ImprovTetris
6.111 Final Project Report

Scott Bezek & Ray Li

December 12, 2011

Table of Contents

1 Abstract

In the classic version of Tetris, the player must orient and piece together randomly-picked falling blocks in an attempt to complete and clear entire rows before everything piles up to the top. ImprovTetris aims to implement the game of Tetris on an FPGA, but with an improvisational twist: instead of just using the standard seven Tetris blocks in ImprovTetris, the falling block shapes are defined by the player's actual physical stance - much like a Microsoft Kinect game. As the block falls, the player can make up their own custom blocks on the fly and use two wireless buttons to move left and right in the game.

2 Overview

In order to provide this real-world control interface, ImprovTetris uses a camera to capture video of the player, which is then analyzed by an FPGA to generate a quantized Tetris piece based on the player's silhouette. An overview of the analysis process is shown in Figure 1, and is detailed in depth in Section 3.1.

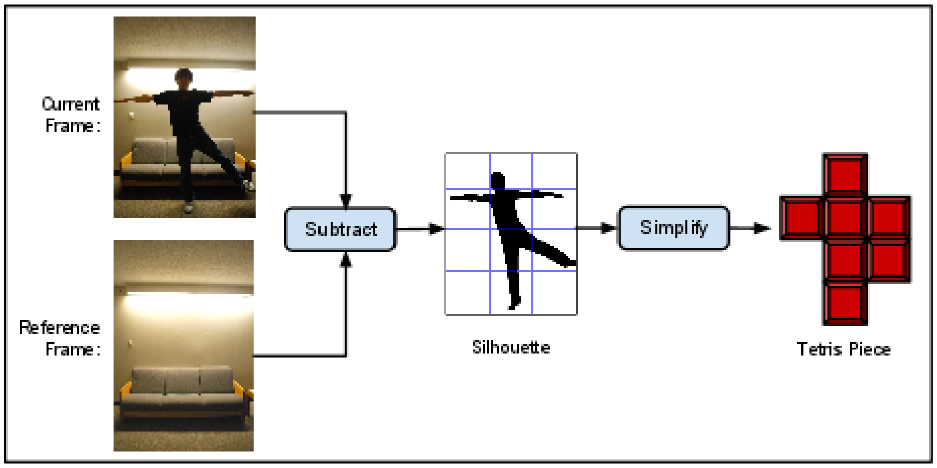


Figure 1 - ImprovTetris analyzes live video and simplifies the player's body shape to generate custom Tetris pieces.

While the block is falling, the player can contort his/her body to change the block's shape in real-time. This allows the player to create blocks that perfectly fit into the existing playing field, or even squeeze a block through a small crevice before expanding it to fill the open space. ImprovTetris provides on-screen feedback of the player's silhouette and the simplified block shape to help with forming useful pieces. Figure 2 shows a screenshot of the implemented display along with the originally proposed mock-up.

The rest of the gameplay is standard Tetris: the current block falls at a constant rate, and the user can slide it left or right by pressing buttons on a wireless controller. When an entire row is filled, it disappears and the remaining blocks shift downward. Each cleared row counts as a point towards the total score. The player can step out of the camera frame to pause the game and step back in to resume playing. If the pieces pile up to the top, the game ends and everything freezes. To restart a new game, the player simply needs to make a cross-shaped block (See Figure 12).

In order to accommodate different skill levels, ImprovTetris has 4 different falling speeds that can be increased and decreased with the up and down buttons on the FPGA. Additionally, the player can choose between 3 different starting boards with pre-filled block patterns:

1. Beginner - Blank board
2. Moderate - Jagged incomplete rows of blocks filled on the bottom
3. Expert - Almost completely-filled board with a gaping hole in the middle that can only be reached with a single block navigating through a narrow tunnel.

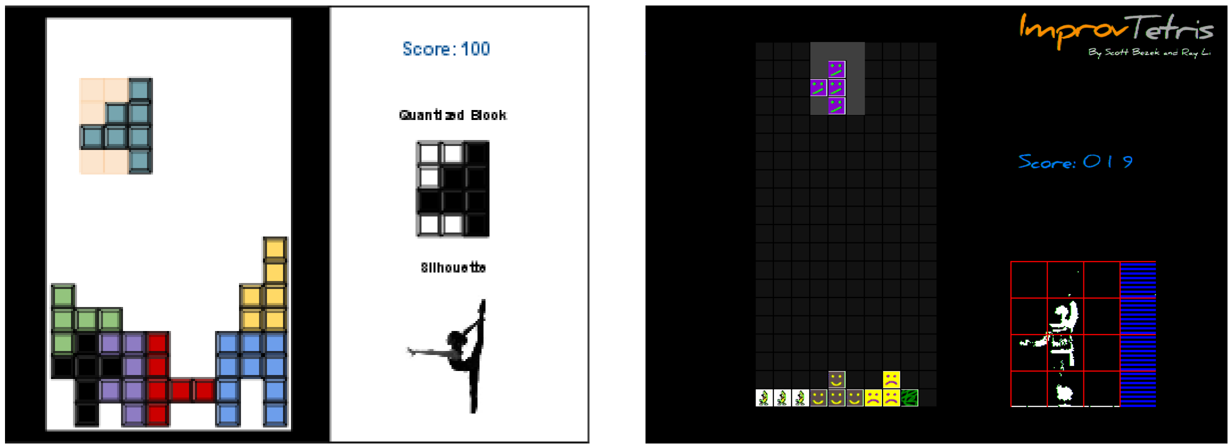


Figure 2 - Side-by-side comparison of the proposed GUI mock-up and the actual display as implemented.

3 Implementation

The implementation of ImprovTetris is broken down into three major subsystems: Image-Processing, the finite state machine logic (FSM), and the Display Module (Figure 3). The Image-Processing subsystem takes in image frames from the camera and outputs a 3x4 grid of the player's block shape to the FSM. The FSM controls the game logic and updates the playing field: it causes the current block to fall, detects when the block has settled, and clears/scores any complete rows. The Display Module takes the silhouette, the quantized Tetris piece, the playing field and score, and images from the ROM to piece together the display output. The following sections describe how these subsystems function and interact.

