Interactive Video Effects and Games

Rebecca Arvanites
Cristina Domnisoru
6.111
Introductory Digital Systems Laboratory
Professors Chris Terman and Ike Chuang

December 14, 2005

Abstract

This report documents the development and implementation of live image processing to produce video effects and two interactive games. The general architecture of the project consists of two sets of modules, the video decoding and effect-adding modules, and the game modules which process game logic. A double buffer system is used to store pixel data from the camera and read the previous frame, and for the effect module to write a frame while another is being displayed. One game implemented in this project is a sliding puzzle game using blocks of the camera video as the puzzle pieces that the player rearranges. Another game overlays falling objects on the camera video and uses color detection to allow the player to interact with the game by catching the falling blobs and avoiding the falling bombs.

Table of Contents

List of Figures

Overview

Our interactive video project had two main goals when we started: to apply interesting effects to video from a camera, and to make games in which the user interacts with the applied video effects. The user of the system will see themselves on the screen, and their actions are incorporated into the game while effects are added. This involves communicating with the camera, adding effects to the video, processing game logic, and displaying the altered video.

The structure of the project is made up of two main sets of modules, the video processing and effect modules, and the game modules. The video processing and effect modules read and store data from the camera, add effects to the frame, then write and display the video frames. A double buffer system is used such that the camera writes to one frame while the video processing modules read from another, and the frames swap when the camera has finished writing; similarly the effect module writes to one frame while the display module reads from the other, until the effect module has written the whole frame and the reader and writer swap frames. The game modules communicate with the video effect modules by telling them what to display according to the game logic.

In the puzzle game, the camera video is rearranged by blocks of pixels, creating a sliding tile puzzle where the player switches adjacent squares with a blank square in attempts to put the video in normal order. Buttons are used to select the current square and swap its location with a blank square.

In the falling blob game, the camera video of the player is overalyed by falling game-generated blob objects, some of which the player wants to catch and some of which the player is trying to avoid. The player uses a red paddle card or their hand to catch the blobs on the screen, gaining points for catching square objects and losing points for coming into contact with falling bombs.

Subsystem 1 (by Cristina Domnisoru)

Overall Design

The overall design of the graphics system is as follows: NTSC data is sent from the camera through a decoder that parses the stream into coherent pixels in YCrCb format and processes the sync signals from the camera. The pixels are then sent to an RGB converter that also delays the sync signals to account for the time required to compute the conversion to RGB. This module keeps 6 bits for each of the 3 channels, each pixel becoming 18 bits long. The 18 bit pixels are sent to a memory writing module that clumps pairs of adjacent pixels together and writes 2 pixels to a ZBT 1 memory location (which is 36 bits wide). An effect module reads the pixels out of that memory, processes them and writes them to a ZBT 0 memory location. (ZBT1 and ZBT0 are the two ZBTs that the Xilinks Labkit contains). A display module reads the pixels out of ZBT 0 (these are processed pixels) and outputs them to the screen. The display module reads one memory address at a time and outputs the two pixels contained in it one after the other.

Memory Constraints

Before elaborating on the particular modules involved, we will provide an explanation of the memory requirements that constrained the entire design. Given a limited amount of memory, we had to make two basic choices: how many pixels to store per frame and how many bits to store per pixel. A 720 x 480 image has 365.600 pixels. A ZBT ram has 524.288 rows. Since the camera outputs 720 * 480 pixels per frame, we decided that there was no need to store more pixels than that per frame (interpolation could always be done at the output to obtain a larger image). At that resolution, there are enough rows in one ZBT to store one frame at one pixel per row, or two frames at 2 pixels per row. Since each row of a ZBT is 36 bits wide, we decided to store 2 pixels per row, implying 18 bits per pixel and 6 bits out of 8 per channel. These choices allowed us to store 4 frames in memory, 2 per ZBT. ZBT's are convenient for image processing because they do not have wasted cycles and can be clocked at 65 MHz , the required speed for XVGA, sufficient to accommodate our display.

Buffer System

Our design uses 4 buffers (since we can store 4 frames in memory). 2 buffers are used by the camera to store input and are read by the effect modules. The other 2 buffers are used by the effect module to store processed pixels and are read by the display module. Two modules, called camera_frame_swapper and display_frame_swapper control which frame is used by which module at a particular point in time. The two buffers used by the effect and display are indispensable because they allow the effect to move pixels from their original location. Effect modules process pixels sequentially, row by row and from left to right. Each pixel however, upon being processed, could be reassigned a position that is in a different location. Therefore, if we were to use only one buffer for display, on a write cycle an effect might write a pixel in some arbitrary location and the display module might display that pixel out of order. This would cause unwanted artifacts. In order to allow the effects to move pixels freely when they construct the following image, the effect modules have to have a work buffer that is not currently being displayed. Consequently, the display module and effect module share the display ZBT (ZBT0) but at any given point in time the effect is writing to one buffer and the display is reading from the other buffer. When the effect completes the working buffer and thus a new frame is ready, the buffer swapper swaps the buffers and now the display is reading the new buffer and the effect is starting to fill another buffer from the beginning. While this buffering is necessary, buffering the input is not.

It is possible not to use memories at all to obtain the input from the camera, since the camera outputs pixels in sequence and we can just grab them as they come. However, the camera outputs pixels at a different frequency (on a different clock) than the monitor requires pixels for display, and to output at the frequency required by the monitor it is much easier to use a frame that is already complete. It is also possible to use only one buffer at the input and simply allow the camera to output whatever pixel it wants on the write cycles and on the read cycles to read and display the pixels in order. Since no processing is done and the frame rate greatly exceeds the speed of most real-world changes, this is not a problem. In fact, this was the initial implementation of this system written by Professor Chuang as an example of ZBT and camera usage and it worked very well. We decided against these two approaches because we wanted to allow ourselves the possibility of "freezing" frames. In other words, we wanted to be able to have the camera

write to a buffer and then write only to the other buffer, leaving the first with a still image. This capability is very useful for generating interesting effects, because it allows us to differentiate between the current events in the picture and the initial events in the picture. For example, this allowed us to freeze frames and then apply effects to those objects in the frame that had changed. This allows for a primitive (but useful) way of applying effects to people (by having them walk into the sight of the camera after the frame has been frozen).

There are two main operation modes for controlling the buffers. In one mode, the buffers swap automatically after frame completion. In the other mode, the buffers can be controlled manually. The freeze frame effects all operate in the manual mode. The manual mode uses 4 switches on the FPGA (switches 7 through 4) and their value indicates the buffer being used, in order from 7 to 4, by camera to write (ZBT1), by effect to read (ZBT1), by effect to write (ZBT0), by display to read (ZBT0).

Addressing Memory

Since we have decided to store 2 pixels per row of memory and only 2 buffers, we can have a computation-free addressing system. This is very useful because it allows for more time for interesting computation. The system is organized such that each 2 adjacent pixels on one row are clumped together. When examined on the logic analyzer, the maximum row was 508 and the maximum column was 720. We only have to store 508 * 360 pairs, so 508 * 360 memory locations per frame. 508 and 360 are both less then 2 ^ 9 (512) and therefore we can just concatenate their binary representations and only take up 18 bits. The 19^{th} address is the buffer bit. The address will be specifically stored as a 19 bit number such that bit 18 is frame, 17:8 is 9 bits of hcount ignoring the least significant (because we're dividing by 2, 2 pixels per address) and vcount is bits 7:0. A separate module is provided for address computation.

Overall Timing

In order to accommodate buffer swapping and using both memories to read and write, the timing of the system is rigorous. Both ZBTs operate using the same write enable signal such that if the camera is writing to ZBT1, the effect is writing to ZBT0. If the effect is reading ZBT1, the display is reading ZBT0. This write enable signal flips every clock cycle such that the timing is read-write-read-write etc. The write enable signal is cotrolled by the parity (lowest order bit) of a counter that increments each clock cycle. In practice this is not an actual counter but the hcount (x value) signal generated by the XVGA module from Lab4. Therefore, we write to each memory every 2 clock cycles and read from each memory every 2 clock cycles. This is possible because we are storing 2 pixels per memory location, and therefore, when the effect module writes every 2 clock cycles, it writes 2 pixels at once. When the display module is reading every 2 clock cycles, it is reading 2 pixels at once. So effectively every clock cycle the display gets a new pixel (and therefore we were able to clock our circuit at 65 Mhz as required by the monitor). If the camera also output pixels at that frequency it would not be a problem for the system to accommodate it, but as it is the camera outputs slower so even though the camera has the opportunity to write 2 new pixels every other clock cycle sometimes it writes the same 2 pixels to the same memory address.

Camera-Display Timing

A particular timing issue that we had not anticipated was the fact that we couldn't clock our monitor at some multiple of the camera clock and that we had to use a synchronizer. The camera part of the processing and the display part of the processing are therefore completely dissociated.

Following is a more thorough description of the modules involved in the graphics datapath . The modules will be described in detail in the general data path direction from camera to monitor.

1. NTSC Decoder

This module (called ntsc_decode.v) was written by Javier Castro and it parses the stream output from the camera. The decoder issues an f signal representing the field bit, either even or odd, from the camera, and a v and h signal for vsync and hsync of the camera. It outputs 30 bits of ycrcb and a data valid dv signal every time a pixel has been parsed. The code is included in the appendix.

2. ZBT 6.111

This module is a ZBT interface module written by Isaac Chuang. It does not hide the 2 cycle latency in reads and writes from ZBTs and it inverts the clock and we to the memory in order to have enough data setup time before the clock edge of the memory. This module is used to interface with both ZBTs and it is very convenient and simple to use.

3. RGB Converter

This module (called fvhdelayer.v) takes the fvh and dv signals issued by the NTSC decoder and delays them by 5 cycles, and also takes the ycrcb input from the NTSC decoder and calls a Xilinx ycrcb to RGB converter. This converter requires 5 cycles in order to complete the multiplications involved in the conversion, so in order to keep the pixel and dv, fvh synchronized we have to delay the latter by the forementioned 5 clock cycles. This module is clocked at the clock of the camera because the fvh signals are changing with respect to that clock.

4. NTSC_TO_ZBT

The ntsc_to_zbt module is central to the image processing subsystem. The original was written by Isaac Chuang and we changed it to accommodate our project. This module has to interface between data from the camera and (delayed and converted to RGB) and a ZBT memory. More specifically, it takes in fvh, dv, RGB, and has to issue an address and

a data value that will afterwards be written in memory to that address. This module has to clump 2 pixels into one and write them to the correct address, by outputting the data they represent at the correct time. The difficulty involved comes from the fact that the camera operates at its own ~27 Mhz clock frequency, whereas the memory has to operate at 65 Mhz to accommodate the monitor which requires that frequency. This requires clock synchronization. Essentially, this module will receive as input f, v, h, dv and data that are all synchronized with each other. In other words, dv corresponds to valid data and v, h, f correspond to that same data stream. However, these signals are generated from the camera and so they are also synchronized to the edges of the camera clock. Since the memory is clocked at a different frequency that is not a multiple of 27, this module must generate some new signals that correspond to these but that are synchronized to the output clock.

The module computes a row and column based on fvh (by counting) . The camera outputs coherent data only when f = 0 and in that case it outputs all the even rows then all the odd rows etc. By detecting the edge of the frame, we compute an even/odd bit that flips if the frame flips. Since NTSC sends all of the odd rows or even rows before sending a vsync, a counter that increments to vsync would only have as a maximum value half of the height of the video. Therefore, when we compute the address to store a pixel in, we must add a bit that specifies whether this pixel is in an odd row or an even row. For example, a pixel in the even field with row = 100 in reality should be displayed in the row 200, because it is currently in the 100^{th} even row. We can compute a signal that tells us whether data is ready to be written by checking that data is valid, that data just became valid (we want the write enable signal to be a pulse) and that the f = 0. We call the even/odd bit even_odd.

Given that we have the row, column, write enable, data and even_odd synchronized to the camera frequency, we must synchronize them to the ZBT frequency. We therefore latch each of those signals twice on the 65 Mhz frequency. Latching only once would be risky because we might be looking too close to a clock edge of the slower clock in which case we might induce meta stability. After latching the signals on the fast clock, we have a synchronized row, column, data and write enable that are at the proper frequency for writing to the ZBT. We also have the field parity bit and they are all now synchronized to the faster clock. The new generated signals are called x , y, we, data and eo. We currently have we that becomes high when data is ready to be written. We need to generate a signal that depends on the edge of we but that only goes high every other edge of we. Since we is synchronized with the x and y signals just computed and those signals count in order across the screen) (in other words, if we get a new row and a new column, meaning, the next x and y, we also get a new we because a new set of data just became valud), we can just use the lowest order bit of x and create a new signal called ntsc_we, that is only high every other edge of we. This ntsc_we will be high for one cycle every 2 valid pixels that we get. Therefore, we keep a running pixel buffer that is 36 bits and whose bits get shifted right by 18 bits every we edge, meaning every new pixel. When we have gotten 2 new pixels, this buffer will contain them and the ntsc_we will go high for once cycle causing the output data to latch the 2 new pixels instead of the 2 old ones etc. The address pertaining to those pixels is computed combinationally by the address module and the output address is updated to be synchronized with the new data. This address is computed using the camera_write_frame input as a highest bit.

Therefore, every time new data comes from the camera, the data and address output by the ntsc to zbt is held for at least two 65 Mhz clock cycles, and since read and writes happen on alternative clock cycles we can be certain that the data will indeed be written.

