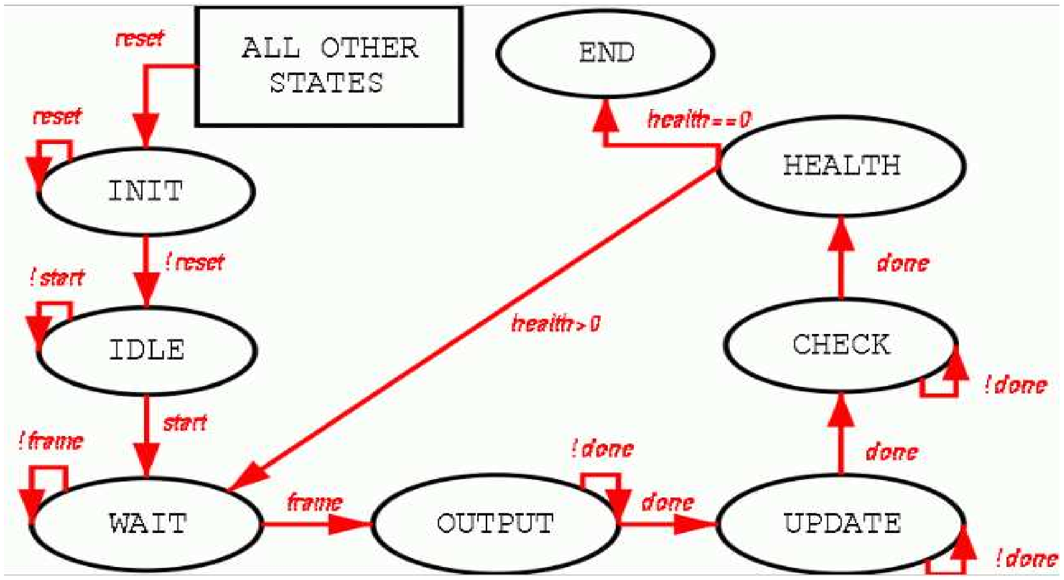


Figure 7: The game FSM of the game controller.

The random number generator cycles through the eight bits, and it is not reset with the rest of the system. Over time, the value contained in the register becomes fairly random, and the probability of the possible values is roughly uniformly distributed.

4.3 Shoot Register

Each time the shoot signal from the input is raised, the shoot register is set. The shoot register also takes a reset signal, which is passed in from the FSMs when the request is processed.

4.4 RAM

The game controller uses two registers, numheli and numobject, to specify the number of each type of objects currently on the screen. The locations of each sprite is stored in a RAM.

The RAM is stored and used by the game FSM, and it is split into two sections. The first part of it stores the positions of the helicopters, and the second part stores the positions of the bombs and the paratroopers. For each part, the objects are continuously sorted, so the valid objects are stored in the lower addresses.

The gun and the bullets are not stored in the RAM because the gun is always on the screen with its location being specified by the video input, and the bullets are stored in registers to facilitate faster checking.

4.5 Output FSM

The output FSM interfaces with the video output, and gives it the sprites and their locations. The interface consists of a request signal, a ready signal, and an eight bit output bus.

When the video output requests a new frame, it sets the request signal high. The output FSM then puts an output in the bus, and sets the ready signal high. The video output lowers the request signal when it takes the data. The output FSM then lowers the ready signal, after which the video output raises the request signal again. When all the data of the current frame is transmitted, an end frame codeword is sent through the bus, and the video output waits until the next frame to request again.

The output FSM follows that interface, and for each of the sprites, it sends a codeword to denote the type of the sprite, the horizontal position of the sprite, and the vertical position of the sprite. Therefore, information about each sprite takes three signals to send. The end frame codeword is designed to be different from any other possible message, so the interface takes at most a frame to synchronize the states of the game controller and the video output.

4.6 Collission detection

The check FSM checks for the collisions between the sprites and the bullets, and also checks whether the bombs and paratroopers hit the ground. It modifies the RAM accordingly, and calculates the damage the player receives.

When the check FSM finds that a bullet overlaps with a bomb, a paratrooper, or a helicopter, it marks both of them for removal. If the check FSM finds that a bomb or paratrooper hits the ground, it also makes it for removal, and also increments the damage calculation for the frame. Each time a paratrooper hits the ground, the player receives one point of damage, and each time a bomb hits the ground, the player is dealt two points of damage.

4.7 Updating game state

The update FSM creates, moves, and destroys the sprites, and it is the most complicated of the FSMs in the game controller. The bullets are created when a shoot signal is received before the frame, and the update FSM resets the shoot register.

For each frame, the update FSM determines whether to create a new helicopter by checking how far the previous helicopter going in that direction has moved. If the previous helicopter is sufficiently far away, a new one is created. Whether the previous helicopter is far enough away is dependent on a parameter calculated using the random number available at the time.

When a helicopter reaches a location specified in its slot in the RAM, it drops a paratrooper or a bomb. Then the next location it will drop an object is generated using the random number input. Whether the dropped object is a bomb or a paratrooper is also randomly determined, with the probability of a bomb being three-eighths and the probability of a paratrooper being five-eighths.

The update FSM moves the existing sprites by modifying the RAM. How fast the sprites move is determined by the difficulty level. The bullets, however, move at a pixel per frame regardless of the level.

The bullets marked for removal are removed immediately by the update FSM. The other sprites, however, are displayed as explosions for fifteen frames before they are removed. During the fifteen frames the sprites are displayed as explosions, they do not move or affect the game-play in any way.

5 Video Output subsystem

The Video Output subsystem acts as the interface from other two subsytems to the video monitor. This subsystem consists of the Output Generator which continuously updates an external RAM with the image to be displayed on the monitor, and the Video Display Interface which continuously reads the image from the RAM and generates the HSync, VSync, and R,G,B signals taken in by the video monitor.

During game play, the Output Generator receives asynchronous signals from the Game Controller, translates these signals into lists of objects to be displayed on the screen, and writes, onto the RAM, the image to be displayed on the monitor. The sprites for the various types of objects, such as helicopters and paratroopers, are read from an internal ROM. During calibration, the Output Generator receives asynchronous signals from the Camera Input subsystem, translates these signals into a stream of pixels corresponding to the image captured by the camera, and writes the image onto the RAM.

The Video Display Interface consists of the MC6847 video controller chip, along with some some analog circuitry, and generates the HSync, VSync, and the R,G,B signals corresponding to the image stored on the RAM. The NMS input of the MC6747 chip is asserted low by the Output Generator when the image in the RAM is being updated, so that the MC6847 will not attempt to update the display while the RAM is being written. When the Output Generator finishes updating the RAM, the NMS signal is raised, so that the MC6847 will update the display with the current image. Once the display has been updated, the MC6847 lowers the NFS signal, and at this time the Output Generator starts updating the RAM with the next image, and the cycle repeats.

Figure 8 illustrates the interconnections among the various components of the Video Output subsystem. Sections 5.1 and 5.2 describe the operation of the Output Generator and the Video Display Interface respectively.