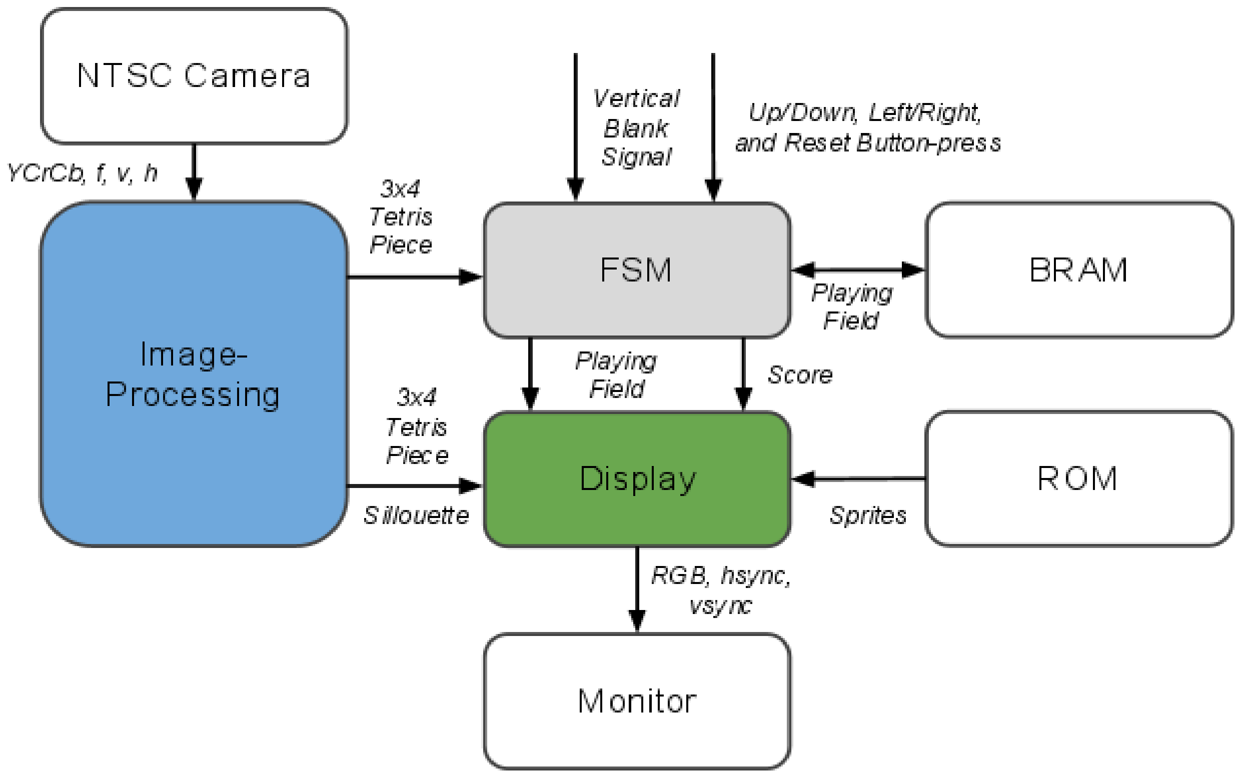


Figure 3 - Block Diagram of ImprovTetris, highlighting the three major subsystems.

3.1 Image-processing Subsystem (Scott)

The image-processing subsystem is responsible for capturing NTSC video, filtering, down-sampling, and quantization to determine what block shape the player's body forms.

The basic process is to capture and save a reference image at the beginning of the game when the player is not standing in front of the camera, and then compare each video frame during the game to this reference image (as shown previously in Figure 1). If any pixel differs enough between the reference frame and the current frame, then that pixel is considered "occupied." These occupied pixels form a silhouette of the player, which is then quantized into a 3x4 grid - this is the custom Tetris piece that falls in the game.

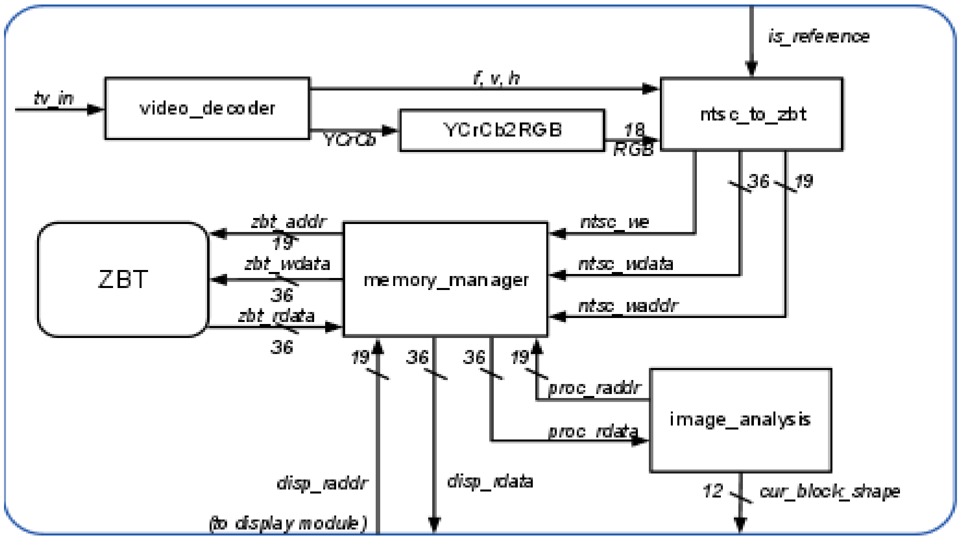


Figure 4 - Block diagram of modules within the Image-Processing subsystem

The Image-Processing subsystem (Figure 4) is composed of the following modules:

1. **video_decoder** - [Provided by 6.111 staff] Accepts signals from an NTSC camera and decodes that information into YCrCb, vertical and horizontal sync signals, and a field bit which denotes even vs. odd interlaced fields
2. **YCrCb2RGB** - [Provided by 6.111 staff] Converts the 30 bit YCrCb data into an 18 bit RGB value.
3. **ntsc_to_zbt** - Determines where to store each pixel of the NTSC video (as 18-bit RGB values) in ZBT memory
4. **image_analysis** - Reads the current video frame out of memory and compares it to the reference frame. Calculates the silhouette and outputs a 3x4 Tetris piece
5. **memory_manager** - Arbitrates access to the ZBT

3.1.1 The ntsc_to_zbt module

The ntsc_to_zbt module accepts the horizontal/vertical/field signals from the video_decoder along with the 18-bit RGB values for the current pixel and outputs the memory address and data to write to memory. A sample implementation of this module was provided, but it needed to be changed substantially to adapt to a different memory-addressing scheme and to correct bugs. The most notable issue we fixed was getting full vertical resolution; previously, incorrect logic prevented odd rows (of the interlaced video) to be saved to the ZBT, duplicating the even rows instead.