The ntsc to zbt module satisfies the requirement that every time the camera outputs two new pixels it presents them on the address and data output and holds them for at least two clock cycles so they are guaranteed to be read. It does not, however, provide a guarantee that there will be new data on every write cycle or provide a guarantee about how many cycles the data will be constant.

4. Camera Frame Swapper

The Frame Swapper has a reference address and a memorized maximum address and it compares the two with one another. If the reference address has reached the maximum address, the frame swapper decides that it is time to swap frames and swaps them. The two frames and the outputs of the frame swapper and they are each represented with one bit. The camera frame swapper operates in two modes plus an additional manual mode. The manual mode is the most straightforward: in this mode, the frame swapper isn't doing anything and we control the frame by hand with two switches. One switch communicates to the ntsc_2_zbt module what frame to write to and the other to the effect module telling it what frame to read from. Both of these frames refer to ZBT1 which is designated as the input SRAM, the Camera ram etc. The normal mode swaps the buffers according to the reference address and the maximum address. An explanation of these two addresses follows:

The reference address comes from the ntsc_to_zbt module and it is the output address of that module. It always represents the address that the ntsc_to_zbt module wants to write to next. Since the camera outputs data sequentially, the module will write to memory sequentially and after writing a whole frame it will reach the pixel with row 508 and column 720 which is the last pixel to be written. Since we're computing the address in by {frame, column/2, row}, we should set the max address to {360, 508} being its 18 least significant bits. The 19^{th} bit is not important since we want our swapper to swap buffers no matter what buffer is currently being written.

In swap_only_once mode, the buffer swapper will swap the write buffer once and then keep the camera always writing to the same buffer. The effect will continue swapping buffers as before. This allows us to overlay a freeze image (a still image) with a moving image , both of which can be manipulated.

We made a design choice to swap the buffers at the frequency of the camera which is lower than that of the display, meaning that we wait until the camera has output a new frame to swap the camera buffers. The other natural alternative would have been to swap buffers at the rate at which the display displays them. However, since the camera is the source of new information it seemed logical to use that as a guide for frame swapping.

5. General Effect Module

There is no one Effect Module. There are multiple effect modules and they each have different properties, take slightly different inputs and outputs. However, their backbone is

the same: They output a read address representing the memory address they wish to read. They take as input the data output by ZBT1. Their memory address is muxed with the write address issued by the ntsc_to_zbt depending on whether there is a write cycle or a read cycle. They output new data to be a written and a new address to write this data to. They also take effect_read_frame and effect_write_frame as inputs from the camera frame swapper and the display frame swapper respectively. Each effect does something different with the information that it is provided, but at a higher level they can all be thought of as functions that all have access to the same parameters and use them in different ways.
The specific contract of effect modules is that every time they get a pixel (meaning, every read cycle, which is every other cycle), they have to process it and output a new location and data and write that on a write cycle to the other memory.

The more information we can provide an Effect Module, the more interesting the effects that we can create. We initially thought that we would like an effect module to have access to the last two frames in order to determine differences between them, and also to an initial frame. However, we didn't have enough room for 3 frames to be stored. Eventually we figured out a way to pretend that we have the 3 frames without actually storing them all. The initial frame can not be circumvented, if we want it we must store it in one of our buffers . However, the preceding frame and the current frame can be obtained by reading a memory address on a read cycle and on the very next write cycle , having the camera overwrite that memory address. This can be done by outputting the address of the ntsc_to_zbt and giving it to the effect module. The effect module then issues a read request with that address (with the proper buffer bit) and we delay the datapath from ntsc_to_zbt to the memory by 2 cycles such that if ntsc_to_zbt issues a write request on a certain clock cycle, the address is first passed to the effect, read, and then overwritten on the next cycle. Ntsc_to_zbt can also pass to the effect module the data that it is going to write to memory, so that now the effect module has access to all 3 frames.

The effect modules also have to know whether the current cycle is a read or write cycle because for many timing must be precise.

However, the effect module still can't read the previous frame and the initial frame. It can read the initial frame and current frame (by stealing from camera) or the previous frame and current frame. However, the first is not terribly relevant because if there have been large changes from the initial frame, the current and previous frame should be good enough approximations of each other.

Effect modules process 2 pixels at once but are constrained to writing both to the same memory location. This is an unfortunate effect of our design choice: since we are reading 2 pixels at once, if we had to write them to two different locations we would end up with 2 write cycles per read cycle and that would completely alter the timing of the whole circuit. Therefore, we restrict effects to write to only one memory location. Pixels are thus stuck together forever.

In addition to the signals specified here, other signals from the circuit could be given as inputs to the effects, there is not any conceptual restriction preventing that.

6. Display Frame Swapper

The display frame swapper is virtually identical to the Camera frame swapper except less flexible. It is not really useful not to swap buffers on the output. Freezing the image on the output side is not nearly as interesting because the frozen image can not be further processed. However, it would be interesting to swap the reading buffer (read frame 1 then read frame 2 etc) and not swap the writing buffer. The image in that case would be a flickering swap between two different images.

7. XVGA

This module was borrowed entirely from Lab4. It has the specifications for XVGA 1024 * 768 display and generates hsync, vsync, blank and hcount and vcount. These are not reference to anything but they have the proper timing to display to the monitor. The lowest order bit of hcount is used as the write enable for both ZBTs and for the two muxes that select the address on read/write cycles for the ZBTs.

8. Vram Display

The Vram Display works as follows: it uses vcount and hcount which are issued by the xvga module and camera_read_frame issued by the display_buffer_swapper and computes the address that it wishes to read from memory. The display modules latches the output from memory and outputs one pixel every clock cycle. It gets a new pair of pixels out of memory every 2 clock cycles.

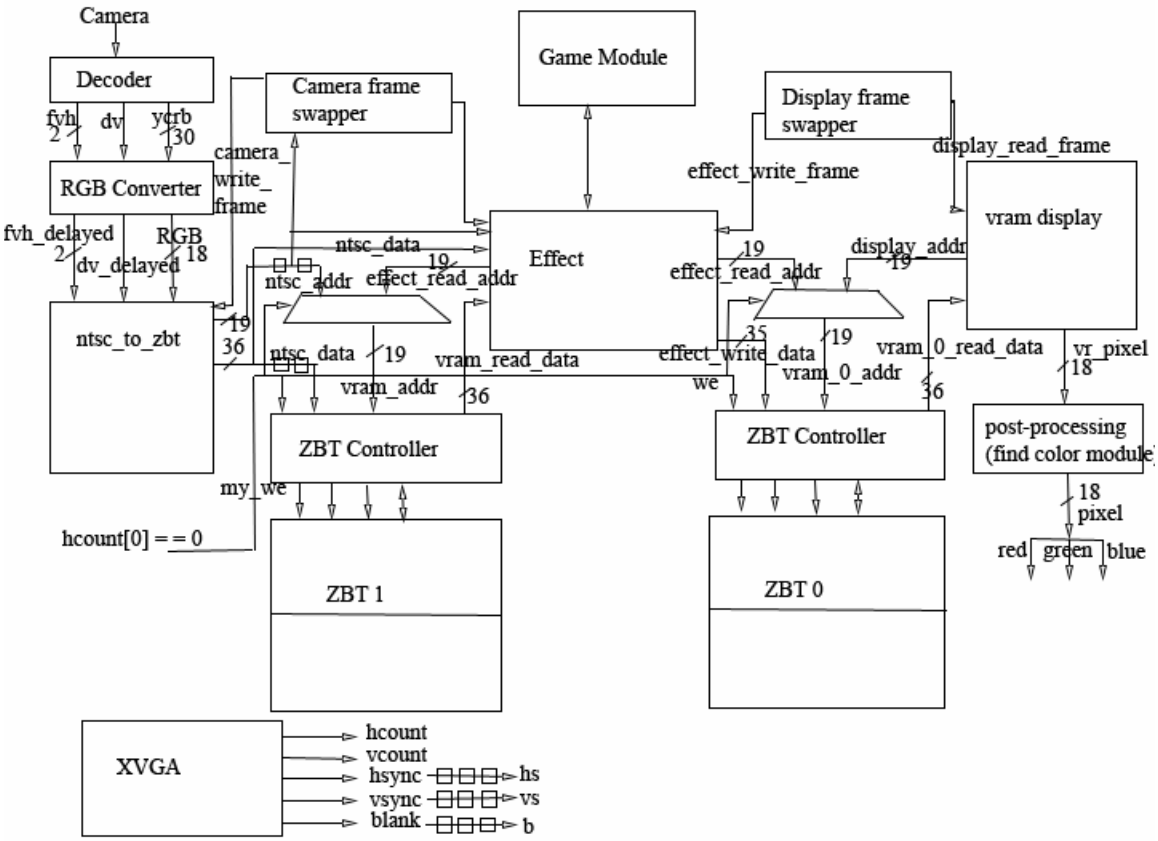


Figure 1: High-Level Block Diagram

Subsystem 2 (by Rebecca Arvanites)
Subsystem 2.1 Puzzle Game Module

The puzzle game is a sliding puzzle where the pieces are randomized blocks of video from the camera. Gameplay consists of sliding the pieces of video around the screen by using the buttons on the labkit to select a current square. There is a blank square which appears black and and a current square which is highlighted in blue. Pressing the left, right, up and down buttons moves the current square one block in the specified direction so long as the new location is onscreen. When the blank square and the current square are adjacent, pressing the enter button swaps the two squares.

The purpose of the puzzle game module is to communicate to the puzzle effect module where to display each block. This task involves keeping track of the position of each puzzle piece, including the current and blank squares, interpreting button presses, and checking when switching squares is allowed. Depending on the input square requested by the puzzle effect module, the puzzle game module outputs the location where that block of video should be displayed.

Two arrays of registers are used to store and access the location information requested by the puzzle effect module (see Fig. 1). The lookup_by_location array uses indices of absolute square locations to store which block is in each location at the moment. The lookup_by_block array uses indices of specific blocks to store which square location each block is in at the moment. Figure 1 shows the absolute square locations numbered 0 through 7, with block 4 in square location 0. When the effect module makes a request for a location, it is using the absolute square location that the current pixel being processed is in. Lookup_by_block of the index requested holds the new location of the block that was originally in the requested location when unscrambled. The number of columns and the width or height of the squares are used to convert from square location to the actual pixel location. The x-position is assigned by ((lookup_by_block[2] % num_col)* blob_width) and the y-position is assigned by ((lookup_by_block[2] / num_col)* blob_height).

Lookup_by_location

0	Stores index of lookup_by_block
1	
2	
3	
4	
5	
6	
7	

Lookup_by_block

0	Current_square
1	Blank_square
2	Stores index of lookup_by_location
3	
4	

Figure 1: Register Arrays to Store Square Location Information

5	
6	
7	
8	
9	

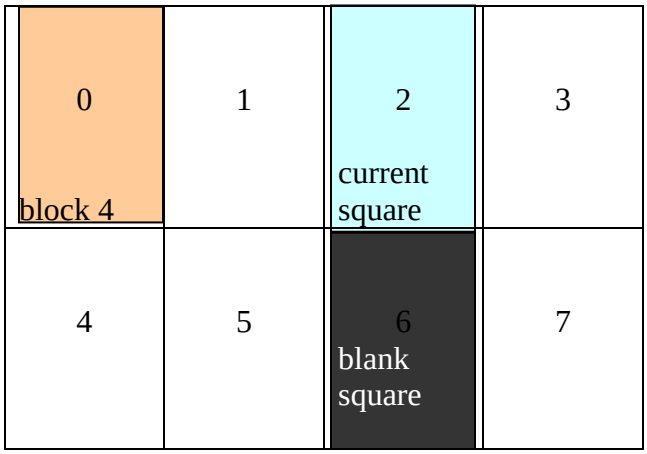


Fig. 2: Absolute square locations

The first two registers in the lookup_by_block array store the current square and blank square locations. When any of the direction buttons on the labkit are pressed, the module detects the first button press, checks if the current square is allowed to move in the indicated direction, and if so updates the location of the current square in the lookup_by_block array. If the current square is adjacent to the blank square and the enter button is pressed, the locations of the squares are updated by cross-referencing between the two arrays. The current and blank square locations are switched, the blocks stored in the current and blank square locations are switched, and the locations of the blocks are also switched.

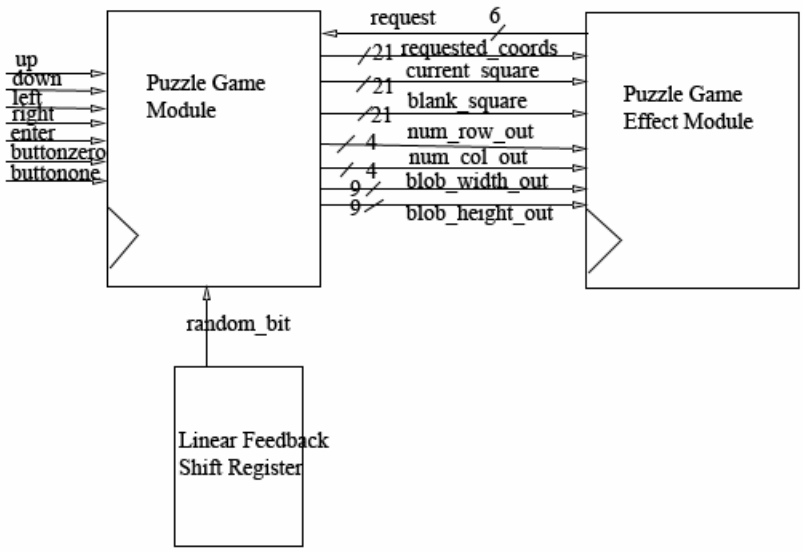


Fig 3 Puzzle Game Block Diagram

Another feature of the puzzle game is random initial scrambling of the video puzzle pieces after button one has been pressed to synchronously reset the random number generator, and button zero is pressed to scramble the puzzle pieces. Button one resets the puzzle pieces to their original ordered positions, and loads a seed value into the linear feedback shift register which then generates a pseudo-random sequence of zeros and ones. The pseudo-random one-bit output, sd_out, is input to the puzzle game module which uses the bit to decide whether to swap puzzle pieces zero with one, two with three, and so on, or pieces one with two, three with four, and so on. One of these swaps is done every clock cycle while button zero is pressed, which swap depending on whether sd_out is zero or one.