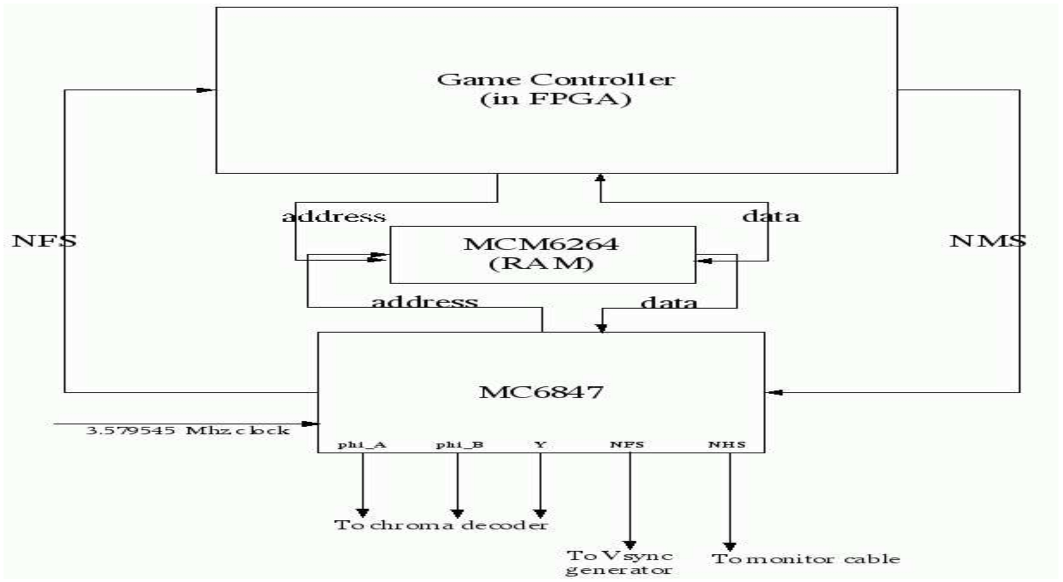
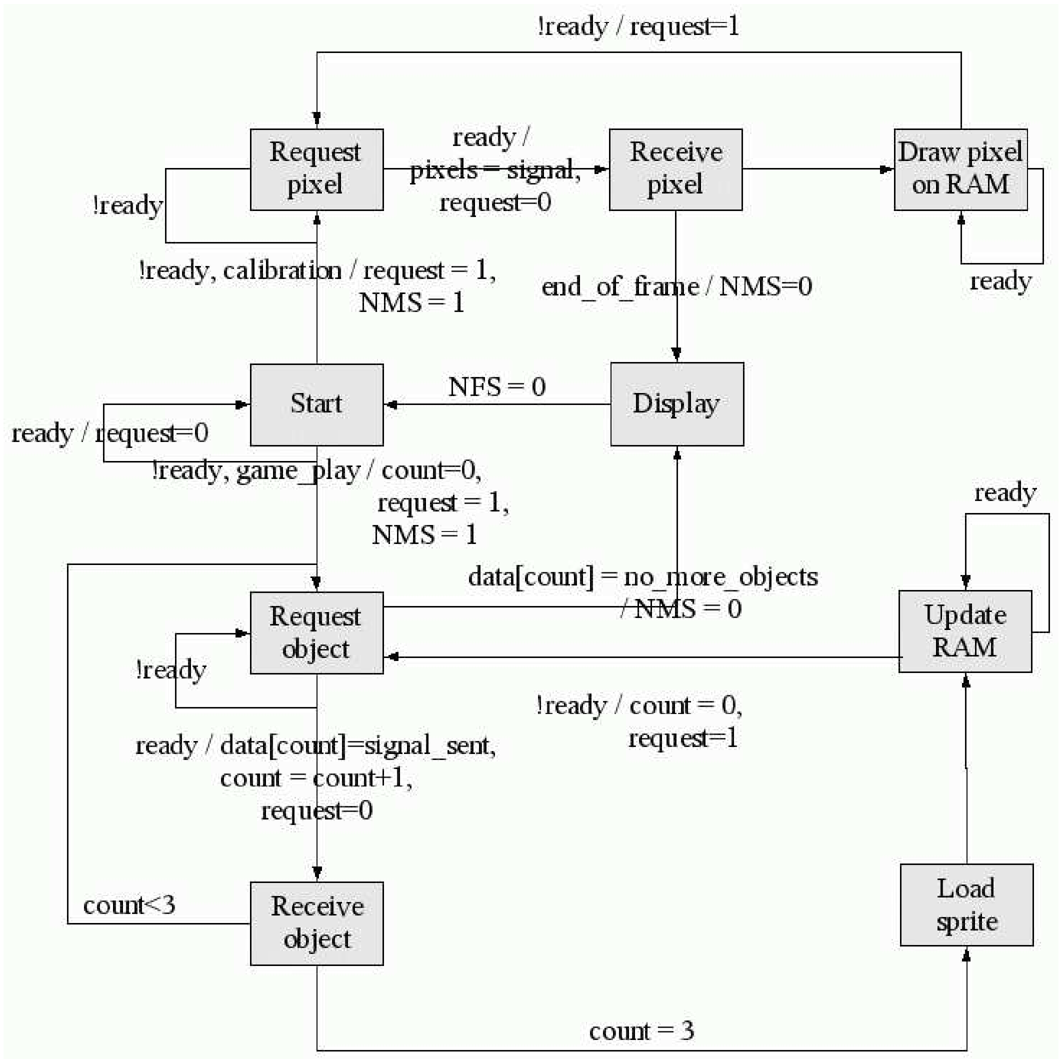


Figure 8: The Video Output subsystem.

5.1 Output Generator FSM

Figure 9 illustrates the FSM of the Output Generator. The `game_play` and `calibrate` signals are inputs to the FSM through switches, and instruct the FSM to move into the game display mode or the calibration display mode respectively.



!ready / request=1
Request pixel
ready / pixels = signal, request=0
Receive pixel
Draw pixel on RAM
!ready
ready
!ready, calibration / request = 1, NMS = 1
end_of_frame / NMS=0
Start
NFS = 0
Display
ready / request=0
!ready, game_play / count=0, request = 1, NMS = 1
ready
data[count] = no_more_objects / NMS = 0
Request object
Update RAM
!ready
!ready / count = 0, request=1
ready / data[count]=signal_sent, count = count+1, request=0
count<3
Receive object
Load sprite
count = 3

Figure 9: State transition diagram for the Output Generator FSM.

When in the game display mode, the FSM repeatedly requests three values from the Game Controller: the type of object, the x-coordinate, and the y-coordinate. As each triple is received, the sprite of the specified type is loaded, and is written at the specified location of the RAM. When the received signal has the value *no_more_objects*, the FSM raises the NMS signal and moves into the Display state. The FSM moves back to the Start state when the NFS signal is lowered by the MC6847.

When in the calibration mode, the FSM repeatedly requests, and writes to the RAM, a pixel from the Camera Input subsystem. The end of the frame is reached when the *end_of_frame* signal is received, at which point the MC6847 chip is instructed to display the image stored in the memory.

The Output Generator communicates with the Game Controller and the Camera Input Subsystem using the request/ready interface which is described in previous sections (and also illustrated in Figure 9).

5.2 Video Display Interface

The MC6847 is used in the Color Graphics Three mode, which provides a resolution of 128x96 and 4 colors. The `phi_A`, `phi_B`, and `Y` outputs are converted to the R, G, B signals using the chroma decoder shown in Figure 10.

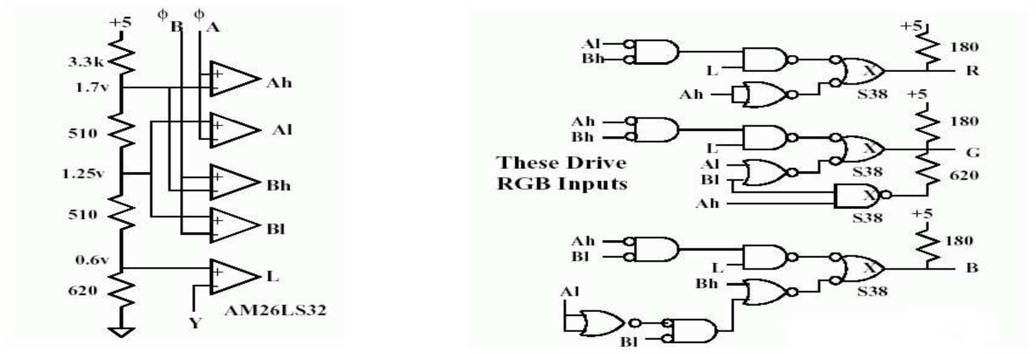


Figure 10: Chroma Decoder.
Taken from old 6.111 course notes.