The core of this module calculates the appropriate address within ZBT to store each camera pixel. It also synchronizes between the two clock domains (27MHz for NTSC, 65MHz for VGA). Since the memory lines of the ZBT SRAM are 36 bits wide, we stored 2 pixels with 18 bits each of color information at each memory address. With 512x512 pixels of video data, this uses 131,072 addresses in the ZBT memory chip, which has a total of 512K addresses (~25% usage). However, since we store a reference frame along with the current frame, this usage is doubled, for a total of 262,144 addresses. In addition, we modified the memory addressing code to mirror the video frames horizontally - this way the silhouette on screen appears to move in the same direction as the player instead of the inverse. An example current/reference image can be seen in Figure 5.

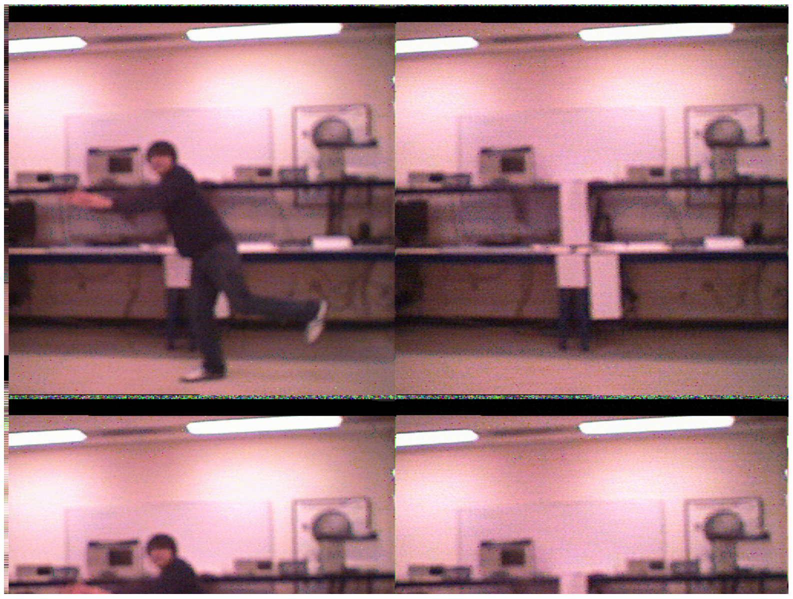


Figure 5 - "Preview" mode shows the 2 banks of ZBT memory, holding the current image (left), along with the reference image (right). Camera is purposefully out of focus to act as a low-pass filter.

Inputs:

1. f, v, h - The field, vertical, and horizontal sync for NTSC
2. RGB - The 18 bit RGB value for the current pixel
3. is_reference - Determines whether or not to save this as the reference image

Outputs:

1. ntsc_waddr - The address to write in ZBT
2. ntsc_wdata - The data to write in ZBT (combined 2 pixels of RGB data)
3. ntsc_we - Controls write-enable on the ZBT

Testing Strategy:
To test this module, we downloaded it to the FPGA and visually inspected the output. This way it was easy to spot bugs (like the vertical resolution issue) that would have been easy to overlook in a simulation.

3.1.2 The image_analysis module

This module reads the current frame and reference frame out of ZBT memory, does some simple filtering, and calculates the "silhouette," It uses the silhouette to calculate the custom 3x4 Tetris piece, which it outputs to the main FSM and display modules.

This image processing is done sequentially (i.e. pixel-by-pixel) during one frame of the VGA display. When the image_analysis module “visits” each pixel, it actually performs 2 reads from the ZBT memory - it needs to read both the current image and the reference image and then determine how much the pixel differs between the two images. The difference between the images is the summed difference of each of the red, green, and blue color components:

$$diff = |r1 - r2| + |g1 - g2| + |b1 - b2|$$

The absolute value is necessary for each color component so that a positive change in red can’t cancel out a negative change in green or blue.

It takes on the order of 262,144 cycles to process the images (there are 512*512 pixels). Rendering the VGA frame takes ~600,000 cycles which provides plenty of time for the processing to take place in parallel.

The pixels in the images are analyzed in small 4x4 pixel blocks, so that noisy fluctuations in a single pixel’s color are outweighed by the other 15 pixels within that block. This acts as a simple filter to reduce the inherent noisiness of cameras. The order in which pixels are visited is shown in Figure 6. By visiting the pixels in this order, it’s easy to keep a running sum of the pixel differences within a particular 4x4 block and reset it when you jump up to the next block (i.e. reset the difference accumulator during 16 → 17 transition in Figure 6).

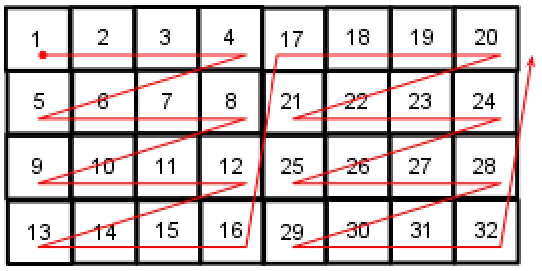


Figure 6 - Pixels are analyzed in chunks of 4x4, as indicated by the red line and numbering.