Subsystem 2.2 Falling Blob Game

In the falling blob game, 50 pixel wide squares and small circles are generated by the game's blob submodules and overlayed on top of the video to appear falling down the screen in front of the camera. Red objects and usually skin color is detected by the find_color module, such that the player can use a red card or their hand to interact with the game by catching the square falling objects and trying to avoid the round bombs (see Fig.). The hex display on the labkit keeps a running score with a point added for catching a square object and a point subtracted for making contact with a round bomb object.

Display of the falling objects is implemented by outputting 18-bit pixel data, blob_game_pixel, which is put into a mux with the 18-bit vr_pixel data from the vram_display module. The falling blob game outputs nonzero pixels for hcount, vcount where the blobs are positioned, and pixel data equal to zero in the absence of blobs. A mux chooses falling_blob_pixel for display when nonzero and otherwise displays vr_pixel which is the video from the camera.

Positions of the eight falling objects, four squares and four bombs, are updated every new frame on the positive edge of the clock. A counter that increments from 0 to 150

allows one square and one bomb to start falling, that is it allows the y-position to increment, every 50 new frames. The game module's pixel assignment also uses the counter to only display the blobs when they start moving downwards in order to stagger the appearance of the objects. When a blob object reaches the bottom of the 548 pixel screen, it regenerates at the top of the screen. Whenever a blob object is generated, a randomized x-position is assigned using the input pd_out from the linear feedback shift register.

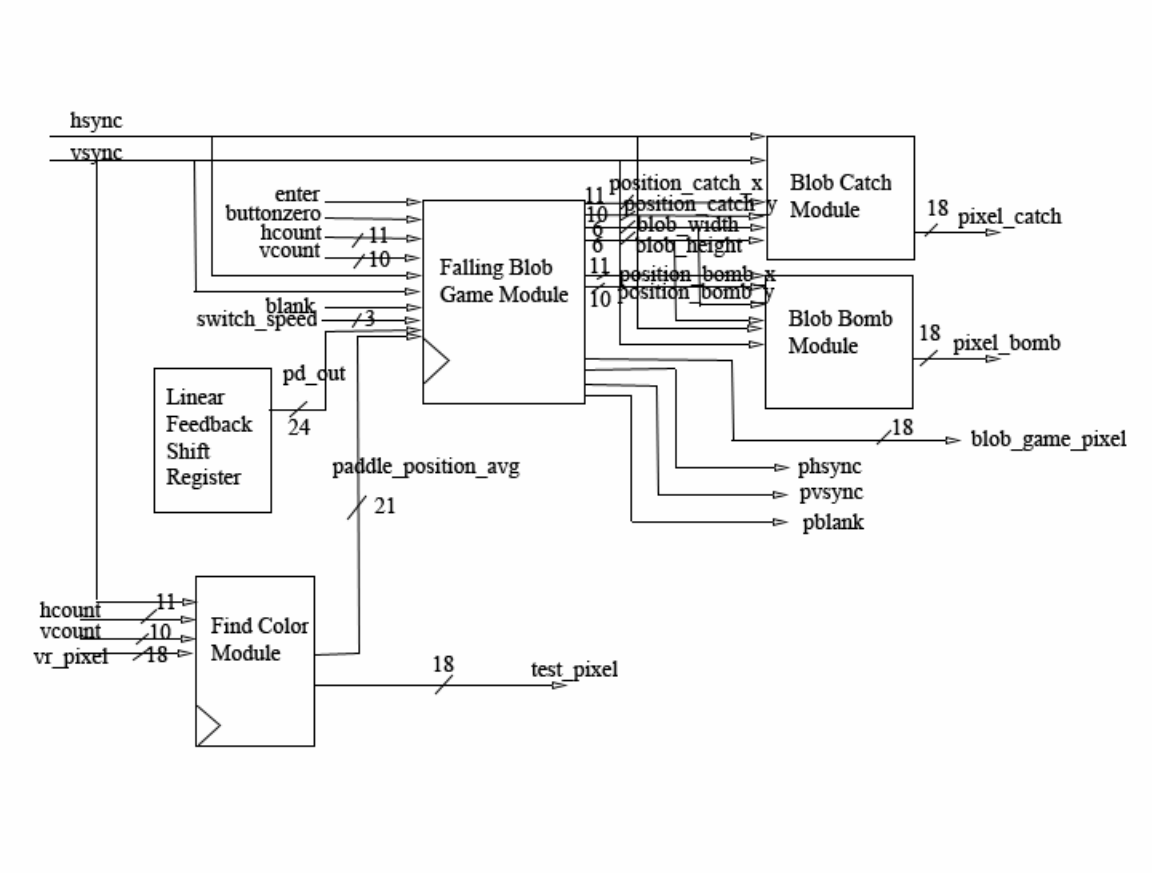
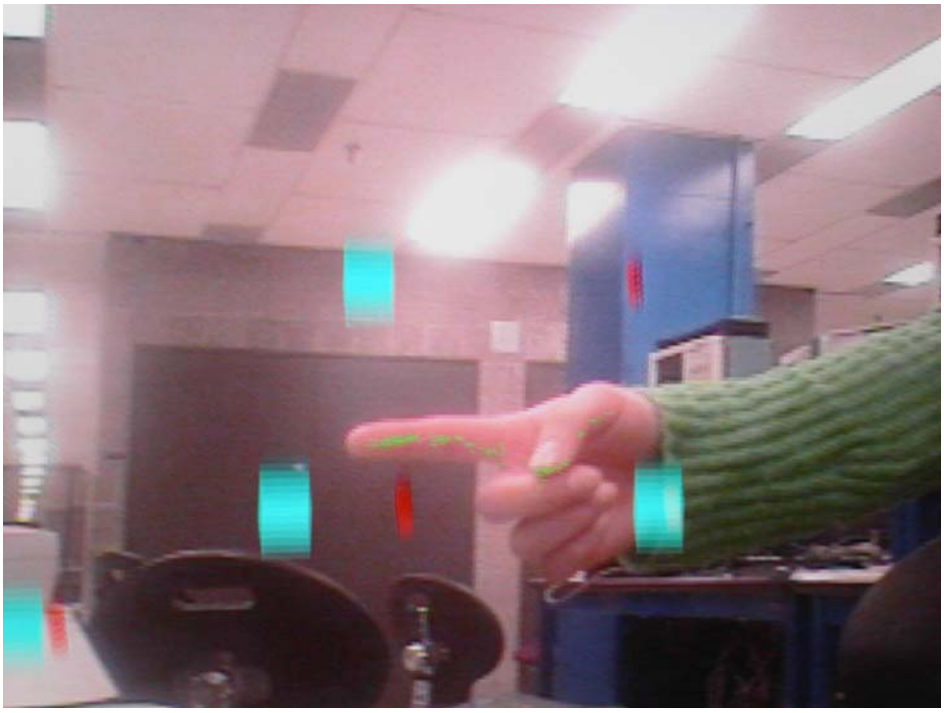


Figure : Falling Blob Game Module and Find Color Module Block Diagrams

The LFSR randgen contains 24 shift registers which use feedback to change their outputs every clock cycle. From the 24-bit output of the LFSR, different 9-bit intervals are assigned to each blob x-position when the blob begins to fall from the zero y-position. The speed at which the blobs move down the screen is set by the switches 4-7 on the labkit.

The interactive aspect of the falling blob game is created by the find_color module which communicates to the falling blob game module the position of the user controlled paddle distinguished by the color red. The game module uses the input paddle_position_avg to determine when the user's paddle has hit the game's blob objects. When the x-position of a blob is such that its x-maximum is larger than the paddle's x-minimum and its x-minimum is less than the paddle's x-maximum, and the y-positions also coincide, then the blob has hit the paddle. In this case the blob is regenerated at the top of the screen and the score is updated depending on whether the blob was a square or a bomb.



Falling Blob Game Screenshot 1

Subsystem 2.3 Linear Feedback Shift Register Module (Xilinx Core)

Two linear feedback shift register modules were used to generate pseudo-random numbers for use in the game modules. The chain of shift registers uses feedback to modify the state of each register so that the output sequence changese with every clock cycle. The output is either a one-bit serial output of the final register, as used for the puzzle game, or a parallel output bus of all the registers, as used for the falling blob game. The resulting sequence is only pseudo-random because it repeats through a mathematically determined cycle, but for the purposes of our games the results are satisfactorily random.

Subsystem 2.4 Find Color Module

The find_color module takes as input vr_pixel from the display module, and looks at the pixels which are being displayed on the screen to determine the average location of the first 128 red pixels in each frame. This paddle_position_avg is output to the falling blob game module to check for intersection with game objects so that the user can interact with the game.

A target red pixel value range which detected the red user paddle card well was determined experimentally, with a red value greater than 40 and green and blue values less than 28. The sensitivity to detecting red was found by replacing vr_pixel with entirely green pixels whenever vr_pixel was within the test range. Replacing red pixels with green pixels was kept for the final game version so that the player can see the area that the game recognizes as the paddle. The entire red area is not considered the paddle,

however, only a 20 pixel wide rectangle starting from the paddle_position_avg point is used for hit detection. The find_color module adds a dark blue rectangle where the paddle is positioned in the game, so that the player can see the in-game position of the paddle (see Fig.).

The average position value of the first 128 red pixels in the frame is calculated by comparing each vr_pixel to the range of red pixels described. When vr_pixel is in the correct color range, hcount_1 and vcount_1 are added to a running total of x and y position respectively, and a counter incremented to keep track of how red pixels have been found. Hcount_1 and vcount_1 are the x and y positions of vr_pixel, hcount and vcount from the vram_display module delayed by 3 clock cycles to allow for the vram_display module's processing of vr_pixel. When the counter reaches the number of pixels to be averaged, 128, the sum of x and y positions of red vr_pixel are bit-shifted by 7 places to calculate the average position of the red pixels, and the result is stored as paddle_position_avg and output to the falling blob game module to use for hit detection.

Falling Blob Game Screenshot 2, with paddle_position_avg shown in dark blue

Testing and Debugging (by Rebecca Arvanites)

Testing and debugging was easily the most time-consuming portion of this project. At every step of implementing the project, there were mistakes and problems at several levels that needed to be fixed. Early on we tried to separate our modules while we were making them so that we could test them separately before integrating them; even so, it was still a challenge to put together modules which we thought were working. With the puzzle game, the video modules were being developed, getting the memories and buffers working, at the same time as the game module was being developed. Each subsystem was

tested independently, and the puzzle game module was first implemented using color blobs as the puzzle pieces.

Before the cross-referencing register array structure was decided upon, the initial puzzle game contained cumbersome case statements to check every square location for a certain block. It was difficult to keep track of the squares because the game module was assigning the blob positions but could not refer to them to figure out which block was where. This led to confusing debugging where some of the blob squares were switching, but always with the same block, until we realized that the blocks being switched were using the absolute locations that the blocks were originally in instead of the changing locations. Then we came up with the idea of tracking every block and every location from the initial positions, so that we did not need to check every block to find which one is in a certain location, or check every location to find a certain block.

Integrating the puzzle game with the puzzle effect module also required many hours of debugging to get the modules working together. The coordinates provided by the game module needed to be translated for the effect module because the effect module was processing two pixels at the same time since our system stores two pixels in each address. Thus every hcount, vcount position intended for one pixel that the game module communicates to the effect module needed to be divided by two to allow space for two pixels instead of one. Once the blocks of video were being displayed, we then ran into what seemed to be a logic error since the blocks were switching predictably and smoothly, just in the wrong places. It seemed that the problem was with the game module making wrong switching assignments, but when we loaded the same code with blob blocks it was making the correct switches. After a while we realized that the problem was the effect module was using the wrong hcount and vcount to calculate the translation between absolute numbered squares and actual square positions of the video blocks.

Another part of the project that required extensive debugging was making the find_color module functional. The first step was finding a good threshold red value to search for, which was specific to the user's paddle or skin color but would not select large parts of the background. It was not difficult replacing the red pixels with green pixels, to show that the color was being detected, but taking an average of the location of the red pixels was more challenging. At first I was trying to use the vsync signal to control when to look for the specified color, as in look for the color and add the positions of the red pixels when vsync is high (in the middle of a frame) and then between frames when vsync goes low divide by the total number of pixels to compute the average position of the red or specified color pixels. However this did not work and for an unknown reason the sum and the average were never computed, as I saw by displaying the signals on the led hex display and observing that they were always zero even though the values the module was supposed to be adding were nonzero.