The NFS (field sync) signal is converted to the VSync (vertical sync) signal using the circuitry shown in Figure 11. The two potentiometers can be adjusted to fix flickering or moving up and down of the image on the monitor.

The signals generated by the Video Display Interface are provided to the monitor cable which is illustrated in Figure 12.

5.3 Testing

The Video Output Subsystem was tested independently of the other two subsystems. To test correct operation during calibration display mode, a PAL was programmed to implement the request/ready interface expected from the Camera Input subsystem, and to deliver a stream of pixels that had a particular pattern. A clock different from the one given to the FPGA was used, to ensure that conditions are equivalent to a situation of inter-kit communcation. The correct pattern was observed on the display.

To test correct operation during game play mode, a game simulator module was written and synthesized onto the FPGA. This simulator implements the request/ready interface, and when requested, provides a list of helicopters, bombs, paratroopers and gun, and changes the positions of these objects in a known manner. The module was too large to fit into a PAL, but the module used a clock different from the one given to the Camera Input subsystem to ensure that conditions are equivalent to a situation of inter-kit communication. The objects were displayed and moved correctly.

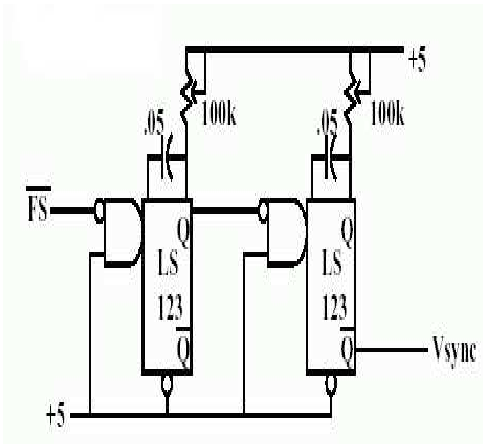
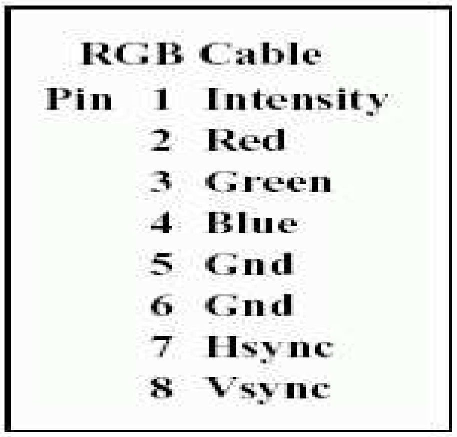


Figure 11: Generating VSync.
Taken from old 6.111 course notes.



RGB Cable		
Pin	1	Intensity
	2	Red
	3	Green
	4	Blue
	5	Gnd
	6	Gnd
	7	Hsync
	8	Vsync

Figure 12: Monitor cable pins.
Taken from old 6.111 course notes.

A Selected Verilog code for Camera Input subsystem

Listing 1: Calibrator

```
module calibrator(clk, reset, start, video_request,
                  mem_data, addr,
                  video_response,
                  video_data, end_of_frame,
                  shoot_low, shoot_high, addr_sel,
                  state
                  //phase
                  );

   input clk, reset, start, video_request;
   input [7:0] mem_data;
   output [3:0] addr;
   output      video_response;
   output [7:0] video_data;
   output       end_of_frame;
   output [6:0] shoot_low, shoot_high;
   output       addr_sel;

   //begin: debug
   output [3:0] state;
   //output     phase;
   /*
    output [6:0] line_number;
    */
   //end: debug

   reg [3:0]    state, next;
   reg [3:0]    addr_int;
   reg          addr_sel; //1: calibrator accesses memory 0: processor accesses memory

   //calibration phase
   reg          phase;

   //video data to Video Output Unit
   reg [127:0]  line_buffer;
   reg [7:0]    video_data_int;
   reg          video_response;
   reg          end_of_frame;

   //shoot level registers.
   reg [6:0]    shoot_high_int;   //line number between 0 - 96
   reg [6:0]    shoot_low_int; //line number between 0 - 96

   //counter variables
   reg [10:0]   frame_count, frame_count_int; //counter for frames processed in calibration mode
   reg [3:0]    byte_count, byte_count_int; // bytes read from memory
   reg [6:0]    line_number, line_number_int; //line number
   reg [3:0]    transmit_count, transmit_count_int;
```

```
//constants
parameter    MAX_ADDR = 4'hf;
parameter    LINES_PER_FRAME = 96; //96; //should be 96
parameter    MAX_FRAME_COUNT = 1500;//1500;// should be 1500 for 5secs //increase to match number of seconds during calib
parameter    HALF_MAX_FRAME_COUNT = 750; //should be 750 for 5secs of calibration


//states
parameter    INITIALIZE = 0;
parameter    IDLE = 1;
parameter    LOAD_LINE = 2;
parameter    CALIBRATE = 3;
parameter    WAIT_VID_REQ = 4;
parameter    SETUP_DATA = 5;
parameter    VERIFY_TRANSMIT = 6;


always @ (posedge clk) begin

   if(reset) next = INITIALIZE;
   else state = next;

   line_number = line_number_int;
   byte_count = byte_count_int;
   frame_count = frame_count_int;
   transmit_count = transmit_count_int;

   case (state)

     INITIALIZE: begin
        byte_count_int = 4'h0;
        frame_count = 0;
        line_number = 0;
        shoot_high_int = 96;
        shoot_low_int = 96;
        line_buffer = 128'h0;
        phase = 0;
        addr_int = 0;
        next = IDLE;
     end

     IDLE: begin
        if(frame_count == MAX_FRAME_COUNT) begin
           next = IDLE;
           addr_sel = 1;
        end
        else if(start) begin
           next = LOAD_LINE;
           addr_sel = 0;
        end
        else next = IDLE;
     end

     LOAD_LINE: begin

        if(byte_count == 4'hf) begin

           //fill last byte
           line_buffer[127:120] = mem_data;

           //reset address and byte_count
           addr_int = 0;
           byte_count_int = 0;

           //increase line count
           line_number_int = line_number + 1;

           //increment frame count
           if(line_number == LINES_PER_FRAME) begin
              line_number_int = 0; //reset line count
              frame_count_int = frame_count + 1;
              end_of_frame = 1; //assert end-of-frame signal
           end else line_number_int = line_number + 1;


           //if half MAX_FRAMES, go to phase 2
           if(frame_count == HALF_MAX_FRAME_COUNT) phase = 1;
           else phase = 0;

           //stop calibration after MAX_FRAME_COUNT is reached
           //if(frame_count == MAX_FRAME_COUNT) next = IDLE;
           //else next = CALIBRATE;

           next = CALIBRATE;

        end else begin

           //increment byte count
```

```
         byte_count_int = byte_count + 1;
         addr_int = addr + 1;

         case(byte_count)
           0: line_buffer[7:0] = mem_data;
           1: line_buffer[15:8] = mem_data;
           2: line_buffer[23:16] = mem_data;
           3: line_buffer[31:24] = mem_data;
           4: line_buffer[39:32] = mem_data;
           5: line_buffer[47:40] = mem_data;
           6: line_buffer[55:48] = mem_data;
           7: line_buffer[63:56] = mem_data;
           8: line_buffer[71:64] = mem_data;
           9: line_buffer[79:72] = mem_data;
           10: line_buffer[87:80] = mem_data;
           11: line_buffer[95:88] = mem_data;
           12: line_buffer[103:96] = mem_data;
           13: line_buffer[111:104] = mem_data;
           14: line_buffer[119:112] = mem_data;
         endcase // case(byte_count)

         next = LOAD_LINE;

      end // else: !if(byte_count == 4'hf)

   end // case: LOAD_LINE

   CALIBRATE: begin

      //set thresholds
      if(line_buffer > 0) begin //may need to change for noise tolerance
         //set high shoot threshold
         if(phase) begin
            if(shoot_high > line_number)
              //implement averaging later
              //shoot_high_level = shoot_high_level + line_number;
              shoot_high_int = line_number;
            else shoot_high_int = shoot_high;
         end else begin
            //set low shoot threshold
            if(shoot_low > line_number)
              //implement averaging later
              //shoot_low_level = shoot_low_level + line_number;
              shoot_low_int = line_number;
            else shoot_low_int = shoot_low;
         end
      end // if (line_buffer > 0)

      //transmit bytes
      next = WAIT_VID_REQ;

      //reset transmission count;
      transmit_count = 0;

   end // case: CALIBRATE


   WAIT_VID_REQ: begin
     if(video_request) begin
        next = SETUP_DATA;
        /*
        if(transmit_count == 15) begin
           transmit_count_int = 0;
           next = LOAD_LINE;
        end else begin
           transmit_count_int = transmit_count + 1;
           next = SETUP_DATA;
        end
         */
     end
     else next = WAIT_VID_REQ;
   end // case: WAIT_VID_REQ


   SETUP_DATA: begin
      case (transmit_count)
        0: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[7:0];
        1: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[15:8];
        2: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[23:16];
        3: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[31:24];
        4: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[39:32];
        5: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[47:40];
        6: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[55:48];
        7: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[63:56];
        8: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[71:64];
        9: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[79:72];
        10: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[87:80];
        11: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[95:88];
```

```
             12: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[103:96]
             13: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[111:104
             14: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[119:112
             15: video_data_int = ((line_number == shoot_low) || (line_number == shoot_high)) ? 8'hff : line_buffer[127:120
           endcase // case(trasmit_count)

           next = VERIFY_TRANSMIT;
        end // case: SETUP_DATA


        VERIFY_TRANSMIT: begin
           if(video_request) begin
              //acknowledge request
              video_response = 1;
              next = VERIFY_TRANSMIT;
           end else begin
              video_response = 0;
              end_of_frame = 0;
              if(transmit_count == 15) begin
                 /*
                 if(line_number == LINES_PER_FRAME) next = IDLE;
                 else begin
                    transmit_count_int = 0;
                    next = LOAD_LINE;
                 end
                  */
                 transmit_count_int = 0;
                 next = IDLE;
              end
              else begin
                 transmit_count_int = transmit_count + 1;
                 next = WAIT_VID_REQ;
              end
           end
        end
      endcase // case(state)
   end // always @ (posedge clk)

   assign addr = addr_int;
   assign shoot_low = shoot_low_int;
   assign shoot_high = shoot_high_int;
   assign video_data = video_data_int;

endmodule // calibrator
```