Since the ZBT reads have a two clock cycle delay, they are pipelined while traversing the image. The pipelined process looks like:

- Step 1: Request current (CUR) pixel 1
- Step 2: Request reference (REF) pixel 1
- Step 3: Request CUR pixel 2. CUR pixel 1 data is now available - save it to a register.
- Step 4: Request REF pixel 2. REF pixel 1 data is now available. Subtract each R, G, B component from the saved CUR pixel (in the register). Sum differences and add to accumulator
- ...
- Finished with 4x4 block

After each 4x4 pixel block, the accumulator holds the total pixel difference of those 16 pixels - this is compared to a user-adjustable threshold (see Figure 7), and if the value is high enough, a corresponding bit is set to 1 in the silhouette BRAM bitmap indicating that the block is “occupied.” In addition to setting a bit in the silhouette BRAM, one of 12 accumulators (one for each of the 12 possible cells for the user’s Tetris piece) is incremented. When the entire image has been analyzed, each of those 12 accumulators is compared to a second threshold, and if high enough the corresponding cell of the user's Tetris piece will become filled.

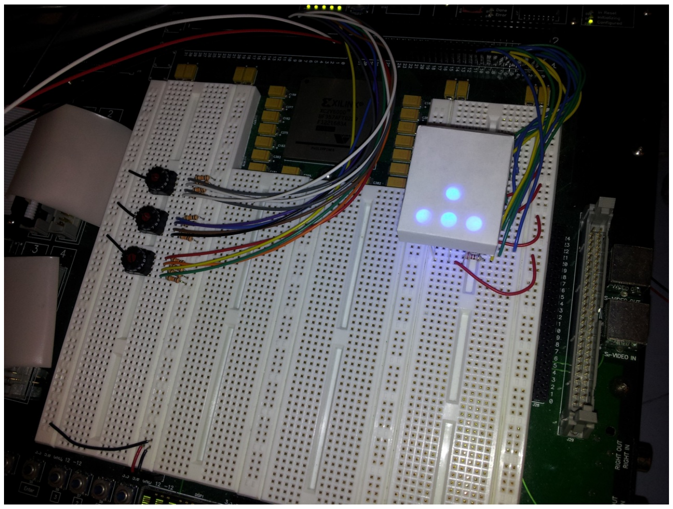


Figure 7 - Brightness thresholds are adjusted using three 4-bit hex switches (left). The LED matrix on the right shows the current user-created Tetris shape.

Inputs:
1. proc_rdata - The data being read from ZBT

Outputs:
1. proc_raddr - The address to read a pixel from the ZBT memory
2. cur_block_shape - The 12-bit representation of the calculated 3x4 Tetris piece

Testing Strategy:
This module was tested directly on the FPGA due to the complexity of the signals involved. To debug and verify correct operation, each of the image analysis steps was individually tied to a switch on the labkit (instead of running in real-time). This way we could see the intermediate output at each step of the way. We also used the second ZBT as a place to store the difference image of the current frame vs. the reference frame for debugging purposes.

3.1.3 The memory_manager module

One of the major challenges of the image-processing subsystem was storing and accessing the camera images in memory. ZBT memory cannot simultaneously read and write at different addresses, so we needed to carefully specify when image data is being stored vs. accessed. There was also a limit on the amount of data that the ZBT memory can hold, so we had to be careful to allocate it appropriately.

The memory_manager uses the "blank" signal from the VGA driver to control access to the ZBT. If *blank* is high, then the NTSC decoder is allowed to write to ZBT, otherwise only the display module is allowed to read the playing field from BRAM. Although the memory_manager is simple, it offers clean modularity between the components.

3.2 FSM (Ray)

The FSM handles the general game logic and keeps track of the scores. The Tetris game starts off in the *Falling* state when the player's custom block is falling down. When it lands, the game goes into the *Clearing* state to calculate all the complete rows. Then, the game goes through a flashing animation in the *Flash* state, and finally shift the blocks down in the *Collapse* state to collapse the complete rows. If nothing can be cleared and the most recent piece hits the top of the playing field, then the game ends in the *Game_Over* state. The player may reset and initiate a new game, which starts over in the *Falling* state. The state diagram is shown in Figure 8.

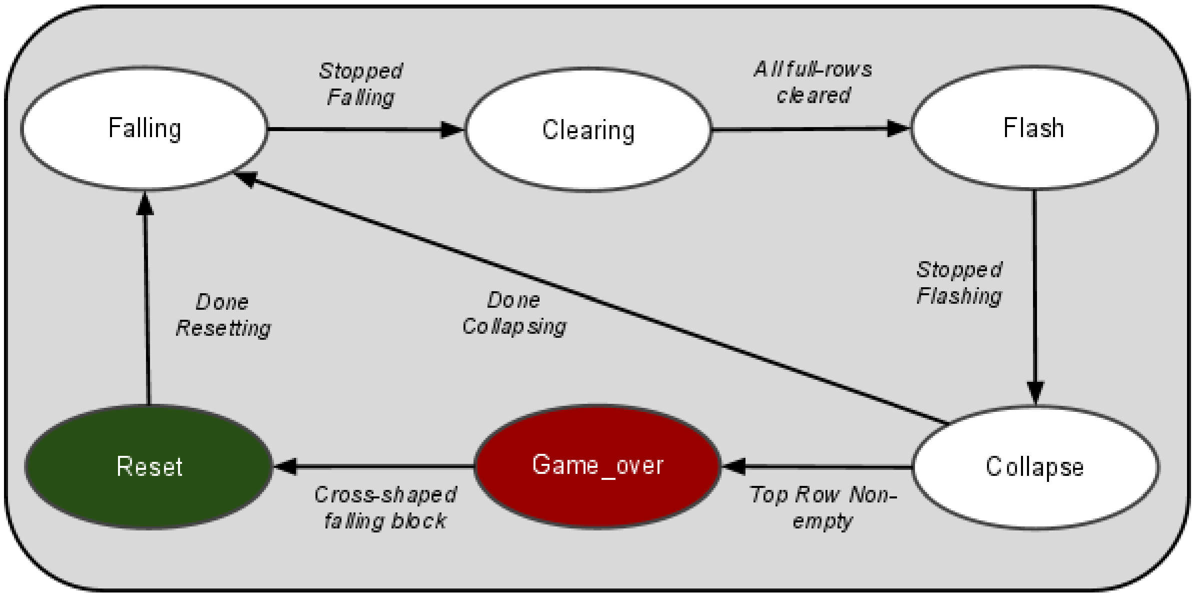


Figure 8 - Diagram of FSM state machine that governs the game logic.