After testing some loops with counters that did not help, I used a new control signal called 'look' which when it is high causes the module to search for red pixels and when it is low the module does not search for red pixels. When look is high, the module looks for red pixels, adds their locations, and when the specified maximum counter number has been reached look is set to low and the average of the red pixel positions is computed. Look is reset to active high every new_frame, so that a new average of red pixel positions is taken for every frame.

Until I changed the control signals for this module, it was confusing that the average position was never computed. I used the led hex display to check that vsync was functioning, and it was, so I displayed the 3-cycle delayed hcount and vcount values, which were changing and should have been added to get a large sum and then divided to find the average. Eventually I decided to implement what I wanted in a another way, that is use new_frame to signal the module to start looking for red, and the counter to stop collecting position values and take the average.

Testing and Debugging (by Cristina Domnisoru)

Testing and Debugging were elaborate and time consuming. I tried to build my part of the project starting on Javier's video ram module, and afterwards realized that we needed ZBTs. I began reading the ZBT documentation and got very confused by the clocking and DLLs until Ike released some example code upon which it was easier to build.

Initially I was expecting the memory management and i/o part of the project to work seamlessly pretty quickly and to spend most of the time getting effects to work. Unfortunately, memory clocking was more involved than I expected. I was not able to really make any progress until I spend time thoroughly understanding the modules that I was trying to modify.

There were several bottlenecks in the process. A memorable one was trying to get the buffer swappers working. I knew, by the design of my addresses, that there had to be some maximum address that was getting written and based on NTSC standards I kept making guesses and checking them with my code. I was looking for a behavior where my guessed maximum would cause buffer swapping and that guessed maximum plus 1 would not. I spent a very long time trying to figure this out, thinking that perhaps my address was computed incorrectly until I decided to set aside my fear of the logic analyzer and ask it. I found out very quickly that the camera had a number of rows and columns that I would never have guessed. Someone more experienced would have known that NTSC has 480 relevant rows, not total rows. The logic analyzer made it easy to figure out what the camera was doing. I thus learned that even if I feel confident of my own modules and I can make a timing diagram to justify their behavior, there can always be surprises when interfacing with devices and code that one did not write. Being able to observe the actual behavior of the system at that point becomes crucial.

I realized throughout this project that I have a problem that I am not sure how to address. If I am trying to figure out a bug, I make implicit and unconscious assumptions about what I think it is and what I think it isn't, so that even if I check all the possibilities I will not carefully check the ones that I don't expect. This is difficult to overcome and it caused me the biggest waste of time during this project. For a 3 day period my display had periodic black lines repeating every other pixel. Since 2 pixels are stored in memory at a time I thought that this might be a problem with the processing of one of the two pixels and that it is systematic. I noticed that these lines were not occurring in the noise coming from memory in the parts that weren't being overwritten. I noticed that if I wrote the pixels in memory reflected, the lines were still vertical. This indicated a problem after the memory read location so in the effect or afterwards. Eventually I started replacing the pixel data along the datapath with white, module after module, to isolate the problem. I

eventually isolated it to the effect module. This was not surprising because the lines were not there when I had only one memory and read/write modules. However, the effect module consisted of 3 lines of code none of which were affecting the data. They were simply copying it from one memory to another. The system was deliberately constructed such that the first effect added would do nothing. I then looked at the warnings and discovered that one of my wires was the wrong size and that I was outputting 19 bits instead of 36. The remaining bits were padded to 0s and that explains the black. The bug itself was very frustrating because it made me think that there was some fundamental flaw in my timing design, but the resolution of the bug made me think that there is a fundamental flaw in my debugging technique.

For the most part of the project I developed it and tested it little by little, taking snapshots every time something worked and every time something really didn't work but did something unexpected. I kept debugging logs (textfiles) in which I wrote what I was thinking about what was and what was not working. Over and over I reached the conclusion that if something doesn't work, the change this bit of code and hope for the best approach is bad. It is bad because if the change doesn't fix the code you might then try another bit of code etc, thus wasting a lot more time than thinking about it would have taken. If it does work then you lose understanding of your code, and after a few lucky guesses you no longer really know why its' working so that next time it breaks it will be much more difficult to figure out. At the end of this project I think I prefer the understanding and going straight for the target rather than fiddling and experimentation approach.

I spent a very long time trying to get the double buffering system working and it was only in the last few days before the due date that I was able to actually focus on what I wanted which is video effect processing. When I started developing effects, working on the project changed in that I didn't really have any specific goals but I was playing around trying to get a multitude of things to work. The effects that I developed are not specific except for the puzzle effect. They are explorations and involve tweaking thresholds and taking pictures of the screen. Effect modules have their own debugging spin. It happens very frequently that I do not understand why a certain effect occurred or if I change variables the system reacts in an unpredictable fashion. I realized that the best way to test an idea in terms of an effect is to provide oneself with switches that can change the parameters of the algorithm, because there might be ranges of parameters that work much better than others. The effect exploration part of the project was fun in certain ways but frustrating in other ways.

The following section illustrates my explorations in digital effect creation.

Puzzle Effect

The puzzle effect module is a simple effect. For each pixel, the module figures out which Square (or rectangle, in this image) it belongs to. The effect then issues a request to the Puzzle module to ask where that block went. The effect then checks whether that pixel is in the blank blobk or in the current block (current block is the block we are selecting right now and then maps the pixel to the correct place in memory

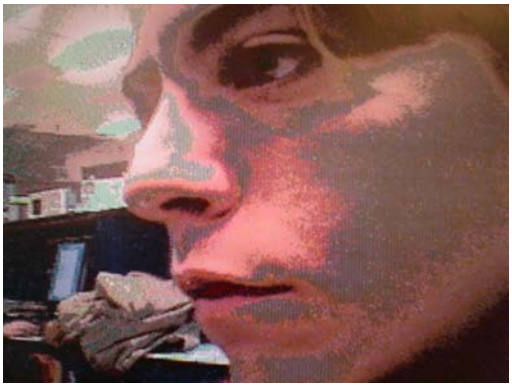
without changing its value. This puzzle is conceptually simple but the computation it needs to do is involved enough that we can't easily stack other effects on top of the puzzle effect without causing glitches.

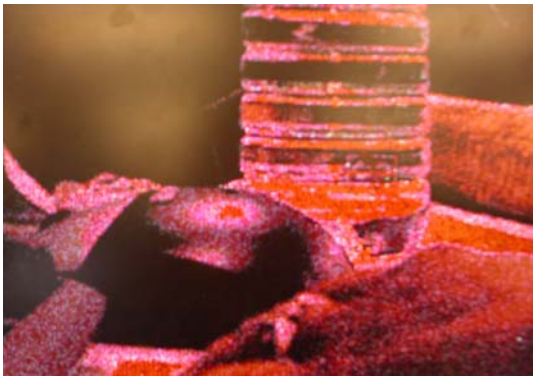
Digital Effect Results

Simple Effects

Simple effects only operate on the current pixel. Such effects would not require the double buffering system.

The image below is reddified.
The image to the left is inverted
The image in the corner has a color filter applied that changes certain colors. It is difficult to specify such effects because RGB also encodes luminance. Virtually any effect that one can think of that is a computable function of the position and color of one pixel can be implemented. Currently, the restriction on these simple effects is time. It is possible to extend the amount of time available for computation of a pixel by latching and pipelining. In the current implementation the delaying has to be done by hand depending on the particular computation.





Frame Swapping Schizophrenia
This effect flips alternating frames. When viewed, it looks like a flickering image that is flipping every second. It is difficult to get a picture where both symmetrical sides are in focus. Stopping the frames from flipping implements a pretty reliable random number generator, where heads and tails represent the real image and the flipped image.

Edge Detector (weffect.v)

This effect compares the current frame that the camera (ntsc_to_zbt) is trying to write to the frame previously written in memory. The algorithm is very simple : For each channel of the two pixels being processed, take the highest order 3 bits and if the current and previous pixel differ by more then 1, write a white pixel in that memory location. Otherwise, write a black pixel in that memory location. A white pixel will be written if a condition like the following is true for all the channels.

((cam_w_delayed_twice_copy_data[12:10] - undelayed_e_write_data1[12:10]) > 1'b1)||
((undelayed_e_write_data1[12:10]- cam_w_delayed_twice_copy_data[12:10]) > 1'b1)

Figure 1 Edge Detection

This was not the intended effect. I was hoping to detect things that had changed from the previous frame in a way that would allow one to detect motion. However, from frame to frame a person is not capable of moving fast enough to cause a perceptible difference. The only pixels that are changing fast enough are noise and edges. Edges are continuous in reality but they become jagged when displayed on a discrete pixel grid. Pixels at object boundaries are constantly changing from belonging to one object or another and are doing so at a frequency that is comparable to the frame rate of the camera. In a way, this edge detector works because of noise.

Effects involving freeze frames (based on blueoff.v and weirdo.v)

This family of effects was achieved by turning the camera frame swapping to manual and at some point

flipping a switch such that the camera is writing always to one of the buffers while the effect is reading from the other one. The effect is essentially stuck always reading the same picture. However, the effect has access to the image that the camera is writing, pixel by pixel, so this allows us to display various things depending on their relationship to the image that was there before.

Basing the effect on freeze frame properties

This image was taken by pointing the camera at my mouth, flipping the switch so that the effect is no longer getting updates, and applying a certain threshold to the computation so that if the pixels in the freeze frame have certain RGB values, they effect will write the freeze frame to memory, and otherwise it will write the current frame to memory. By manipulating the color thresholds this allows us to replace parts of images with other images. This image was taken with a digital camera. In reality, the big open mouth is standing still and the face inside can move around. The threshold can be set manually using switches (up to 3 bits)

Figure 2 using blueoff.v

Figure 3 freeze frame

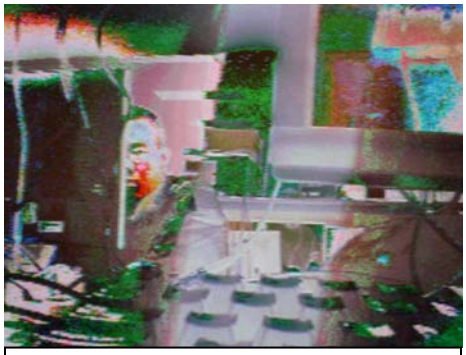


Figure 4 freeze frame, different threshold

The picture with the mouth uses the base algorithm, in which both images look correct color-wise. If we introduce delays in the signals that are subtracted, for example delay the current frame by one cycle, or affect certain bits, the images retain their overlayed aspect but become much more arbitrary. We thus get pretty noise such as in the images above.

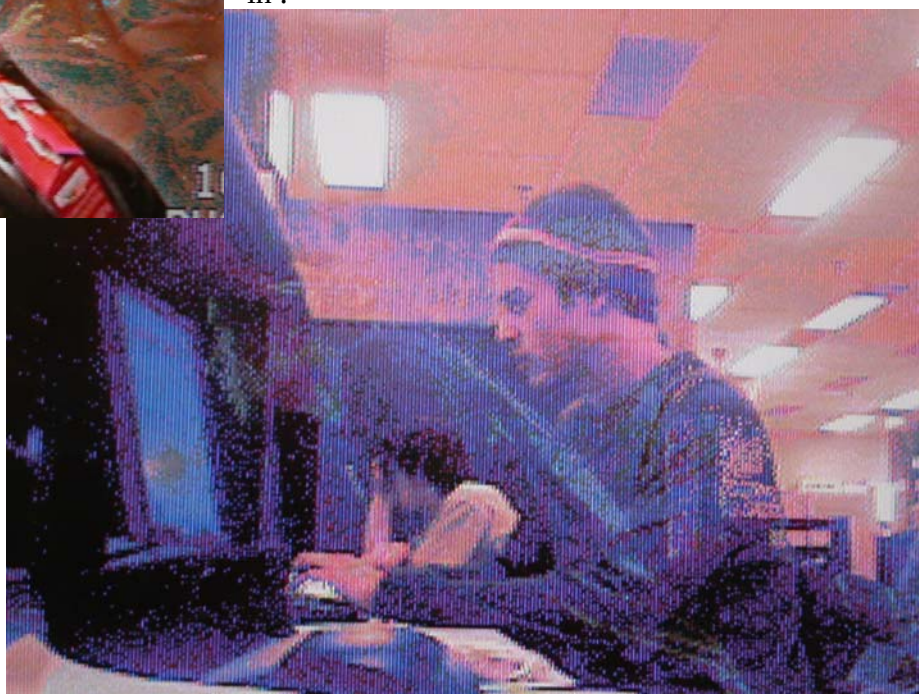
In this category of effects, in general, a deformation is based on the properties of the freeze frame. This would allow us to produce images where people put their face on a funny body (the standard amusement park trick) or where we replace particular parts of images with something else. The effect structure is very general and if we were able to accurately determine the RGB values of a color we would have great control over this effect. It would be interesting to find the value of skin color, even if restricted to certain lighting, and replace only that. This way, when people walked across the screen, their skin would become affected by whatever algorithm we desire (or replaced by another image etc)

Basing the effect on current frame properties

This category is in a way equivalent to the previous one. It does the opposite in that the effect is dependent on the properties of the current image. In this image I took a freeze frame of a 100 euro bill with the map of Italy and France and then set the algorithm to show the freeze frame image only if the current image had certain properties in that area. The current image itselve was altered to pink and green. (There is a map of Italy on the sweater in the picture).

Effects based on the difference between the current frame and the freeze frame

This category of effects is one that I was particularly interested in .