Listing 2: Processor

```
module processor(clk, reset, start, in_data, shoot_low, shoot_high,
                 addr, position, shoot,
                 //begin:debug
                 state
                 /*
                  line_number, low_addr, high_addr
                  */
                 //end:debug
                 );

   input clk, reset, start;
   input [7:0] in_data;
   input [6:0] shoot_low;
   input [6:0] shoot_high;

   output [3:0] addr;
   output [7:0] position;
   output       shoot;

   //begin: debug
   output [3:0] state;
   /*
    output [6:0] line_number;
    output [7:0] low_addr, high_addr;
    */
   //end: debug

   reg [3:0]    state, next;
   reg [3:0]    addr_int;

   //output registers
   reg          shoot, shoot_int;
   reg [7:0]    position;


   //left and right coordinates
   reg [7:0]    left, right;
   reg [7:0]    sum;


   //16-pixel buffer with corresponding addresses
   reg [7:0]    low_byte, high_byte;
   reg [7:0]    low_addr, high_addr;

   //line number
   reg [6:0]    line_number, line_number_int;

   //Implement if time permits:: shifter buff needed to avoid using Verilog * sign. Saves some logic


   //constants
   parameter    MAX_ADDR = 4'hf;
   parameter    LINES_PER_FRAME = 96; //actual value should be 96
   parameter    WINDOW = 8'hff;
   parameter    MAX_FRAME_COUNT = 480;
   parameter    HALF_MAX_FRAME_COUNT = 240;


   //states
   parameter    INITIALIZE = 0;
   parameter    IDLE = 1;
   parameter    INIT_HIGH_BYTE = 2;
   parameter    SETUP_LOW_BYTE = 3;
   parameter    INIT_LOW_BYTE = 4;
   parameter    COMPUTE_BOUNDS = 5;
   parameter    SHIFT_BYTE = 6;
   parameter    LOAD_ADDR_LOW_BYTE = 7;
   parameter    LOAD_DATA_LOW_BYTE = 8;
   parameter    TRANSMIT = 9;


   always @ (posedge clk) begin

      if(reset) state = INITIALIZE;
      else state = next;

      line_number = line_number_int;

      case (state)

        INITIALIZE: begin
           addr_int = 4'hf;
           low_addr = 4'h1;
           high_addr = 4'h0;
           line_number = 0;
           left = 128;
           next = IDLE; //change when you implement calibration
        end

        IDLE: begin
           addr_int = 4'h0;
           low_addr = 4'h1;
           high_addr = 4'h0;
           if(start) begin
              line_number_int = line_number + 1;
              next = INIT_HIGH_BYTE;
           end else next = IDLE;
        end

        INIT_HIGH_BYTE: begin
           high_byte = in_data;
           next = SETUP_LOW_BYTE;
        end

        SETUP_LOW_BYTE: begin
           addr_int = 4'h1;
           next = INIT_LOW_BYTE;
        end

        INIT_LOW_BYTE: begin
           low_byte = in_data;
           next = COMPUTE_BOUNDS;
        end

        COMPUTE_BOUNDS: begin

           //detect shoot gesture
           //check if hand is above upper threshold
           if({low_byte[7:0], high_byte[7:0]} > 0) begin
              //if last frame was a shoot, don't assert shoot
              if(line_number <= shoot_high) shoot_int = 1'b1;
              else if(line_number >= shoot_low) shoot_int = 1'b0;
              else shoot_int = shoot;
           end else shoot_int = shoot; //may cause problems later??

           if(left == 128) begin
              //find left boundary of player if it hasn't been found yet
              if(high_byte == WINDOW) left = (high_addr * 8);
              else if({high_byte[6:0], low_byte[7]} == WINDOW) left = (high_addr * 8) + 1;
              else if({high_byte[5:0], low_byte[7:6]} == WINDOW) left = (high_addr * 8) + 2;
              else if({high_byte[4:0], low_byte[7:5]} == WINDOW) left = (high_addr * 8) + 3;
              else if({high_byte[3:0], low_byte[7:4]} == WINDOW) left = (high_addr * 8) + 4;
              else if({high_byte[2:0], low_byte[7:3]} == WINDOW) left = (high_addr * 8) + 5;
              else if({high_byte[1:0], low_byte[7:2]} == WINDOW) left = (high_addr * 8) + 6;
              else if({high_byte[0], low_byte[7:1]} == WINDOW) left = (high_addr * 8) + 7;
              else if(low_byte == WINDOW) left = (low_addr * 8); //need to resolve address mapping to position
              else left = 128;
           end else left = left; // if (left == 8'hff)

           //find right boundary
           if(low_byte == WINDOW) right = (low_addr * 8) + 7;
           else if({high_byte[0], low_byte[7:1]} == WINDOW) right = (low_addr * 8) + 6;
           else if({high_byte[1:0], low_byte[7:2]} == WINDOW) right = (low_addr * 8) + 5;
           else if({high_byte[2:0], low_byte[7:3]} == WINDOW) right = (low_addr * 8) + 4;
           else if({high_byte[3:0], low_byte[7:4]} == WINDOW) right = (low_addr * 8) + 3;
           else if({high_byte[4:0], low_byte[7:5]} == WINDOW) right = (low_addr * 8) + 2;
           else if({high_byte[5:0], low_byte[7:6]} == WINDOW) right = (low_addr * 8) + 1;
           else if({high_byte[6:0], low_byte[7]} == WINDOW) right = (low_addr * 8);
           else if(high_byte == WINDOW) right = (high_addr * 8) + 7; //need to resolve address mapping to position
           //else right = 0;

           next = SHIFT_BYTE;

        end

        SHIFT_BYTE: begin

           high_byte = low_byte;
           high_addr = low_addr;

           if(addr == MAX_ADDR && line_number == LINES_PER_FRAME) next = TRANSMIT; //lines per frame condition not being me
           else if(addr == MAX_ADDR) next = IDLE;
           else next = LOAD_ADDR_LOW_BYTE;
        end

        LOAD_ADDR_LOW_BYTE: begin
           addr_int = addr + 1;
           next = LOAD_DATA_LOW_BYTE;
        end

        LOAD_DATA_LOW_BYTE: begin
           low_addr = addr;
           low_byte = in_data;
           next = COMPUTE_BOUNDS;
```

```
        end

        TRANSMIT: begin

           //assign shoot signal
           shoot = shoot_int;

           //assert shoot and position outputs
           if(left != 8'hff) begin
              sum = left + right;
              position = {1'b0,sum[7:1]};
           end //else send default value indicating no position data available

           //reset line count
           line_number_int = 0;
           next = IDLE;
        end

      endcase // case(state)
   end // always @ (posedge clk)

   assign addr = addr_int;

endmodule // processor
```