Inputs:

1. **(FP):** Shape (3x4 grid) of player's quantized silhouette
2. **(left, right)**: Left/right button presses
3. **(up, down):** Up/down button presses
4. **(blank)**: Vertical Blank Signal
5. **(coord):** Coordinate of block color in playing field BRAM (4-bit x, 5-bit y)
6. **(reset):** Resets the FSM and restarts the game.

Outputs:

1. **(score):** Score (to display)
2. **(blockColor):** 3-bit pixel color of block selected by coord.
3. **(rowFilled):** Tell Display if current block selected is in a rowed that's completed. This is used to generate the flashing animation.

3.2.1 Timing

Since the Display module needs to read the playing field state from the BRAM to know what to display, the FSM module cannot have access to the BRAM and calculate next states at the same time. Thus, the FSM will only step through its states during the vertical blanking periods of the monitor and halt all reading and writing operation from/to the BRAM when Display is running (See Figure 9). The vertical blanking period lasts about 12,160 clock cycles - way more than enough to step through any state completely.

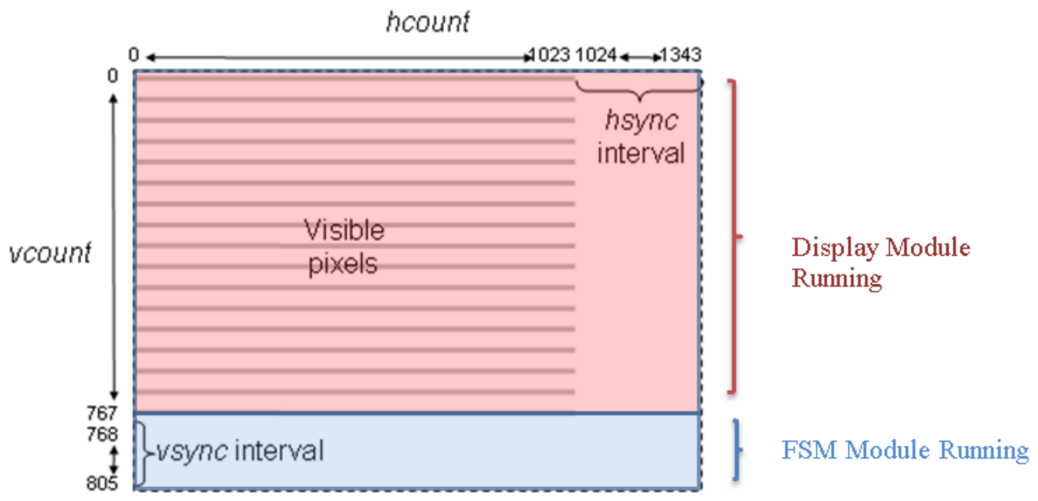
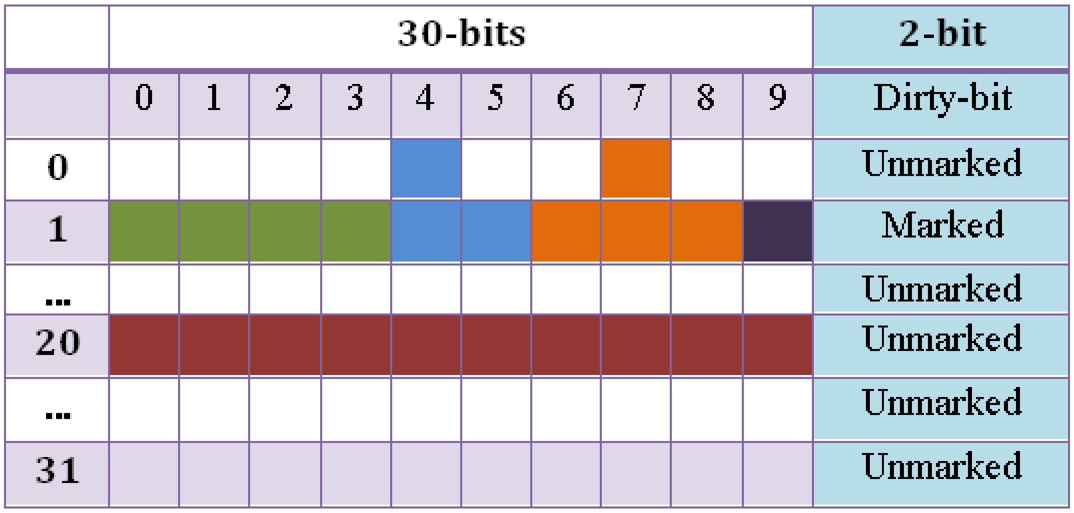


Figure 9 - Pink part of the timing gives Display Module access to BRAM memory while the Blue part gives FSM Module access to BRAM for calculations

3.2.2 Memory

<u>Playing Field</u>

The FSM uses double-buffered BRAMs with each 32-by-32 bits large (2048 bits in total) to represent the playing field of the tetris board (See Figure 10).



	30-bits										2-bit
	0	1	2	3	4	5	6	7	8	9	Dirty-bit
0					🟦			🟧			Unmarked
1	🟩	🟩	🟩	🟩	🟦	🟦	🟧	🟧	🟧	🟪	Marked
...											Unmarked
20	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	Unmarked
...											Unmarked
31											Unmarked

Figure 10 - Structure of each BRAM representing the Playing Field. The 21st row has the whole row permanently filled solid with its dirty-bit unmarked while in 2nd row, the dirty-bit has been marked for the FSM to flash the row and then collapse it.

Each 32-bit memory block represents a row in the playing field. 30-bits hold each of the 10 blocks' 3-bit color and the remaining 2-bit *dirty bits* are used for book-tracking when collapsing the rows and calculating the score.