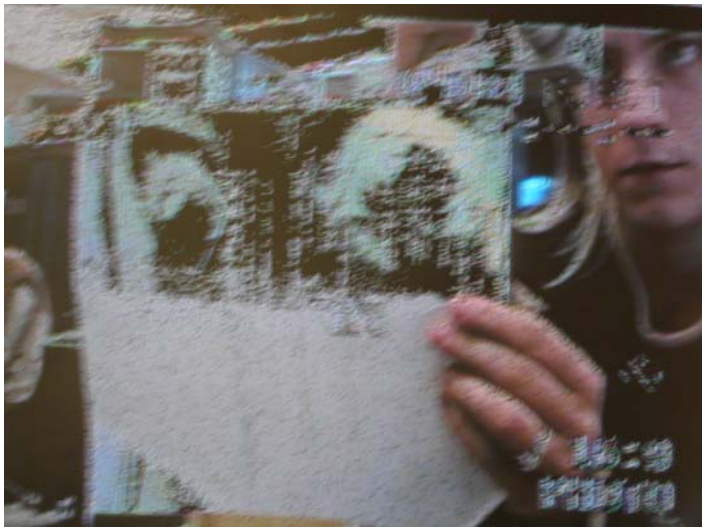


In this image you can see a boy sitting in a chair and a ghostlike copy of that boy bending towards the monitor. This image was taken by freezing the frame when the boy was standing in the chair and the subsequent motion was ghostlike. This kind of effect can be achieved by comparing the current frame and the freeze frame and based on their difference decide whether they are the same. I found that the tweaking of this process was very difficult and that if I did too much arithmetic the image would gain artifacts (such as turn pink like the one above). This image is a variation of the above where new things that appear onto the image have a shimmery yellow/blue like effect. The foreground is a hand and the yellow boy in the back wasn't there when the freeze frame was taken.

The image below shows another equivalent attempt, except that this one caused the shadows to be red. The hand that's frozen is the one above whereas the one blow is actually moving.

I had a lot of problems understanding how come certain effects are happening the way they are, in particular color changes in situations where I am not changing the color at all. The process of affecting the colors that we want was hard until I thought of the idea of taking a freeze frame of the color that I want and then looking for it in the image rather than trying to guess it ahead of time. This had better results :

In this picture, my leg is showing a pattern that was frozen in the other buffer. The image on the right shows a face that is frozen but



can only be seen if one holds a white paper over it (or skin color etc).

Conclusions (by Rebecca Arvanites)

By the end of the project, we had developed a library of varied video effects that could be added in parallel with the games we developed. The games were functional and entertaining to play, especially given all the hard work it had taken to get everything working. Some aspects that we had previously thought would be trivial parts of the total project, such as displaying the video as block puzzle pieces, turned out to be more challenging than expected. Initially we decided to implement the puzzle game as our first task because we considered it the most simple or at least straight-forward. However, the integration took longer than expected due to a small logic error which became harder to find once the system consisted of all modules combined and interacting.

The puzzle game was implemented as we designed it, although our final version was a four by two configuration which we realized may not actually be solvable for some random initializations. Our parametrized design allows us to easily change the number of squares to fix this, the number of rows and columns being limited by the computation limits of the fpga to powers of two.

The fire game was implemented fairly close to what we discussed, without the final touch of adding video effects to the player when they miss objects or are hit by bombs. When we designed the game we were not exactly sure how we would enable the player to interact with the game, we thought about edge detection to find the person onscreen, as well as color detection. The color detection at first seemed like it would not be very robust, but we were pleased to find that tweaking the color range produced fairly consistent results.

We completed our checklist of developing the specific puzzle and falling blob game, video, and effect modules, however we wish we had more time to explore the additional ideas that came up with while working on the project. Given more time we would like to play with storing previous camera frames and trying to implement a video effect where objects on camera only appear while moving, and when they stop moving they fade into the background. We would also like to add features to the falling blob game, such as the originally intended effects applied when the player misses catching objects or is hit by bombs, or use a ROM to make the blob objects that player catches into specific items such as cats or ice cream cones.

Digital Effects Conclusion (by Cristina Domnisoru)

Given the setup of our system it has been easy to come up with various visual effects, some of which are interesting, some of which are reproducible. The best so far happened by accident and I can not reproduce it. It slowly dissolved moving objects into the background but if one stood still it one would appear on the screen. I have a video recording of it but I don’t know why it happened . Many effects were created and overwritten after pictures were taken of them. The files contained in the project currently are the backbones of the basic types of effects. Making digital effects is not hard. Making the digital effect that you had in mind is more difficult.

-

If I did this again from the beginning:

I would change my expectations of the amount of time various parts would require. In terms of design, I would implement the 4 buffers without specifically tying 2 to display and 2 to input, but rather I would have a single buffer swapper that issues the signals for all those buffers. That way, if an effect didn't' need to use the double buffering capability of the input, it could have 3 frames instead of 2 for output and processing and that would open up a new array of possibilities.

Fvhdelayer Module

```
//////////////////////////////////////////////////////////////////////
// purpose: to take 30-bit ycrcb pixel input from the video decoder module and the fvh
//and dv signals issued by that module and convert the pixel data into RGB. Also to delay
//the fvh and dv signals to account for the RGB conversion delay
//
/*Inputs:  - reset, clk (system reset, system clock 65 Mhz)
         - fvh_in1 are 3 bits given by the video decoder module and they represent
           a field bit (for even and odd field), vsync and hsync as issued by the camera
         -data_valid_in1 is the data valid bit issued by the video decoder module. If high it
         means that the ycrcb input is a valid input that corresponds to the camera.
         -ycrcb is the 30 bit colored pixel output by the camera (processed by the video
         decoder module)
Outputs: - fvh_out is the 5 cycle delayed version of fvh_in1
          -data_valid_out is the 5 cycle delayed version of data_valid_in1
          -RGB is the 18 bit equivalent in RGB of the 30 bit ycrcb input. The 18 bits are
          the 6 most significant bits of R, G, B in that order.
*/
//////////////////////////////////////////////////////////////////////



module fvhdelayer(reset, clk, fvh_in1, fvh_out, data_valid_in1,
                  data_valid_out ,ycrcb, RGB);
        input reset, clk;
        input [2:0] fvh_in1; // 3 bits given by the video decoder module
        input data_valid_in1; // issued by the video decoder module.
        input [29:0] ycrcb; //30 bit colored pixel output by the camera

        output [2:0] fvh_out; // 5 cycle delayed version of fvh_in1

        output data_valid_out; //5 cycle delayed version of data_valid_in1
        output [17:0] RGB; //Red, Green, Blue, 6 bits each

        wire [7:0] R, G, B;

        //this is the one of the Xilinx ycrcb to RGB converters
        //it takes 5 clock cycles to compute
```

```
        YCrCb2RGB converter (.R(R), .G(G) , .B(B) , .clk(clk) ,
                .rst(reset) , .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]),
                .Cb(ycrcb[9:0]));

        //select the most significant 6 bits of the R G B channels
        assign RGB [5:0] = B[7:2];
        assign RGB [11:6] = G[7:2];
        assign RGB [17:12] = R[7:2];

        /*delay fvh_out, data_valid_out by 5 clock cycles because the
          CrCb2RGB converter takes 5 clock cycles to
          compute the proper RGB values. */

        reg [2:0] fvh_in2, fvh_in3, fvh_in4, fvh_in5, fvh_in6;
        reg [2:0] fvh_out;
        reg data_valid_in2, data_valid_in3, data_valid_in4,
        reg data_valid_in5, data_valid_in6;
        reg data_valid_out;

        always @ (posedge clk)
                begin
                data_valid_in2 <= data_valid_in1;
                data_valid_in3 <= data_valid_in2;
                data_valid_in4 <= data_valid_in3;
                data_valid_in5 <= data_valid_in4;

                data_valid_out <= data_valid_in5;

                fvh_in2 <= fvh_in1;
                fvh_in3 <= fvh_in2;
                fvh_in4 <= fvh_in3;
                fvh_in5 <= fvh_in4;

                fvh_out <= fvh_in5;
                end
        endmodule
```

Ntsc_to_zbt Module

```
// Written by: I. Chuang <ichuang@mit.edu>
// Modified by : Cristina Domnisoru
//
//
//
//The original version of this module writes 4 8-bit pixels to one
//memory address
//every 4 clock cycles. This version writes 2 18-bit pixels to one
//memory address
//every 2 clock cycles. In the current version the address to be written
//is computed
//differently from the original, taking a camera_write_frame bit input.
//this input is either 0 or 1
//and will represent the highest order bit of the addresses being
//written. This allows an
//external user to control which buffer the module is writing data to
////////////////////////////////////////////////////////////////////////

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data,
                   ntsc_we, camera_write_frame);

   input         clk;   // system clock
   input         vclk;  // video clock from camera
```

```
input [2:0]  fvh; // delayed signals from delayer module
input        dv; // delayed data valid to match fvh and din
input [17:0]         din; //RGB data converted from camera
input camera_write_frame; //bit indicating which buffer to write to

output [18:0] ntsc_addr; //address to write
output [35:0] ntsc_data; //data to write to that address. Contains 2
                         //pixels
output       ntsc_we;       // write enable for NTSC data


//the image will be aligned to these 2 start values.
parameter    COL_START = 10'd0;
parameter    ROW_START = 10'd0;


//the row and col registers are used to compute the current
//coordinates of the pixel based on fvh.


//This module receives input which changes depending on vclock, the
//clock of the camera, and outputs data which changes on the edges of
//clk, the 65 Mhz clock used in the rest of the system. The module
//must synchronize the data.

reg [9:0]    col = 0; //synchronized to vclock
reg [9:0]    row = 0; //vclock
reg [17:0]   vdata = 0; //vclock
reg          vwe; //vclock
reg          old_dv;
reg          old_frame;     // frames are even / odd interlaced
reg          even_odd;      // decode interlaced frame to this wire

wire camera_write_frame;

wire         frame = fvh[2];
wire         frame_edge = frame & ~old_frame;


always @ (posedge vclk)     //LLC1 is reference
  begin
     old_dv <= dv;
     //this means data just became valid
     vwe <= dv && !fvh[2] & ~old_dv;
     old_frame <= frame;
     even_odd = frame_edge ? ~even_odd : even_odd;

     //all of the meaningful data from the camera is transmitted with
     //f bit being 0, and it alternates by transmitting all odd rows,
     //then all even rows, then all odd rows etc.
     if (!fvh[2])
       begin
          col <= fvh[0] ? COL_START :
                 (!fvh[2] && !fvh[1] && dv && (col < 1024)) ?
                    col + 1 : col;

          row <= fvh[1] ? ROW_START :
                 (!fvh[2] && fvh[0]  && (row < 768)) ? row + 1 : row;

          vdata <= (dv && !fvh[2]) ? din : vdata;

       end
  end
```

```
   // synchronize with system clock

   reg [9:0] x[1:0],y[1:0];
   reg [17:0] data[1:0];
   reg       we[1:0];
   reg       eo[1:0];

   //latch twice to make sure data is stable in case the first latching
   //occurred too close to a rising clock edge of the faster clock.
   always @(posedge clk)
     begin
        {x[1],x[0]} <= {x[0],col};
        {y[1],y[0]} <= {y[0],row};
        {data[1],data[0]} <= {data[0],vdata};
        {we[1],we[0]} <= {we[0],vwe};
        {eo[1],eo[0]} <= {eo[0],even_odd};
     end

   // edge detection on write enable signal



   reg old_we;
   wire we_edge = we[1] & ~old_we;
   always @(posedge clk) old_we <= we[1];


   //
   reg [35:0] mydata;
   always @(posedge clk)
     if (we_edge)
        //if we just got a we, shift the data to accommodate the newly
        //acquired pixel in the lowest order 18 bits of mydata
       mydata = { mydata[17:0], data[1] };

   // compute address to store data in

   wire [18:0] myaddr;

   //compute the hcount and vcount equivalents to get the address.
   //the eo bit is for even frame or odd frame.
   //since the camera outputs all the odd rows or all the even rows and
   //then issues a vsync, we must know whether we are have odd rows or
   //even rows and multiply the relevant vertical counter by 2 and add
   //the even/odd bit.

   wire [9:0] local_vcount  = { y[1][8:0], eo[1] };
   wire [10:0] local_hcount = { 1'b0, x[1]};

   address altern(local_hcount, local_vcount, myaddr,
camera_write_frame);

   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;

   //ignore every other we_edge. Still update mydata but only every
   //other clock cycle, aka when the column's lowest order bit is 0 ,
   //issue a we_edge. This ensures that a we_edge is only happening
   //every other time the data is valid so that until the next we_edge
   //we have time to get another 2 pixels

   wire       ntsc_we = (we_edge & (x[1][0]==1'b0));
```

```
   always @(posedge clk)
     //only update the output values every other clock cycle
     if ( ntsc_we )
       begin
          //make sure the highest order bit if the address is the frame
          ntsc_addr <=  {camera_write_frame, myaddr[17:0]};
          ntsc_data <=  mydata;
       end


endmodule // ntsc_to_zbt
```