XX

B Selected Verilog code for Game Controller

Listing 3: Updating objects

```
module minoroutput (clk, reset, startoutput, position, outputdone
, out_request, out_ready, out_bus
, out_addr, q
, numheli, numobject, numbullet, bullet1, bullet2, bullet3, bullet4, tester);

input clk;
input reset;
input startoutput;
input [6:0] position;
output outputdone;
input out_request;
output out_ready;
output [7:0] out_bus;
output [4:0] out_addr;
input [20:0] q;
input [4:0] numheli;
input [4:0] numobject;
input [2:0] numbullet;
input [13:0] bullet1;
input [13:0] bullet2;
input [13:0] bullet3;
input [13:0] bullet4;

output [4:0] tester;//TESTER

reg [4:0] state;
reg [4:0] next;
reg [4:0] return_state;
reg [4:0] next_return_state;
reg [4:0] count;
reg [4:0] nextcount;
reg [4:0] addr;
reg [4:0] nextaddr;
reg out_ready_reg;
reg [7:0] out_bus_reg;
reg next_out_ready;
reg [7:0] next_out_bus;

parameter TROOPER_CODE=8'b00000000;
parameter BOMB_CODE=8'b00000001;
parameter HELI1_CODE=8'b00000010;
parameter HELI2_CODE=8'b00000011;
parameter EXPLODE1_CODE=8'b00000100;
parameter EXPLODE2_CODE=8'b00000101;
parameter GUN1_CODE=8'b00000110;
parameter GUN2_CODE=8'b00000111;
parameter HELI1REV_CODE=8'b00001000;
parameter HELI2REV_CODE=8'b00001001;
parameter BULLET_CODE=8'b00001010;
parameter END_CODE=8'b11000000;

parameter HEIGHT_HELI_LEFT=7'b0000000;
parameter HEIGHT_HELI_RIGHT=7'b0001000;
parameter DROP_OFFSET=7'b0000110;
parameter ORIGIN_HELI_LEFT=8'b01111111;
parameter ORIGIN_HELI_RIGHT=8'b11110001;
parameter FIRSTHELI=5'b00000;
parameter FIRSTOBJECT=5'b01000;

parameter INIT=0;
parameter IDLE=1;
parameter WAIT=2;
parameter GUN_ONE1=3;
parameter GUN_ONE2=4;
parameter GUN_ONE3=5;
parameter GUN_TWO1=6;
parameter GUN_TWO2=7;
parameter GUN_TWO3=8;
parameter OBJECT=9;
parameter OBJECT_WAIT=10;
parameter OBJECT1=11;
parameter OBJECT2=12;
parameter OBJECT3=13;
parameter OBJECT_EXPLODED1=14;
parameter OBJECT_EXPLODED2=15;
parameter OBJECT_EXPLODED3=16;
parameter HELI=17;
parameter HELI_WAIT=18;
parameter HELI_ONE1=19;
parameter HELI_ONE2=20;
parameter HELI_ONE3=21;
parameter HELI_TWO1=22;
parameter HELI_TWO2=23;
parameter HELI_TWO3=24;
parameter BULLET=25;
parameter BULLET1=26;
parameter BULLET2=27;
parameter BULLET3=28;
parameter END=29;
parameter END2=30;

always @ (posedge clk)
begin
if(reset)
          begin
          state<=INIT;
          count<=0;
          out_ready_reg<=0;
          out_bus_reg<=0;
          end
else
          begin
          state<=next;
          count<=nextcount;
          return_state<=next_return_state;
          out_ready_reg<=next_out_ready;
          out_bus_reg<=next_out_bus;
          addr<=nextaddr;
          end
end

always @ (state)
begin
nextaddr=addr;
next=state;
nextcount=count;
next_out_ready=out_ready_reg;
next_out_bus=out_bus_reg;
next_return_state=return_state;
case(state)
INIT:
          next=IDLE;
IDLE:
          if(startoutput) next=GUN_ONE1;
WAIT:
          begin
          if(out_request==1) next=return_state;
          next_out_ready=0;
          end
GUN_ONE1:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=GUN_ONE2;
                    end
          next_out_ready=1;
          next_out_bus=GUN1_CODE;
          end
GUN_ONE2:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=GUN_ONE3;
                    end
          next_out_ready=1;
          next_out_bus={1'b0,position}-8'b00000111;
          end
GUN_ONE3:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=GUN_TWO1;
                    end
          next_out_ready=1;
          next_out_bus=8'b01010011;
          end
GUN_TWO1:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=GUN_TWO2;
                    end
          next_out_ready=1;
          next_out_bus=GUN2_CODE;
          end
GUN_TWO2:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                next_return_state=GUN_TWO3;
                end
        next_out_ready=1;
        next_out_bus={1'b0,position}+8'b00000001;
        end
GUN_TWO3:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                next_return_state=OBJECT;
                nextcount=0;
                end
        next_out_ready=1;
        next_out_bus=8'b01010011;
        end
OBJECT:
        begin
        if(count<numobject)
                begin
                next=OBJECT_WAIT;
                nextaddr=FIRSTOBJECT+count;
                end
        else
                begin
                next=HELI;
                nextcount=0;
                end
        end
OBJECT_WAIT:
        next=OBJECT1;
OBJECT1:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                next_return_state=OBJECT2;
                end
        next_out_ready=1;
        if(q[20]==0 && q[0]==0) next_out_bus=TROOPER_CODE;
        else if(q[20]==0 && q[0]==1) next_out_bus=BOMB_CODE;
        else next_out_bus=EXPLODE1_CODE;
        end
OBJECT2:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                next_return_state=OBJECT3;
                end
        next_out_ready=1;
        if(q[20]==0) next_out_bus={1'b0,q[19:13]};
        else next_out_bus={1'b0,q[19:13]}-8'b00000100;
        end
OBJECT3:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                nextcount=count+1;
                if(q[20]==0) next_return_state=OBJECT;
                else next_return_state=OBJECT_EXPLODED1;
                end
        next_out_ready=1;
        next_out_bus={1'b0,q[12:6]};
        end
OBJECT_EXPLODED1:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                next_return_state=OBJECT_EXPLODED2;
                end
        next_out_ready=1;
        next_out_bus=EXPLODE2_CODE;
        end