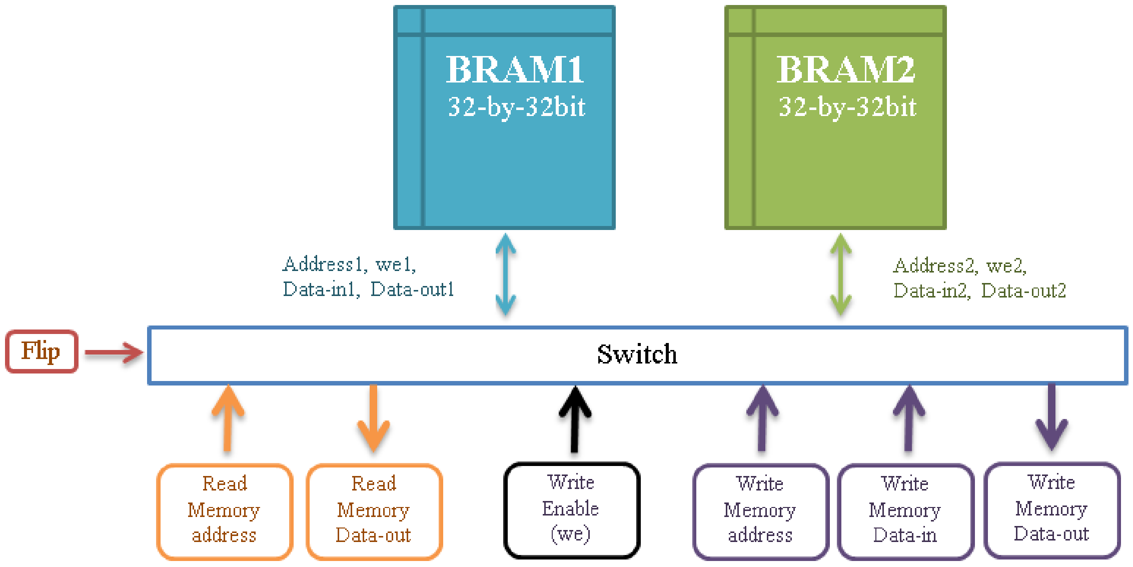


Figure 11 - Diagram of the double-buffered BRAMs. The switch module abstracts out the two flip-flopping BRAMs into a reading BRAM (Orange) and a Writing BRAM (Purple). The flip input flips which BRAM is being and which BRAM is being read.

We used double-buffered BRAMs to make reading and writing to memory instantaneous and hassle-free (See Figure 11). This makes the logic for calculating the playing board states much cleaner and easier since we don't have to buffer the memory read when writing it back into the same memory address (Required in collapsing rows as well as flashing animations). The playing field has the 21st row permanently filled with white blocks and its *dirty-bits* unmarked to ensure collision checking to stop all falling piece when they hit the bottom. Non of the states will ever clear this row.

Falling Piece

The FSM uses the 12-bit register passed-in from image_processing and a 3-bit register to represent the shape and color of the falling piece.

3.2.3 Falling State

The game spends most of its time in this state. Each falling block is represented in memory as a12-bit register passed in from the image_processing module. The block keeps track of its current upper-left x and y coordinates in the variables FP_x (0 to 9 going right) and FP_y (0 to 19 going down). A *speed* counter determines how fast the block falls and can vary from 3 to 15 with 3 being the fastest speed and 15 the slowest. The the up and down buttons on the FPGA decrements and increments this counter, respectively, to speed up and down the game. Every (*speed* * 140,000) cycles on a 65mHz clock (~ *speed* * 0.002 second), the state will calculate if:

- **(GAME_OVER):** It's game over by checking to see if the falling block hasn't dropped from its initial position (FP_x = 3, FP_y = 0) and has already collided with blocks in the playing field stored in the BRAM.
    - **(YES):** If game over, the state will freeze the playing field and transition to game_over state.
    - **(NO):** If the game can continue, the falling state will attempt to decrement the falling block's FP_y counter and check if there's a collision with blocks in the playing field.
        - **(COLLISION):** If there's a collision, the state will simply store the falling block shape into the BRAM playing field at coordinates FP_x and FP_y and begin treating it as a filled block in the playing field. Additionally, the state will transition to the Clearing state to begin clearing potentially completed rows.
        - **(NO COLLISION):** If there's no collision, the state increments the FP_y counter and the falling piece will automatically shift down on the next monitor frame refresh.

The remaining time when the Falling state is not calculating the next position of the falling piece, it's handling left/right button requests from the player to move the falling piece left or right. However before handling the left/right request, the state tries to move the piece left/right and checks for collision with the playing field in BRAM:

- **(COLLISION):** If there's a collision, then the state won't move falling block left or right.
- **(NO COLLISION):** If there's no collision, then the state will increment/decrement FP_x between 0 and 7 depending on the button pressed and the block image will update its

position on the next monitor refresh

3.2.4 Clearing State

This state takes roughly 20 clock cycles to step through. So to the player, this state is invisible. The Clearing state steps through each row of the playing field and checks to see if any row has been completed - all the colors of each block is not black (3’b000) and marks their *dirty bits* to be (2’b10) and transitions to the Flash state.

3.2.5 Flash State

This state simply waits for ~0.002 * *speed,* the amount of time it takes to drop the falling block one step, and transitions to Collapse state. During this time, display will flicker rows with marked dirty bits on and off between the colors black and white to produce a flashing effect for completed rows about to be cleared.

3.2.6 Collapse State

The Collapse state takes at most ~160 clock cycles to find all rows with marked dirty bits (Since you can only have cleared at most 4 rows with one 3-by-4 falling block) and shift all the rows above down. This is a simple for loop nested within another. The outer loop iterates through each row and looks for the row marked for clearing. Once found, it increments the score counter by 1 and enters the inner loop that iterates through each each row above the marked row and shifts each down by one row. The outer loop keeps on searching for marked rows and collapsing them with the inner loop until it can no longer find any more marked rows to clear. When done with collapsing, the state simply resets FP_x and FP_y to their initial conditions (0, 3), generates a new color for the falling block and transitions back to the Falling state to start dropping the new falling piece.

3.2.7 Game_over State

This State halts the game and waits until the falling piece (12-bit registers) become a cross shape (See Figure 12). Once the player makes a cross shape, the state will transition to Reset state to reset the board and load in the new falling piece.

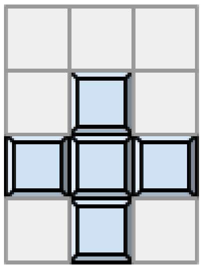


Figure 12 - Shape of falling block that resets and restarts the game.