Camera_frame_swapper Module

```
/*purpose : to control the 2 buffers located in memory 1 that are used
by the camera and by the effect to read. This module has two main modes
of operation. One of them is normal buffer swapping, the other is
swapping once where the module begins by writing to one frame and then
the other frame but it never returns to the first frame it wrote.

Inputs :
       - clk, reset (system clock and reset)
       - address_currently_being_written = the address that the swapper
       Uses to figure out that it is time to swap buffers. The simplest
way to tell that it is time to swap the camera buffer is to look at the
address that ntsc_to_zbt just issued a write request for. Since this
module writes in order, when it gets to the maximum address that it
will write to, it will loop back to the beginning. The maximum address
can be computed with the address function and by knowing the maximum
row and column that the module will reach. In this case, the number is
written in binary and was obtained from the logic analyzer. Note that
only 18 bits of the address are considered. This is because the 19th bit
is the frame bit and we want to swap frames regardless of the current
frame.
       -swap_only_once. If this is 1, the camera buffer will start at 0
and switch to 1 and never return to 0.

Outputs: -write frame
         -read frame

These are bit inputs and they are always the opposite of each other in
normal operation. They are initialized to 0, 1 and they swap values
every time the address_currently_being_written reaches the maximum
value.
*/


module camera_frame_swapper(clk, reset,
       address_currently_being_written, write_frame, read_frame,
       swap_only_once);

       input clk; // clock
       input reset;
       input [18:0] address_currently_being_written; // reference
address
       input swap_only_once; //control input
```

```
        output write_frame; //frame control
        output read_frame;

        //initialize frames
        reg write_frame = 0;
        reg read_frame = 1;

        //this wire has value 1 whenever the current address has reached
        //maximum value.
        wire addr_eq_max =  (address_currently_being_written[17:0]
               == 18'b101101000111111101);

        reg old_addr_eq_max = 0;

        wire addr_eq_max_edge ;
        assign addr_eq_max_edge = addr_eq_max & ~old_addr_eq_max;

        //if we have an edge , that mans the address just reached max
and we want to swap buffers
        always @ (posedge clk)
            begin
                old_addr_eq_max <= addr_eq_max;

                if ((swap_only_once == 0) || (swap_only_once == 1 &&
                     write_frame == 0))
                        begin
                            write_frame <= addr_eq_max_edge
                                ? ~write_frame : write_frame;
                            read_frame = ~write_frame;
                        end
                else if (swap_only_once == 1)
                        read_frame = sfft_eq_max_edge
                                ? ~read_frame : read_frame;
            end
        endmodule
```

Display_frame_swapper Module

```
/* purpose: swap display buffers
This module is virtually identical to the camera swapping module
It is a simplified version of that module and more detailed comments
are provided there. Unlike the camera swapping module, this module
always swaps buffers. Like the camera swapping module, this module can
be turned off altogether and replaced with manual buffer control.

module display_frame_swapper(clk,reset,address_currently_being_written,
                             write_frame, read_frame);
        input clk; // system clock
        input reset; // system reset
        input [18:0] address_currently_being_written; // reference
address

        output write_frame; // control frames
        output read_frame;

        reg write_frame = 0;
```

```
        reg read_frame = 1;

        wire addr_eq_max;

        assign addr_eq_max =  (address_currently_being_written[17:0]
                               == 18'b101101000111111101);
        reg old_addr_eq_max = 0;

        wire addr_eq_max_edge ;
        assign addr_eq_max_edge = addr_eq_max & ~old_addr_eq_max;

        //

        always @ (posedge clk)
                begin
                        old_addr_eq_max <= addr_eq_max;
                        write_frame <= addr_eq_max_edge
                                ? ~write_frame:write_frame;
                        read_frame = ~write_frame;
                end
        endmodule
```

Vram_Display Module

/*
Purpose: Read pixels out of the display buffer from ZBT 0 and output them to screen

Inputs: reset, clk;

Hcount, vcount : takes these signals from the xvga module and computes an address to read from memory

Display_read_frame : takes this input from display_frame_swapper and uses it to compute highest order bit of address to read from memory

Vram_read_data : data output by memory

Outputs:

Vr_pixel : 18 bit single pixel to display to screen

Vram_addr : address to issue read request to memory

*/

```
module vram_display(reset,clk,hcount,vcount,vr_pixel,
                    vram_addr,vram_read_data,test,display_read_frame);

   input reset, clk;
   input [10:0] hcount;
   input [9:0]  vcount;
   input test;
   input display_read_frame;

   output [17:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;


   wire [18:0] alternate_addr;
   address newaddress(hcount, vcount, alternate_addr,
                      display_read_frame);
```

```
   wire [18:0] vram_addr = test? {vcount, hcount[9:1]}:alternate_addr;

   wire hc2 = hcount[0];
   reg [17:0]   vr_pixel;
   reg [35:0]   vr_data_latched;
   reg [35:0]   last_vr_data;


   always @(posedge clk)
        last_vr_data <= (hc2 == 1'b0) ? vr_data_latched: last_vr_data;
        vr_data_latched <= (hc2 == 1'b1)
                ? vram_read_data :vr_data_latched;

   // must display the last data latched from memory for the next 2
//cycles while  waiting for the new read request to be processed by
//memory
   always @(hc2)              /*each 36-bit word from RAM is decoded
                                to 4 bytes */
        case (hc2)

          1'b0: vr_pixel = last_vr_data [35:18];
          1'b1: vr_pixel = last_vr_data [17:0];

        endcase
endmodule
```

Puzzle Game Module

```
//////////////////////////////////////////////////////////////////////
//
/*purpose: tell puzzle effect module where to display each block of
video
Inputs:
-reset, clk: system reset, 65mhz clock
-request: 6-bit input from puzzle effect module specifies a request for
a square location
-up,down,left,right,enter: buttons to move current square
-buttonone: initializes random number generator, puts squares in correct
original locations
-buttonzero: scrambles puzzle
-random_bit: sd_out from linear feedback shift register
Outputs:
-current_square, blank_square: hcount,vcount locations of current and
blank squares
-requested_coords: hcount,vcount locations of the requested square for
puzzle effect module
-num_row_out, num_col_out, blob_width_out, blob_height_out: parameters
*/
//////////////////////////////////////////////////////////////////////


module puzzle_game (clk, reset, up, down, left, right, enter,
   buttonzero, buttonone, random_bit, current_square, blank_square,
   requested_coords, num_row_out, num_col_out, request, blob_width_out,
   blob_height_out);

   input [5:0] request;
   input vclock;    // 65MHz clock
   input reset;
   input up;
   input down;
   input left;
   input right;
```

```
input enter;
input buttonone;
input buttonzero;
input random_bit;   /*random bit is sd_out from linear feedback shift
                           register rngen  */
output [20:0] current_square;
output [20:0] blank_square;
output [20:0] requested_coords;
output [3:0] num_row_out;
output [3:0] num_col_out;
output [8:0] blob_width_out, blob_height_out;

parameter  num_squares = 8;
parameter coords_index = 21*(num_squares)-1;
parameter screen_width = 720;
parameter screen_height = 548;
parameter max_width = 360;
parameter max_height = 508;
parameter num_row = 2;
parameter num_col = 4;

parameter blob_width = 180;
parameter blob_height = 254;

reg [3:0] num_row_out = num_row;
reg [3:0] num_col_out = num_col;

wire [20:0] current_square;
wire [20:0] blank_square;

/*puzzle square positions: stored in 8 registers size [20:0], [20:10]
xpos, [9:0] ypos; positions of squares from left to right, then top
to bottom
*/

wire [20:0] position [num_squares-1:0];

     /*array indices are the square locations which don't change,
array values are the lookup_by_block indices which refer to a
specific block
*/

reg [3:0] lookup_by_location [num_squares-1:0];
reg [3:0] lookup_by_block [num_squares+1:0];
/* need location for each block, current_square, and blank_square;
   lookup_by_block[0] is current_square, lookup_by_block[1] is
   blank_square     */

reg game_over = 0;  //game_over =1 stops play
reg old_right=0;
wire press_right;
assign press_right = ~old_right & right;
reg old_left=0;
wire press_left;
assign press_left = ~old_left & left;
reg old_up=0;
wire press_up;
assign press_up = ~old_up & up;
reg old_down=0;
wire press_down;
assign press_down = ~old_down & down;
reg old_enter=0;
wire press_enter;
assign press_enter = ~old_enter & enter;
```

```
   reg old_buttonzero=0;
   wire press_buttonzero;
   assign press_buttonzero = ~old_buttonzero & buttonzero;

   /* button 1 resets squares to correct position, initializes random
      number generator,
      button 0 scrambles squares      */

always @(posedge vclock) begin
    old_vsync <= vsync;

   if (buttonone) begin
         game_over <= 0;
         lookup_by_location[0]<= 4'd2;
         lookup_by_location[1]<= 4'd3;
         lookup_by_location[2]<= 4'd4;
         lookup_by_location[3]<= 4'd5;
         lookup_by_location[4]<= 4'd6;
         lookup_by_location[5]<= 4'd7;
         lookup_by_location[6]<= 4'd8;
         lookup_by_location[7]<= 4'd9;

         lookup_by_block[0]<= 4'd0;
         lookup_by_block[1]<= 4'd1;
         lookup_by_block[2]<= 4'd0;
         lookup_by_block[3]<= 4'd1;
         lookup_by_block[4]<= 4'd2;
         lookup_by_block[5]<= 4'd3;
         lookup_by_block[6]<= 4'd4;
         lookup_by_block[7]<= 4'd5;
         lookup_by_block[8]<= 4'd6;
         lookup_by_block[9]<= 4'd7;
         end   //end if buttonone

   else if (buttonzero) begin
         if (random_bit) begin
               lookup_by_location[0]<= lookup_by_location[1];
               lookup_by_location[1]<= lookup_by_location[0];
               lookup_by_location[2]<= lookup_by_location[3];
               lookup_by_location[3]<= lookup_by_location[2];
               lookup_by_location[4]<= lookup_by_location[5];
               lookup_by_location[5]<= lookup_by_location[4];
               lookup_by_location[6]<= lookup_by_location[7];
               lookup_by_location[7]<= lookup_by_location[6];

               lookup_by_block[lookup_by_location[0]]<=
                     lookup_by_block[lookup_by_location[1]];
               lookup_by_block[lookup_by_location[1]]<=
                     lookup_by_block[lookup_by_location[0]];
               lookup_by_block[lookup_by_location[2]]<=
                     lookup_by_block[lookup_by_location[3]];
               lookup_by_block[lookup_by_location[3]]<=
                     lookup_by_block[lookup_by_location[2]];
               lookup_by_block[lookup_by_location[4]]<=
                     lookup_by_block[lookup_by_location[5]];
               lookup_by_block[lookup_by_location[5]]<=
                     lookup_by_block[lookup_by_location[4]];
               lookup_by_block[lookup_by_location[6]]<=
                     lookup_by_block[lookup_by_location[7]];
               lookup_by_block[lookup_by_location[7]]<=
                     lookup_by_block[lookup_by_location[6]];

               end    //end if random_bit
```

```
      else begin
            lookup_by_location[0]<= lookup_by_location[7];
            lookup_by_location[1]<= lookup_by_location[2];
            lookup_by_location[2]<= lookup_by_location[1];
            lookup_by_location[3]<= lookup_by_location[4];
            lookup_by_location[4]<= lookup_by_location[3];
            lookup_by_location[5]<= lookup_by_location[6];
            lookup_by_location[6]<= lookup_by_location[5];
            lookup_by_location[7]<= lookup_by_location[0];

            lookup_by_block[lookup_by_location[0]]<=
                  lookup_by_block[lookup_by_location[7]];
            lookup_by_block[lookup_by_location[1]]<=
                  lookup_by_block[lookup_by_location[2]];
            lookup_by_block[lookup_by_location[2]]<=
                  lookup_by_block[lookup_by_location[1]];
            lookup_by_block[lookup_by_location[3]]<=
                  lookup_by_block[lookup_by_location[4]];
            lookup_by_block[lookup_by_location[4]]<=
                  lookup_by_block[lookup_by_location[3]];
            lookup_by_block[lookup_by_location[5]]<=
                  lookup_by_block[lookup_by_location[6]];
            lookup_by_block[lookup_by_location[6]]<=
                  lookup_by_block[lookup_by_location[5]];
            lookup_by_block[lookup_by_location[7]]<=
                  lookup_by_block[lookup_by_location[0]];
            end
      end   //end if buttonzero

else if     //game_over if squares in original locations
      (lookup_by_location[0]== 4'd2&&
      lookup_by_location[1] == 4'd3 &&
      lookup_by_location[2] == 4'd4 &&
      lookup_by_location[3] == 4'd5 &&
      lookup_by_location[4] == 4'd6 &&
      lookup_by_location[5] == 4'd7 &&
      lookup_by_location[6] == 4'd8 &&
      lookup_by_location[7] == 4'd9
      )
            game_over <=1;

else
     old_right <= right;
     old_left <= left;
     old_up <= up;
     old_down <= down;
     old_enter <= enter;
     old_buttonzero <= buttonzero;

//when buttons pressed check if should move current_square
if    (press_right &&
      ((lookup_by_block[0])/num_col==(lookup_by_block[0]+1)/num_col))
            lookup_by_block[0] <= lookup_by_block[0]+1;
else if (press_left && (lookup_by_block[0] % num_col)>0)
            lookup_by_block[0] <= lookup_by_block[0]-1;
else if (press_up && (lookup_by_block[0]+1>num_col))
            lookup_by_block[0] <= lookup_by_block[0]-num_col;
else if (press_down && (num_squares-num_col>lookup_by_block[0]))
            lookup_by_block[0] <= lookup_by_block[0]+num_col;

//if current_square is next to blank_square user can swap the squares
else if (press_enter)
     if(
       (
```

```
          ((lookup_by_block[0]+1==lookup_by_block[1]) ||
           (lookup_by_block[1]+1==lookup_by_block[0]))
                &&
           (lookup_by_block[0]/num_col ==lookup_by_block[1]/num_col)) ||
           (lookup_by_block[0]+num_col==lookup_by_block[1]) ||
           (lookup_by_block[1]+num_col==lookup_by_block[0])
           )
                begin //beginB
                      /*switch blank_square and current_square in
                        lookup_by_location */

                      lookup_by_block[0] <= lookup_by_block[1];
                      lookup_by_block[1] <= lookup_by_block[0];

                      //switch squares in lookup_by_location

                      lookup_by_location[lookup_by_block[0]] <=
                            lookup_by_location[lookup_by_block[1]];
                       lookup_by_location[lookup_by_block[1]] <=
                            lookup_by_location[lookup_by_block[0]];

                //switch actual blocks in lookup_by_block

               lookup_by_block[lookup_by_location[lookup_by_block[0]]]
   <= lookup_by_block[lookup_by_location[lookup_by_block[1]]];
               lookup_by_block[lookup_by_location[lookup_by_block[1]]]
   <= lookup_by_block[lookup_by_location[lookup_by_block[0]]];

                end //end beginB
      end //end else if ~game_over
end //end @ posedge vclock

//need to assign pixel locations from square locations

   assign position [0][20:10] = ((lookup_by_block[2] % num_col)*
        blob_width);
   assign position [0][9:0] = ((lookup_by_block[2] / num_col)*
        blob_height);
   assign position [1][20:10] = ((lookup_by_block[3] % num_col)*
        blob_width);
   assign position [1][9:0] = ((lookup_by_block[3] / num_col)*
        blob_height);
   assign position [2][20:10] = ((lookup_by_block[4] % num_col)*
        blob_width);
   assign position [2][9:0] = ((lookup_by_block[4] / num_col)*
        blob_height);
   assign position [3][20:10] = ((lookup_by_block[5] % num_col)*
        blob_width);
   assign position [3][9:0] = ((lookup_by_block[5] / num_col)*
        blob_height);
   assign position [4][20:10] = ((lookup_by_block[6] % num_col)*
        blob_width);
   assign position [4][9:0] = ((lookup_by_block[6] / num_col)*
        blob_height);
   assign position [5][20:10] = ((lookup_by_block[7] % num_col)*
        blob_width);
   assign position [5][9:0] = ((lookup_by_block[7] / num_col)*
        blob_height);
   assign position [6][20:10] = ((lookup_by_block[8] % num_col)*
        blob_width);
   assign position [6][9:0] = ((lookup_by_block[8] / num_col)*
        blob_height);
   assign position [7][20:10] = ((lookup_by_block[9] % num_col)*
        blob_width);
```

```
   assign position [7][9:0] = ((lookup_by_block[9] / num_col)*
        blob_height);

   assign current_square[20:10] = ((lookup_by_block[0] % num_col)*
        blob_width);
   assign current_square[9:0] = ((lookup_by_block[0] / num_col)*
        blob_height);
   assign blank_square[20:10] = ((lookup_by_block[1] % num_col)*
        blob_width);
   assign blank_square[9:0] =  ((lookup_by_block[1] / num_col)*
        blob_height);

   assign requested_coords [20:0] = position [ request ];
endmodule
```