OBJECT_EXPLODED2:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                next_return_state=OBJECT_EXPLODED3;
                end
        next_out_ready=1;
```

```
          next_out_bus={1'b0,q[19:13]}+8'b00000100;
          end
OBJECT_EXPLODED3:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=OBJECT;
                    end
          next_out_ready=1;
          next_out_bus={1'b0,q[12:6]};
          end
HELI:
          begin
          if(count<numheli)
                    begin
                    next=HELI_WAIT;
                    nextaddr=FIRSTHELI+count;
                    end
          else
                    begin
                    next=BULLET;
                    nextcount=0;
                    end
          end
HELI_WAIT:
          next=HELI_ONE1;
HELI_ONE1:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=HELI_ONE2;
                    end
          next_out_ready=1;
          if(q[20]==0 && q[19]==0) next_out_bus=HELI1_CODE;
          else if(q[20]==0 && q[19]==1) next_out_bus=HELI2REV_CODE;
          else next_out_bus=EXPLODE1_CODE;
          end
HELI_ONE2:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=HELI_ONE3;
                    end
          next_out_ready=1;
          next_out_bus=q[18:11];
          end
HELI_ONE3:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=HELI_TWO1;
                    end
          next_out_ready=1;
          if(q[19]==0) next_out_bus={1'b0,HEIGHT_HELI_LEFT};
          else if(q[19]==1) next_out_bus={1'b0,HEIGHT_HELI_RIGHT};
          end
HELI_TWO1:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=HELI_TWO2;
                    end
          next_out_ready=1;
          if(q[20]==0 && q[19]==0) next_out_bus=HELI2_CODE;
          else if(q[20]==0 && q[19]==1) next_out_bus=HELI1REV_CODE;
          else next_out_bus=EXPLODE2_CODE;
          end
HELI_TWO2:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=HELI_TWO3;
                    end
          next_out_ready=1;
          next_out_bus=q[18:11]+8'b00001000;
          end
HELI_TWO3:
          begin
          if(out_request==0)
                    begin
                    next=WAIT;
                    next_return_state=HELI;
```

```
                nextcount=count+1;
                end
        next_out_ready=1;
        if(q[19]==0) next_out_bus={1'b0,HEIGHT_HELI_LEFT};
        else if(q[19]==1) next_out_bus={1'b0,HEIGHT_HELI_RIGHT};
        end
BULLET:
        if(count[2:0]<numbullet && count==5'b00000 && (&bullet1[6:0])!=1) next=BULLET1;
        else if(count[2:0]<numbullet && count==5'b00001 && (&bullet2[6:0])!=1) next=BULLET1;
        else if(count[2:0]<numbullet && count==5'b00010 && (&bullet3[6:0])!=1) next=BULLET1;
        else if(count[2:0]<numbullet && count==5'b00011 && (&bullet4[6:0])!=1) next=BULLET1;
        else if(count[2:0]<numbullet)
                begin
                next=BULLET;
                nextcount=count+1;
                end
        else next=END;
BULLET1:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                next_return_state=BULLET2;
                end
        next_out_ready=1;
        next_out_bus=BULLET_CODE;
        end
BULLET2:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                next_return_state=BULLET3;
                end
        next_out_ready=1;
        if(count==5'b00000) next_out_bus={1'b0,bullet1[13:7]};
        if(count==5'b00001) next_out_bus={1'b0,bullet2[13:7]};
        if(count==5'b00010) next_out_bus={1'b0,bullet3[13:7]};
        if(count==5'b00011) next_out_bus={1'b0,bullet4[13:7]};
        end
BULLET3:
        begin
        if(out_request==0)
                begin
                next=WAIT;
                nextcount=count+1;
                next_return_state=BULLET;
                end
        next_out_ready=1;
        if(count==5'b00000) next_out_bus={1'b0,bullet1[6:0]};
        if(count==5'b00001) next_out_bus={1'b0,bullet2[6:0]};
        if(count==5'b00010) next_out_bus={1'b0,bullet3[6:0]};
        if(count==5'b00011) next_out_bus={1'b0,bullet4[6:0]};
        end
END:
        begin
        if(out_request==0) next=END2;
        next_out_ready=1;
        next_out_bus=END_CODE;
        end
END2:
        begin
        next=IDLE;
        next_out_ready=0;
        end
endcase
end

assign out_ready=out_ready_reg;
assign out_bus=out_bus_reg;
assign out_addr=addr;
assign outputdone=(state==END2);

assign tester[4:0]=numheli;//TESTER

endmodule
```

Listing 4: Random number generation

```
module random (clk, in, rand);

input clk;
input in;
output [7:0] rand;

reg [7:0] rand_reg;
reg [7:0] next_rand;
reg [2:0] state;
reg [2:0] next;

always @ (posedge clk)
begin
state<=next;
rand_reg<=next_rand;
end

parameter BIT0=0;
parameter BIT1=1;
parameter BIT2=2;
parameter BIT3=3;
parameter BIT4=4;
parameter BIT5=5;
parameter BIT6=6;
parameter BIT7=7;

always @ (state)
begin
next_rand=rand_reg;
case(state)
BIT0:
        begin
        next=BIT1;
        next_rand[0]=rand_reg[0] ^ in;
        end
BIT1:
        begin
        next=BIT2;
        next_rand[1]=rand_reg[1] ^ in;
        end
BIT2:
        begin
        next=BIT3;
        next_rand[2]=rand_reg[2] ^ in;
        end
BIT3:
        begin
        next=BIT4;
        next_rand[3]=rand_reg[3] ^ in;
        end
BIT4:
        begin
        next=BIT5;
        next_rand[4]=rand_reg[4] ^ in;
        end
BIT5:
        begin
        next=BIT6;
        next_rand[5]=rand_reg[5] ^ in;
        end
BIT6:
        begin
        next=BIT7;
        next_rand[6]=rand_reg[6] ^ in;
        end
BIT7:
        begin
        next=BIT0;
        next_rand[7]=rand_reg[7] ^ in;
        end
endcase
end

assign rand=rand_reg;

endmodule
```