3.2.8 Reset State

This state resets all internal variables (scores, FP_x, FP_y, etc.., clears every row and their *dirty-bits* in the playing field and writes a solid row on the 21st row with its *dirty-bits* unmarked (See Memory section for the reason). This state also alternately writes custom preset block patterns to the playing field depending on which level of difficulty the user has selected. Once done resetting, the state transitions to the Falling state and begins the game.

3.2.9 Testing Strategy

To test this module, we specified the shape of the falling block through the switches[7:0] and the left/right movement with button_left and button_right. This was enough to test all the states of the game to check for correctness.

3.3 Display Module (Ray and Scott)

The display pieces together the sprites from the ROM, the silhouette from the SRAM, the score and the game state from the FSM, and the calculated quantized block shape. These elements are combined on a pixel-by-pixel basis so that the display module can output the appropriate signals to the computer monitor. It also handles special effects, such as flashing blocks rapidly when a row is cleared.

Depending on the current (x,y) coordinate being drawn on screen, the display module will get the pixel data from a different source. For instance, if the coordinate lies within the playing field region, it will request the 3-bit color of the playing field cell (from the FSM), and then use that 3-bit "color" number along with the x and y coordinate as an address into the tile ROM to get the graphical pixel to draw. The display module requests the 3-bit color by outputting the current coordinate (x = 0 to 9, y = 0 to 19), and receives the 3-bit color via the *blockColor* input from the FSM. A listing of the ROMs we generated for this project is available in Appendix B.

Inputs:

1. **(blockColor)**: render the blocks in grid
2. **(ROM):**render fancy logo and custom block images for styling
3. **(score):** renders the score using cstring module written long-time ago.
4. **(sil_r, sil_g, sil_b)**: render silhouette of player

Outputs:

1. **(coord):** Selects blockColor of piece at x, y coordinate in the playing field
2. **(r, g, b):** Monitor

3.3.1 Testing strategy:

We simply inputted static scores, falling block shapes, playing field colors and checked for pixel correctness on the monitor.

4 Challenges

We initially used two 2-D register arrays to represent the state of the playing field as well as the position of the falling piece. This took up about ~1200 registers and made wire routing impossible. The code was left compiling over multiple days and still couldn't finish. There simply weren't enough wires in the FPGA coming into/out of each register to make the routing feasible. We then switched to double-buffered BRAMs to store the playing field states and reduced the routing problem significantly. The constraint ratio went from 98/100 down to

2/100 and the code finally compiled under 2 minutes. However, getting the double-buffered BRAMs to read and write properly took significant efforts because reading no longer took 1 clock cycle delays to setup correctly but rather 2 cycles and had to be model-simulated extensively to smooth out all the timing issues.

We decoupled our modules fairly well and only share about 12-bits of memory (falling block shape). All out components ran on the same 65mHz clock so we didn't run into any timing issues when piecing together all the components. The worst integration issue we ran into was that we hadn't agreed on the bit order of the 12-bit block shape, so the bits had to be reversed when connecting the modules in order to get the game to work correctly.

5 External Hardware

ImprovTetris requires minimal external hardware - only an NTSC video camera and 2 wireless push buttons are needed to capture the movements of the player and control the left/right movement of the falling block far away from the FPGA. All of the remaining image processing and game logic can be implemented directly on the FPGA labkit.

The wireless pushbuttons were configured using two "HappyBoard" microcontrollers loaned to us by 6.270. One HappyBoard monitored the digital input of two buttons (see Figure 13) and sent this information over RF to another HappyBoard base-station which had digital outputs connected to the USER inputs of the labkit.

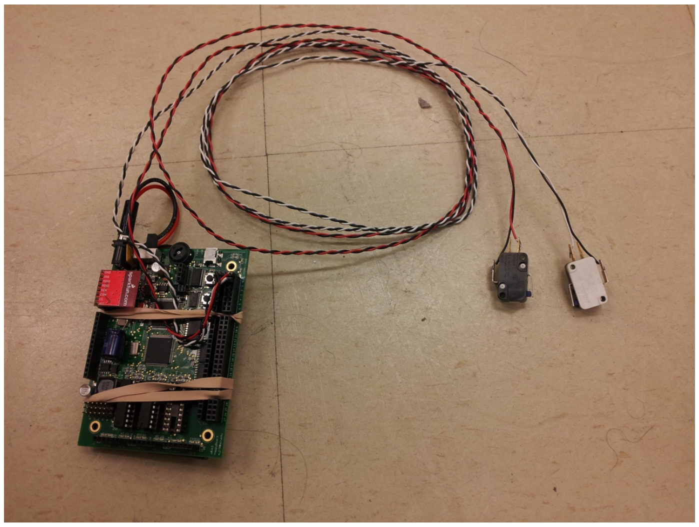


Figure 13: HappyBoard (left) with wireless transmitter, with two microswitches (right) connected to the digital inputs

6 Conclusion

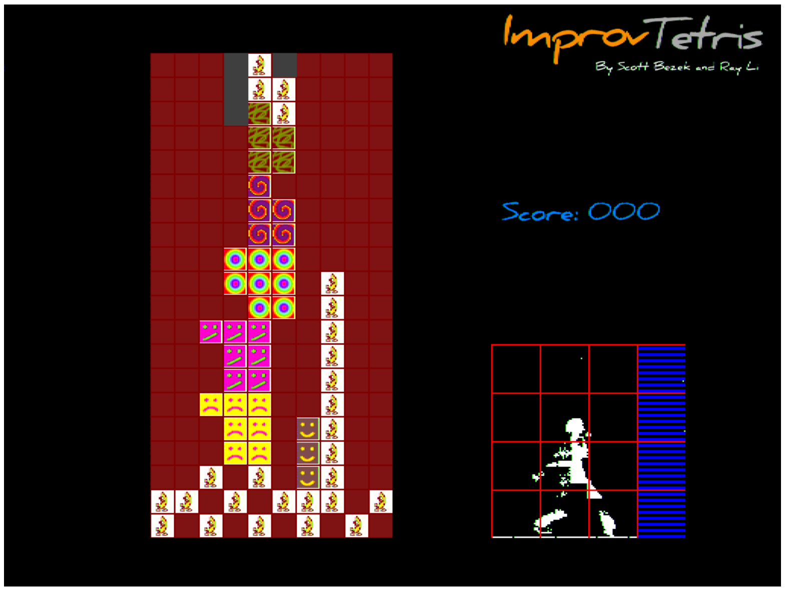
This project re-implemented the classic Tetris game to allow the user to generate custom falling block shapes on the fly and control left/right movements via wireless buttons much like a Wii/

Kinect game console. We used push buttons hooked up to HappyBoards to communicate via RF with the FPGA, along with an NTSC camera to capture the player's body shape. By combining these different pieces - image analysis and game logic - we were able to create a functional real-world interface to the classic game of Tetris.