Falling Blob Game, Blob Submodule

```
//////////////////////////////////////////////////////////////////////
//
//purpose: generates pixels of square objects
/*Inputs:
-x,y: upper-left position of square blob
-width, height: dimensions of square blob
-hcount,vcount: current pixel location being processed
Output:
-pixel: pixel color to display
*/
//////////////////////////////////////////////////////////////////////

module blob (x,y,width,height,hcount,vcount,pixel);
    input [5:0] width;
    input [5:0] height;
    parameter COLOR = 3'b011;    //default color: --

    input [10:0]x,hcount;
    input [9:0]y,vcount;
    output [2:0] pixel;
    reg [2:0] pixel;

    always @(x or y or hcount or vcount) begin
           if ((hcount >= x && hcount < (x+width)) &&
            (vcount >= y && vcount < (y+height)))
            pixel = COLOR;
            else pixel = 0;
    end
endmodule
```

Falling Blob Game, Blob_bomb Submodule

```
//////////////////////////////////////////////////////////////////////
//
//purpose: generates pixels of round bomb objects
/*Inputs:
-x,y: upper-left position of round blob
-width, height: dimensions of blob
-hcount,vcount: current pixel location being processed
Output:
-pixel: pixel color to display
*/
//////////////////////////////////////////////////////////////////////

module blob_bomb (x,y,width,height,hcount,vcount,pixel);
```

```
    input [5:0] width;
    input [5:0] height;
    parameter COLOR = 3'b100;

    input [10:0]x,hcount;
    input [9:0]y,vcount;
    output [2:0] pixel;
    reg [2:0] pixel;

    wire [5:0] r_sqr;
    assign r_sqr = width*width/4;
    wire [10:0] x_1;
    assign x_1 = x+(width/2);
    wire [9:0] y_1;
    assign y_1 = y+(height/2);

    always @(x_1 or y_1 or hcount or vcount) begin
         if (
              (hcount+width>=x_1) &&
              (hcount<= (x_1+width)) &&
              (vcount+width>=y_1) &&
              (vcount<=(y_1+height))
             )
              if (
                       ((x_1-hcount)*(x_1-hcount) +
                    (y_1-vcount)*(y_1-vcount)) <= r_sqr
                      ||
                   ((hcount-x_1)*(hcount-x_1) +
                    (vcount-y_1)*(vcount-y_1) <= r_sqr))
                     pixel = COLOR;
              else pixel = 0;
   end
endmodule
```

Falling Blob Game Module

```
//////////////////////////////////////////////////////////////////////
//
/*purpose: manage positions of blobs displayed over video, determine
intersection between player and blobs, display player's game score
*/
//////////////////////////////////////////////////////////////////////

/*buttonzero initializes random number generator,
  press_enter resets game */

module falling_blob_game (vclock,reset,enter,buttonzero, hcount, vcount,
   hsync, vsync, blank, switch_speed, pd_out, paddle_position_avg,
   phsync, pvsync, pblank, blob_game_pixel, game_over, data_for_hex);

   input vclock;         // 65MHz clock
   input reset;
   input enter;         //press_enter used as reset
   input buttonzero;
   input [10:0] hcount; // horizontal index of current pixel (0..1023)
   input [9:0]  vcount; // vertical index of current pixel (0..767)
   input hsync;         // XVGA horizontal sync signal (active low)
   input vsync;         // XVGA vertical sync signal (active low)
   input blank;         // XVGA blanking (1 means output black pixel)
   input [2:0] switch_speed;   /*speed that objects fall down screen,
                                 set by switches [7:4] */
   input [23:0] pd_out;
   input [20:0] paddle_position_avg;  /*current upper-left coordinate
                                        of user-controlled paddle */
```

```
output phsync;       // game's horizontal sync
output pvsync;       // game's vertical sync
output pblank;       // game's blanking
output [17:0] blob_game_pixel;       // game's pixel
output game_over              //when game_over is 1, gameplay stops
output [63:0] data_for_hex;   //to display game score

parameter screen_width = 720;
parameter screen_height = 520;
parameter blob_width = 6'd30;
parameter blob_height = 6'd30;
parameter paddle_width = 20;

wire [2:0] pixel;
reg [7:0]  counter = 0;   //counter used to delay appearance of blobs
reg [63:0] data_for_hex;
reg [6:0]  score;
reg [10:0] position_catch_x1;
reg [10:0] position_catch_x2;
reg [10:0] position_catch_x3;
reg [10:0] position_catch_x4;
reg [9:0]  position_catch_y1=1;
reg [9:0]  position_catch_y2=1;
reg [9:0]  position_catch_y3=1;
reg [9:0]  position_catch_y4=1;
reg [10:0] position_bomb_x1;
reg [10:0] position_bomb_x2;
reg [10:0] position_bomb_x3;
reg [10:0] position_bomb_x4;
reg [9:0]  position_bomb_y1=1;
reg [9:0]  position_bomb_y2=1;
reg [9:0]  position_bomb_y3=1;
reg [9:0]  position_bomb_y4=1;
wire [3:0] speed;
assign speed = 2*switch_speed;

wire [2:0] pixel_catch1;
wire [2:0] pixel_catch2;
wire [2:0] pixel_catch3;
wire [2:0] pixel_catch4;
wire [2:0] pixel_bomb1;
wire [2:0] pixel_bomb2;
wire [2:0] pixel_bomb3;
wire [2:0] pixel_bomb4;

reg old_vsync;
wire new_frame;
assign new_frame = old_vsync & ~vsync;
reg old_enter=0;
wire press_enter;
assign press_enter = ~old_enter & enter;
reg old_buttonzero=0;
wire press_buttonzero;
assign press_buttonzero = ~old_buttonzero & buttonzero;

assign phsync = hsync;
assign pvsync = vsync;
assign pblank = blank;

blob catch1(position_catch_x1,position_catch_y1,blob_width,
    blob_height,hcount, vcount, pixel_catch1);
blob catch2(position_catch_x2, position_catch_y2,blob_width,
    blob_height, hcount, vcount, pixel_catch2);
blob catch3(position_catch_x3, position_catch_y3,blob_width,
```

```
        blob_height, hcount, vcount, pixel_catch3);
    blob catch4(position_catch_x4, position_catch_y4,blob_width,
        blob_height, hcount, vcount, pixel_catch4);

    blob_bomb bomb1(position_bomb_x1, position_bomb_y1,blob_width,
        blob_height, hcount, vcount, pixel_bomb1);
    blob_bomb bomb2(position_bomb_x2, position_bomb_y2,blob_width,
        blob_height, hcount, vcount, pixel_bomb2);
    blob_bomb bomb3(position_bomb_x3, position_bomb_y3,blob_width,
        blob_height, hcount, vcount, pixel_bomb3);
    blob_bomb bomb4(position_bomb_x4, position_bomb_y4,blob_width,
        blob_height, hcount, vcount, pixel_bomb4);

    //new pixel_catch and pixel_bomb appear every 50 new_frames

    assign pixel = counter>149 ?
             pixel_catch1 | pixel_catch2 | pixel_catch3 | pixel_catch4 |
             pixel_bomb1 | pixel_bomb2 | pixel_bomb3 | pixel_bomb4

         : counter>99 ?
             pixel_catch1 | pixel_catch2 | pixel_catch3| pixel_bomb1 |
             pixel_bomb2 | pixel_bomb3

         : counter>49  ?
             pixel_catch1 | pixel_catch2 | pixel_bomb1 | pixel_bomb2

         : pixel_catch1 | pixel_bomb1;

    assign blob_game_pixel[17:12] = (pixel[2]==1) ?  6'b111111 : 6'b0;
    assign blob_game_pixel[11:6] = (pixel[1]==1) ? 6'b111111 : 6'b0;
    assign blob_game_pixel[5:0] = (pixel[0]==1) ? 6'b111111 : 6'b0;

//hit detection using paddle_position_avg from find_color module
    wire [10:0]paddle_x;
    wire [9:0]paddle_y;
    assign paddle_x = paddle_position_avg[20:10];
    assign paddle_y = paddle_position_avg[9:0];
    wire [7:0] hit;   /*if 0-3 catch objects were caught or 4-7 bombs
                        were hit (1 hit, 0 nothing) */
    assign hit[0] = ( (position_catch_y1 >= paddle_y) &&
              ((position_catch_x1+blob_width) >paddle_x) &&
               (position_catch_x1<(paddle_x+paddle_width)) )
                 ? 1 : 0;
    assign hit[1] = ( (position_catch_y2 >= paddle_y) &&
              ((position_catch_x2+blob_width) >paddle_x) &&
               (position_catch_x2<(paddle_x+paddle_width)) )
                 ? 1 : 0;
    assign hit[2] = ( (position_catch_y3 >= paddle_y) &&
              ((position_catch_x3+blob_width) >paddle_x) &&
               (position_catch_x3<(paddle_x+paddle_width)) )
                 ? 1 : 0;
    assign hit[3] = ( (position_catch_y4 >= paddle_y) &&
              ((position_catch_x4+blob_width) >paddle_x) &&
               (position_catch_x4<(paddle_x+paddle_width)) )
                 ? 1 : 0;
    assign hit[4] = ( (position_bomb_y1 >= paddle_y) &&
              ((position_bomb_x1+blob_width) >paddle_x) &&
               (position_bomb_x1<(paddle_x+paddle_width)) )
                 ? 1 : 0;
    assign hit[5] = ( (position_bomb_y2 >= paddle_y) &&
              ((position_bomb_x2+blob_width) >paddle_x) &&
               (position_bomb_x2<(paddle_x+paddle_width)) )
                 ? 1 : 0;
    assign hit[6] = ( (position_bomb_y3 >= paddle_y) &&
```

```
            ((position_bomb_x3+blob_width) >paddle_x) &&
             (position_bomb_x3<(paddle_x+paddle_width)) )
               ? 1 : 0;
   assign hit[7] = ( (position_bomb_y4 >= paddle_y) &&
            ((position_bomb_x4+blob_width) >paddle_x) &&
             (position_bomb_x4<(paddle_x+paddle_width)) )
               ? 1 : 0;

   //player loses points for missing square blobs
   wire [3:0] blob_miss;
   assign blob_miss[0] = (position_catch_y1==screen_height) ? 1 : 0;
   assign blob_miss[1] = (position_catch_y2==screen_height) ? 1 : 0;
   assign blob_miss[2] = (position_catch_y3==screen_height) ? 1 : 0;
   assign blob_miss[3] = (position_catch_y4==screen_height) ? 1 : 0;

   always @(posedge vclock) begin
        old_vsync <= vsync;
        if (new_frame) begin
             old_enter <= enter;
             old_buttonzero <= buttonzero;
             counter <= press_enter ? 0
               : counter<150 ? counter + 1
               : counter;

//update positions of blobs and player score

             position_catch_x1 <= (hit[0] || press_enter ||
               (position_catch_y1 + speed + blob_height)>screen_height)
               ? pd_out[8:0] : position_catch_x1;

             position_catch_x2 <= (hit[1] || press_enter ||
               (position_catch_y2 + speed + blob_height)>screen_height)
               ? pd_out[12:4] : position_catch_x2;

             position_catch_x3 <= (hit[2] || press_enter ||
               (position_catch_y3 + speed + blob_height)>screen_height)
               ? pd_out[14:6]+80 : position_catch_x3;

            position_catch_x4 <= (hit[3] || press_enter ||
               (position_catch_y4 + speed + blob_height)>screen_height)
               ? pd_out[16:8] : position_catch_x4;

            position_bomb_x1 <= (hit[4] || press_enter ||
               (position_bomb_y1 + speed + blob_height)>screen_height)
               ? pd_out[17:9] : position_bomb_x1;

            position_bomb_x2 <= (hit[5] || press_enter ||
               (position_bomb_y2 + speed + blob_height)>screen_height)
               ? pd_out[15:7]+80 : position_bomb_x2;

            position_bomb_x3 <= (hit[6] || press_enter ||
               (position_bomb_y3 + speed + blob_height)>screen_height)
               ? pd_out[13:5]+50 : position_bomb_x3;

            position_bomb_x4 <= (hit[7] || press_enter ||
               (position_bomb_y4 + speed + blob_height)>screen_height)
               ? pd_out[11:3] : position_bomb_x4;

            position_catch_y1 <= (hit[0] || press_enter ||
               (position_catch_y1 + speed + blob_height)>screen_height)
               ? 0 : (position_catch_y1 + speed);

            position_catch_y2 <= (hit[1] || counter<50 || press_enter ||
               (position_catch_y2 + speed + blob_height)>screen_height)
```

```
                ? 0 : (position_catch_y2 + speed);

            position_catch_y3 <= (hit[2] || counter<100 || press_enter||
                (position_catch_y3 + speed + blob_height)>screen_height)
                ? 0 : (position_catch_y3 + speed);

            position_catch_y4 <= (hit[3] || counter<150 || press_enter||
                (position_catch_y4 + speed + blob_height)>screen_height)
                ? 0 : (position_catch_y4 + speed);

            position_bomb_y1  <= (hit[4] || press_enter ||
                (position_bomb_y1 + speed + blob_height)>screen_height)
                ? 0 : (position_catch_y1 + speed);

            position_bomb_y2  <= (hit[5] || counter<50 || press_enter ||
                (position_bomb_y2 + speed + blob_height)>screen_height)
                ? 0 : (position_catch_y2 + speed);

            position_bomb_y3  <= (hit[6] || counter<100 || press_enter||
                (position_bomb_y3 + speed + blob_height)>screen_height)
                ? 0 : (position_catch_y3 + speed);

            position_bomb_y4  <= (hit[7] || counter<150 || press_enter||
                (position_bomb_y4 + speed + blob_height)>screen_height)
                ? 0 : (position_catch_y4 + speed);
            end //end else if ~game_over

    //update score each frame depending on value of hit
    score <= score +hit[0]+hit[1]+hit[2]+hit[3]-
             hit[4]-hit[5]-hit[6]-hit[7];
    data_for_hex <= {57'b0,score};

    end //end new_frame
end //end @ posedge vclock

endmodule
```