C Selected Verilog code for Video Output subsystem

Listing 5: Top level module

```
module video_output(clk,
        reset,
        ext_addr, //[12:0]
        nfs,
        nms,
        ext_data, //[7:0]
        G_b, W_b,

        calib,
        request,
        ready,
        pixels, //[7:0]
        endframe,

        game,
        do_game_sim);

        input clk;
        input reset;
        inout[12:0] ext_addr;
        input nfs;
        output nms;     reg nms;
        inout[7:0] ext_data;
        output G_b;
        output W_b;

        input calib;
        output request; reg request;
        input ready; reg ready_sync;
        input[7:0] pixels; reg[7:0] pixels_sync;
        input endframe; reg endframe_sync;

        input game;
        input do_game_sim;


        reg read, write;
        wire[2:0] cntr_state;
        wire data_oen, address_load, data_sample;
        reg[12:0] ram_address;
        reg[7:0] data_write;
        reg[7:0] data_read;
        reg address_oen;
        controller mycontroller(clk, 1, G_b, W_b, ram_address,
                ext_addr, data_write, data_read, ext_data, read,
                write, cntr_state, data_oen, address_load, data_sample,
                address_oen);


        reg[6:0] image_address;
        wire[15:0] image_q;
        image_rom image_rom1(image_address, image_q);

        reg[6:0] gun_address;
        wire[15:0] gun_q;
        gun_rom2 gun_rom21(gun_address[4:0], gun_q);


        reg[6:0] digits_address;
        wire[7:0] digits_q;
        digits_rom digits_rom1(digits_address, digits_q);


        wire sim_ready;
        wire[7:0] sim_pixels;
        gamesim mygamesim(clk,
                        reset,
                        do_game_sim,

                        request,
                        sim_ready,
                        sim_pixels);




        reg[5:0] state;
        parameter DISPLAY = 0;
        parameter START_WRITING = 1;

        parameter WRITE_CALIB = 2;
        parameter WAIT_READY = 3;
        parameter GRAB_DATA = 4;
        parameter GRAB_DATA_OTHER = 5;
        parameter WRITE_2 = 6;
        parameter WRITE_3 = 7;
        parameter WRITE_4 = 8;


        parameter BLANK_GAME = 19;
        parameter BLANK_GAME_2 = 20;
        parameter BLANK_GAME_3 = 21;
        parameter BLANK_GAME_4 = 22;

        parameter WRITE_GAME = 9;
        parameter WAIT_GAME_READY = 10;
        parameter GRAB_GAME_DATA = 11;
        parameter WRITE_GAME_2 = 12;
        parameter WRITE_GAME_3 = 13;
        parameter WRITE_GAME_4 = 14;
        parameter WRITE_GAME_5 = 15;
        parameter WRITE_GAME_6 = 16;
        parameter WRITE_GAME_7 = 17;

        reg[7:0] halfline;
        reg[7:0] write_byte1;
        reg[7:0] write_byte2;
        reg which_byte;
        parameter REVERSE = 27;
        parameter SMOOTH_WRITE = 23;
        parameter SMOOTH_READ = 24;
        parameter SMOOTH_READ_2 = 25;
        parameter SMOOTH_WRITE_2= 26;

        reg[1:0] text_count;
        parameter WRITE_TEXT = 28;
        parameter WRITE_TEXT_2 = 29;
        parameter WRITE_TEXT_3 = 30;
        parameter WRITE_TEXT_4 = 31;
        parameter WRITE_TEXT_5 = 32;
        parameter WRITE_TEXT_6 = 33;
        parameter WRITE_TEXT_7 = 34;

        parameter DONE_WRITING = 18;



        reg[7:0] grabbed_pixels;
        reg grabbed_other;

        reg[7:0] type;
        reg[7:0] xpos;
        reg[7:0] ypos;
        reg[1:0] game_grabbed_count;
        reg[3:0] digit;

        reg[3:0] row;
        reg which_half;
        wire[5:0] thing;
        assign thing = (xpos[7:2] + which_half + 1);

        always @(posedge clk) begin
                if (do_game_sim) begin
                        ready_sync <= sim_ready;
                        pixels_sync <= sim_pixels;
                end
                else begin
                        ready_sync <= ready;
                        pixels_sync <= pixels;
                end
                endframe_sync <= endframe;

                if (reset) begin
                        nms <= 0;
                        address_oen <= 1;
                        read <= 0;
                        write <= 0;
                        ram_address <= 13'd 0;
                        request <= 0;
                        state <= START_WRITING;
                end
                else if (state != DISPLAY) begin
                        //write the display ram
                        case (state)
                                START_WRITING: begin
                                        nms <= 0;
                                        address_oen <= 1;
                                        read <= 0;
                                        write <= 0;
                        ram_address <= 13'd 0;
                        request <= 0;
                        if (calib)
                                state <= WRITE_CALIB;
                        else if (game) begin
                                game_grabbed_count <= 2'd 0;
                                state <= BLANK_GAME;
                        end
                end




                // *** calibration ***
                WRITE_CALIB: begin
                        if (!ready_sync) begin
                                request <= 1;
                                state <= WAIT_READY;
                        end
                end

                WAIT_READY: begin
                        if (ready_sync) begin
                                state <= GRAB_DATA;
                        end
                end

                GRAB_DATA: begin
                        grabbed_pixels <= pixels;
                        data_write[7] <= pixels[7];
                        data_write[6] <= pixels[7];
                        data_write[5] <= pixels[6];
                        data_write[4] <= pixels[6];
                        data_write[3] <= pixels[5];
                        data_write[2] <= pixels[5];
                        data_write[1] <= pixels[4];
                        data_write[0] <= pixels[4];
                        grabbed_other <= 0;
                        request <= 0;
                        if (endframe_sync)
                                state <= DONE_WRITING;
                        else
                                state <= WRITE_2;
                end

                GRAB_DATA_OTHER: begin
                        data_write[7] <= grabbed_pixels[3];
                        data_write[6] <= grabbed_pixels[3];
                        data_write[5] <= grabbed_pixels[2];
                        data_write[4] <= grabbed_pixels[2];
                        data_write[3] <= grabbed_pixels[1];
                        data_write[2] <= grabbed_pixels[1];
                        data_write[1] <= grabbed_pixels[0];
                        data_write[0] <= grabbed_pixels[0];
                        grabbed_other <= 1;
                        state <= WRITE_2;
                end

                WRITE_2: begin
                        write <= 1;
                        state <= WRITE_3;
                end
                WRITE_3: begin
                        write <= 0;
                        state <= WRITE_4;
                end
                WRITE_4: begin
                        if (cntr_state == 3'b 0) begin
                                if (ram_address < 13'd 3071) begin
                                        ram_address <= ram_address + 13'd 1;
                                        if (grabbed_other)
                                                state <= WRITE_CALIB;
                                        else
                                                state <= GRAB_DATA_OTHER;
                                end
                                else
                                        state <= DONE_WRITING;
                        end
                end
                // *** end of calibration ***




                // *** game play ***
                BLANK_GAME: begin
                        if ( (ram_address < 13'd 32) ||
                        (ram_address >= 13'd 3040) )
                                data_write <= 8'b 11111111;
                        else if (ram_address[4:0] == 5'b 00000)
                                data_write <= 8'b 11000000;
                        else if (ram_address[4:0] == 5'b 11111)
                                data_write <= 8'b 00000011;
                        else
                                data_write <= 8'b 00000000;
                        state <= BLANK_GAME_2;
                end
                BLANK_GAME_2: begin
                        write <= 1;
                        state <= BLANK_GAME_3;
                end
                BLANK_GAME_3: begin
                        write <= 0;
                        state <= BLANK_GAME_4;
                end
                BLANK_GAME_4: begin
                        if (cntr_state == 3'b 0) begin
                                if (ram_address < 13'd 3071) begin
                                        ram_address <= ram_address + 13'd 1;
                                        state <= BLANK_GAME;
                                end
                                else begin
                                        state <= WRITE_GAME;
                                end
                        end
                end



                WRITE_GAME: begin
                        if (!ready_sync) begin
                                request <= 1;
                                state <= WAIT_GAME_READY;
                        end
                end

                WAIT_GAME_READY: begin
                        if (ready_sync) begin
                                state <= GRAB_GAME_DATA;
                        end
                end

                GRAB_GAME_DATA: begin
                        if (game_grabbed_count == 2'd 0)
                                type <= pixels_sync;
                        else if (game_grabbed_count == 2'd 1)
                                xpos <= pixels_sync;
                        else if (game_grabbed_count == 2'd 2)
                                ypos <= pixels_sync;
                        game_grabbed_count <= game_grabbed_count + 2'd 1;
                        request <= 0;

                        if (pixels_sync == 8'b 11000000)
                                state <= DONE_WRITING;
                        else
                                if (game_grabbed_count < 2'd 2)
                                        state <= WRITE_GAME;
                                else
                                        state <= WRITE_GAME_2;
                end

                WRITE_GAME_2: begin
                        ram_address <= {ypos[7:0], 5'b 00000} + xpos[7:2];

                        //image_address <= type[6:0] * 7'd 12;
                        //gun_address <= (type[6:0]-7'd 6) * 7'd 12;
                        if (type < 8'd 6)
                                image_address <= type[6:0] * 7'd 12;
                        else if (type == 8'd 8)
                                image_address <= 7'd 24;
                        else if (type == 8'd 9)
                                image_address <= 7'd 36;
                        gun_address <= (type[6:0] - 7'd 6) * 7'd 12;

                        if ((type == 8'd 11 || type == 8'd 12)) begin
                                state <= WRITE_TEXT;
                        end
                        else begin
                                row <= 0;
                                state <= WRITE_GAME_3;
                        end
                end

                WRITE_GAME_3: begin
                        which_half <= 0;
                        state <= WRITE_GAME_4;
                end

                WRITE_GAME_4: begin
                        if (! which_half) begin
                                if (type < 8'd 6)
                                        halfline <= image_q[15:8];
                                else if (type == 8'd 6 || type == 8'd 7)
                                        halfline <= gun_q[15:8];
                                else if (type == 8'd 8 || type == 8'd 9)
                                        halfline <= image_q[7:0];
                                else if (type == 8'd 10)
                                        if (row == 4'd 0)
                                                halfline <= 8'b 11000000;
                                        else
                                                halfline <= 8'b 00000000;
                        end
                        else begin
                                if (type < 8'd 6)
                                        halfline <= image_q[7:0];
                                else if (type == 8'd 6 || type == 8'd 7)
                                        halfline <= gun_q[7:0];
                                else if (type == 8'd 8 || type == 8'd 9)
                                        halfline <= image_q[15:8];
                                else if (type == 8'd 10)
                                        halfline <= 8'b 00000000;
                        end
                        state <= REVERSE;
                end

                REVERSE: begin
                        if (type == 8'd 8 || type == 8'd 9) begin
                                halfline[0] <= halfline[7];
                                halfline[1] <= halfline[6];
                                halfline[2] <= halfline[5];
                                halfline[3] <= halfline[4];
                                halfline[4] <= halfline[3];
                                halfline[5] <= halfline[2];
                                halfline[6] <= halfline[1];
                                halfline[7] <= halfline[0];
                        end
                        state <= SMOOTH_WRITE;
                end


                SMOOTH_WRITE: begin
                        case (xpos[1:0])
                                2'd 0: begin
                                        write_byte1 <= halfline[7:0];
                                        write_byte2 <= 8'b 00000000;
                                end
                                2'd 1: begin
                                        write_byte1 <= {2'b 00, halfline[7:2]};
                                        write_byte2 <= {halfline[1:0], 6'b 000000};
                                end
                                2'd 2: begin
                                        write_byte1 <= {4'b 0000, halfline[7:4]};
                                        write_byte2 <= {halfline[3:0], 4'b 0000};
                                end
                                2'd 3: begin
                                        write_byte1 <= {6'b 000000, halfline[7:6]};
                                        write_byte2 <= {halfline[5:0], 2'b 00};
                                end
                        endcase
                        which_byte <= 0;
                        state <= SMOOTH_READ;
                end

                SMOOTH_READ: begin
                        read <= 1;
                        state <= SMOOTH_READ_2;
                end

                SMOOTH_READ_2: begin
                        read <= 0;
                        state <= SMOOTH_WRITE_2;
                end

                SMOOTH_WRITE_2: begin
                        if (cntr_state == 3'b 0) begin
                                if (! which_byte)
                                        data_write <= write_byte1 | data_read;
                                else
                                        data_write <= write_byte2 | data_read;

                                //don't display parts that are out of bounds
                                if ((ram_address[12:5] == (ypos+row)))
```

```
                                state <= WRITE_GAME_5;
                        else
                                state <= WRITE_GAME_7;

                end
        end

        WRITE_GAME_5: begin
                write <= 1;
                state <= WRITE_GAME_6;
        end

        WRITE_GAME_6: begin
                write <= 0;
                state <= WRITE_GAME_7;
        end

        WRITE_GAME_7: begin
                if (cntr_state == 3'b 0) begin
                        if (! which_byte) begin
                                //ram_address <= ram_address + 13'd 1;
                                ram_address <= {ypos[7:0], 5'b 00000} + thing + (13'd 32 * row);
                                which_byte <= 1;
                                state <= SMOOTH_READ;
                        end
                        else begin
                                if (! which_half) begin
                                        //ram_address <= ram_address + 13'd 1;
                                        which_half <= 1;
                                        state <= WRITE_GAME_4;
                                end
                                else begin
                                        if (row < 11) begin
                                                ram_address <= ram_address + 13'd 30;
                                                image_address <= image_address + 7'd 1;
                                                gun_address <= gun_address + 7'd 1;
                                                row <= row + 4'd 1;
                                                state <= WRITE_GAME_3;
                                        end
                                        else begin
                                                game_grabbed_count <= 2'd 0;
                                                state <= WRITE_GAME;
                                        end
                                end
                        end
                end
        end


        WRITE_TEXT: begin
                if (type == 8'd 11)
                        ram_address <= 13'd 2851;
                else
                        ram_address <= 13'd 2879;
                digit <= xpos - (xpos / 8'd 10) * 8'd 10;
                xpos <= xpos / 8'd 10;
                text_count <= 2'd 0;
                state <= WRITE_TEXT_2;
        end

        WRITE_TEXT_2: begin
                digits_address <= digit * 7'd 7;
                state <= WRITE_TEXT_3;
        end

        WRITE_TEXT_3: begin
                state <= WRITE_TEXT_4;
        end

        WRITE_TEXT_4: begin
                data_write <= digits_q;
                state <= WRITE_TEXT_5;
        end

        WRITE_TEXT_5: begin
                write <= 1;
                state <= WRITE_TEXT_6;
        end

        WRITE_TEXT_6: begin
                write <= 0;
                state <= WRITE_TEXT_7;
        end

        WRITE_TEXT_7: begin
                if (cntr_state == 3'b 0) begin
                        if (text_count < 2'd 2) begin
                                ram_address <= ram_address - 13'd 1;
```

```
                            digit <= xpos - (xpos / 8'd 10) * 8'd 10;
                            xpos <= xpos / 8'd 10;
                            text_count <= text_count + 2'd 1;
                            state <= WRITE_TEXT_2;
                        end
                        else begin
                            state <= WRITE_GAME;
                        end
                    end
                end
                // *** end of game play ***




                DONE_WRITING: begin
                    nms <= 1;
                    address_oen <= 0;
                    read <= 1;
                    state <= DISPLAY;
                end
                default: begin
                    state <= START_WRITING;
                end
            endcase
        end
        else begin
            //let the display chip read from the ram
            if (!nfs) begin
                state <= START_WRITING;
            end
            else begin
            end
        end
    end
endmodule
```