Rngen Module (Xilinx Core)

```
////////////////////////////////////////////////////////////////////////////
//
/*purpose: generate pseudo-random 1-bit output to scramble puzzle game
*/
////////////////////////////////////////////////////////////////////////////


/*****************************************************************
*********
*     This file is owned and controlled by Xilinx and must be used
*
*     solely for design, simulation, implementation and creation of
*
*     design files limited to Xilinx devices or technologies. Use
*
*     with non-Xilinx devices or technologies is expressly prohibited
*
*     and immediately terminates your license.
*
*
*
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
*
```

```
*      SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR
*
*      XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION
*
*      AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION
*
*      OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS
*
*      IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
*
*      AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
*
*      FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
*
*      WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
*
*      IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
*
*      REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
*
*      INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
*
*      FOR A PARTICULAR PURPOSE.
*
*
*
*      Xilinx products are not intended for use in life support
*
*      appliances, devices, or systems. Use in such applications are
*
*      expressly prohibited.
*
*
*
*      (c) Copyright 1995-2004 Xilinx, Inc.
*
*      All rights reserved.
*
**********************************************************************
********/
// The synopsys directives "translate_off/translate_on" specified below
are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity
synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file rngen.v when simulating
// the core, rngen. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Guide".

module rngen (
        clk,
        sd_out,
        sinit);    // synthesis black_box

input clk;
```

```
output sd_out;
input sinit;

// synopsys translate_off

        LFSR_V3_0 #(
                "0000000001",  // c_ainit_val
                0,      // c_enable_rlocs
                0,      // c_gate
                0,      // c_has_ainit
                0,      // c_has_ce
                0,      // c_has_data_valid
                0,      // c_has_load
                0,      // c_has_load_taps
                0,      // c_has_new_seed
                0,      // c_has_pd_in
                0,      // c_has_pd_out
                0,      // c_has_sd_in
                1,      // c_has_sd_out
                1,      // c_has_sinit
                0,      // c_has_taps_in
                0,      // c_has_term_cnt
                0,      // c_implementation
                0,      // c_max_len_logic
                0,      // c_max_len_logic_type
                "0000000001",  // c_sinit_val
                10,     // c_size
                "0000001001",  // c_tap_pos
                0)      // c_type
        inst (
                .CLK(clk),
                .SD_OUT(sd_out),
                .SINIT(sinit),
                .PD_OUT(),
                .LOAD(),
                .PD_IN(),
                .SD_IN(),
                .CE(),
                .DATA_VALID(),
                .LOAD_TAPS(),
                .TAPS_IN(),
                .AINIT(),
                .NEW_SEED(),
                .TERM_CNT());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of rngen is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of rngen is "black_box"

endmodule
```

Randgen Module (Xilinx Core)

```
////////////////////////////////////////////////////////////////////////////////
//
/*purpose: generate pseudo-random 24-bit output to generate random x-
positions for falling blob game blobs
*/
////////////////////////////////////////////////////////////////////////////////
/*****************************************************************************
********
*     This file is owned and controlled by Xilinx and must be used
*
*     solely for design, simulation, implementation and creation of
*
*     design files limited to Xilinx devices or technologies. Use
*
*     with non-Xilinx devices or technologies is expressly prohibited
*
*     and immediately terminates your license.
*
*
*
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
*
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR
*
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION
*
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION
*
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS
*
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
*
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
*
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
*
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
*
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
*
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
*
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
*
*     FOR A PARTICULAR PURPOSE.
*
*
*
*     Xilinx products are not intended for use in life support
*
*     appliances, devices, or systems. Use in such applications are
*
*     expressly prohibited.
*
*
*
*     (c) Copyright 1995-2004 Xilinx, Inc.
*
*     All rights reserved.
*
*****************************************************************************
*******/
```

```
// The synopsys directives "translate_off/translate_on" specified below
are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity
synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file randgen.v when simulating
// the core, randgen. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Guide".

module randgen (
   clk,
   pd_out,
   load,
   pd_in,
   sinit);    // synthesis black_box

input clk;
output [23 : 0] pd_out;
input load;
input [23 : 0] pd_in;
input sinit;

// synopsys translate_off

   LFSR_V3_0 #(
        "000000000000000000000001",    // c_ainit_val
        0,   // c_enable_rlocs
        0,   // c_gate
        0,   // c_has_ainit
        0,   // c_has_ce
        0,   // c_has_data_valid
        1,   // c_has_load
        0,   // c_has_load_taps
        0,   // c_has_new_seed
        1,   // c_has_pd_in
        1,   // c_has_pd_out
        0,   // c_has_sd_in
        0,   // c_has_sd_out
        1,   // c_has_sinit
        0,   // c_has_taps_in
        0,   // c_has_term_cnt
        0,   // c_implementation
        0,   // c_max_len_logic
        0,   // c_max_len_logic_type
        "000000000000000000000001",    // c_sinit_val
        24,  // c_size
        "000000000000000010000111",    // c_tap_pos
        0)   // c_type
   inst (
        .CLK(clk),
        .PD_OUT(pd_out),
        .LOAD(load),
        .PD_IN(pd_in),
        .SINIT(sinit),
        .SD_OUT(),
        .SD_IN(),
        .CE(),
        .DATA_VALID(),
        .LOAD_TAPS(),
        .TAPS_IN(),
        .AINIT(),
        .NEW_SEED(),
```

```
        .TERM_CNT());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of randgen is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of randgen is "black_box"

endmodule
```

Find Color Module Code

```
//////////////////////////////////////////
//
/*purpose: looks at display pixels to find 128 pixel averaged location
of user-controlled paddle of a specified color
*/
//////////////////////////////////////////////

module find_color(clk, reset,enter, hcount, vcount,vsync,
   vr_pixel,get_color,switch2,
   paddle_position_avg,test_pixel);

    input clk;    //same as vram_display
    input reset;
    input enter;
    input [10:0] hcount;
    input [9:0] vcount;
    input vsync;
    input [17:0] vr_pixel;
    input get_color;        //when switch[5] store pixel color
    input switch2;            //switch[2] enable look for color
    output [20:0] paddle_position_avg;
    output[17:0] test_pixel;

    parameter tolerance = 28;
    parameter avg_num = 128;
    parameter paddle_width = 20;

    reg [18:0]paddle_position_x =0;    /*need 8 extra bit places to add
                                         256 11-bit position values */
    reg [17:0] paddle_position_y=0;
    reg [20:0] paddle_position_avg=0;
    reg [8:0]counter = 0;
    reg [10:0] hcount_d1=0;
    reg [10:0] hcount_d2=0;
    reg [10:0] hcount_d3=0;
    reg [9:0] vcount_d1=0;
    reg [9:0] vcount_d2=0;
    reg [9:0] vcount_d3=0;
    reg look;                 //used to control looking for color
    wire [17:0]test_pixel;

    assign test_pixel = (
```

```
 ((hcount_d3-paddle_width)<(paddle_position_avg[20:10])) &&
 ((hcount_d3 +paddle_width) > paddle_position_avg[20:10]) &&
 ((vcount_d3-paddle_width)<(paddle_position_avg[9:0])
  ) &&
 ( (vcount_d3 +paddle_width) > paddle_position_avg[9:0]) )
   ? {12'b0,6'b111111}
   :
    (40<vr_pixel[17:12] &&
     0<=vr_pixel[11:6] && vr_pixel [11:6] < tolerance &&
     0<=vr_pixel[5:0]  && vr_pixel [5:0]  < tolerance )
   ? {6'b0,6'b111111,6'b0}
   : vr_pixel;

reg old_vsync;
wire new_frame;
assign new_frame = old_vsync & ~vsync;

reg old_enter;
wire press_enter;
assign press_enter = ~old_enter & enter;

always @(posedge clk) begin

   old_vsync <= vsync;
   old_enter <= enter;

   /*delay hcount and vcount by 3 clock cycles to allow for
     Processing by vram_display module- vr_pixel goes with
     hcount and vcount from 3 cycles ago*/

   hcount_d3 <= hcount_d2;
   hcount_d2 <= hcount_d1;
   hcount_d1 <= hcount;
   vcount_d3 <= vcount_d2;
   vcount_d2 <= vcount_d1;
   vcount_d1 <= vcount;

   /*check if pixel being displayed by vram_display is within
     tolerance of the color specified (red) */

   if(new_frame) begin
       look<=1;         //want to look for color every frame
       paddle_position_x <= 0;
       paddle_position_y <= 0;
       counter <= 0;
       end

   if( look==1 &&
      hcount_d3>50 && hcount_d3<720 &&
      vcount_d3>50 && vcount<548)

        if(
           40 < pixel[17:12] &&
           0 <= vr_pixel[11:6] && vr_pixel[11:6] < tolerance &&
           0 <= vr_pixel[5:0] && vr_pixel[5:0]  < tolerance
           )
```

```
                    /*want to average 128 position values of correct
                      color pixels- hcount_1 is current paddle x
                      position, vcount_1 is current paddle y position */

                    begin
                         paddle_position_x <= paddle_position_x +
                             hcount_d3;
                         paddle_position_y <= paddle_position_y +
                             vcount_d3;
                         counter <= counter+1;
                         /*stop looking for color after 128 position
                           values added */
                         if (counter>=avg_num-2) look<=0
                         end
                    else begin
                         paddle_position_avg[20:10] <=
                                paddle_position_x/avg_num;
                         paddle_position_avg[9:0] <=
                                paddle_position_y/avg_num;
                          end
    end    //end posedge clk
endmodule
```