7 Acknowledgements

ImprovTetris was inspired by Maximilian Wendt and Markus Wellmann's "Silhouetris" project.[1] We would like to thank all the TAs/LAs/Instructors: Professor Stojanovic, Gim, Kevin, and Jon for all the help and advice that made everything possible. Thank you.

[1] http://vimeo.com/25952122



Game Over

Appendix A: Code listings

image_analysis.v

```
// Image Processing Module


module image_analysis(
       input clk,
       input enable,
     input restart,
       output [18:0] zbt_read_addr,
       input [35:0] zbt_read_data,

       output reg [11:0] cur_block_shape,

     input [10:0] occupied_threshold,

     // silhouette-related io
     input [11:0] block_diff_threshold,
     output reg [13:0] sil_waddr,
     output reg sil_wdata,
     output reg sil_we,

     // debug io
     output [8:0] x_dbg, // TODO: removeme
     output [8:0] y_dbg,  // TODO: removeme
     output [11:0] diff_dbg,
     output step_dbg
       );

    reg [8:0] x = 0;
    reg [8:0] y = 0;


    // There are 2 steps that must occur for each pixel - reading
    // the current frame pixel, reading the reference frame pixel.
    // This also works out conveniently so when we are in READ_CUR,
    // the read_data is the current frame data from the previous READ_CUR
    parameter READ_CUR = 0;
    parameter READ_REF = 1;
    reg step = READ_CUR;

    assign step_dbg = step;

    // delayed position registers (these are the x,y we're currently looking at)
    // x_d[1] is x delayed 2 cycles
    reg [8:0] x_d [2:0];
    reg [8:0] y_d [2:0];

    assign x_dbg = x_d[1];
    assign y_dbg = y_d[1];

    reg [35:0] current_two_pixels;


    // Calculate the R,G,B pixel differences (absolute value) between the
    // "current" pixel in register and the reference pixel being
    // read from zbt
    wire [3:0] red_0_diff = (current_two_pixels[35:32] > zbt_read_data[35:32]) ?
                                                         (current_two_pixels[35:32]-
zbt_read_data[35:32]):
                                                         (zbt_read_data[35:32]-
current_two_pixels[35:32]);
    wire [3:0] green_0_diff = (current_two_pixels[29:26] > zbt_read_data[29:26]) ?
```

```
                                                              (current_two_pixels[29:26]-
zbt_read_data[29:26]):
                                                              (zbt_read_data[29:26]-
current_two_pixels[29:26]);
    wire [3:0] blue_0_diff = (current_two_pixels[23:20] > zbt_read_data[23:20]) ?
                                                              (current_two_pixels[23:20]-
zbt_read_data[23:20]):
                                                              (zbt_read_data[23:20]-
current_two_pixels[23:20]);
    wire [5:0] diff_0 = red_0_diff + green_0_diff + blue_0_diff;


    wire [3:0] red_1_diff = (current_two_pixels[17:14] > zbt_read_data[17:14]) ?
                                                              (current_two_pixels[17:14]-
zbt_read_data[17:14]):
                                                              (zbt_read_data[17:14]-
current_two_pixels[17:14]);
    wire [3:0] green_1_diff = (current_two_pixels[11:8] > zbt_read_data[11:8]) ?
                                                              (current_two_pixels[11:8]-
zbt_read_data[11:8]):
                                                              (zbt_read_data[11:8]-
current_two_pixels[11:8]);
    wire [3:0] blue_1_diff = (current_two_pixels[5:2] > zbt_read_data[5:2]) ?
                                                              (current_two_pixels[5:2]-
zbt_read_data[5:2]):
                                                              (zbt_read_data[5:2]-
current_two_pixels[5:2]);
    wire [5:0] diff_1 = red_1_diff + green_1_diff + blue_1_diff;

    // accumulator for total pixel difference in a 4x4 block
    // 4px*4px = 16px * [max diff per px = 63*3 = 189] = 3024 (12 bits)
    reg [8:0] block_diff_0 = 0;
    reg [8:0] block_diff_1 = 0;

    assign diff_dbg = block_diff_0;


    reg [10:0] occupied_count [15:0];


    reg running;
    reg was_running;

    always @ (posedge clk) begin
                was_running <= running;

                // keep running if we're not at the last pixel, or if restart is 1
                running <= (running && !(x_d[1] == 510 && y_d[1] == 511)) | restart;


                // Update X and Y to move through entire image in a specific
                // pattern which makes 4x4 block analysis easy.
                //
                // This is the order that pixels are visited:
                //  0  1  2  3  16 17 18 19
                //  4  5  6  7  20 21 22 23 . . .
                //  8  9  10 11 24 25 26 27
                //  12 13 14 15 28 29 30 31
                //            .
                //            .
                //            .

        if (running) begin
                // always iterate steps
                step <= (step == READ_REF || ~enable) ? 0 : step + 1;

                // update x position
                if (step == READ_REF && enable) begin
                        // done with READ_REF step, and enabled, so it's time to move on
                        if (x[1:0] == 0 || y[1:0] == 3) begin
                                // if we're on the left edge of a 4x4px block (x[1:0]==0)
                                // or if we're on the bottom row of the 4x4 block (y[1:0]==3)
                                // then we should increment the x pos by 2
                                x <= x+2;
                        end else begin
                                // otherwise we actually step back 2
                                x <= x-2;
                        end
                end else begin
                        // not enabled, or not done with READ_REF step, so stay here
                        x <= x;
                end

                // update y position
                if (x[1:0] == 2 && step == READ_REF && enable) begin
                        // we will only change the y position if we've finished a row
                        // of the 4x4 block (x[1:0] == 2) and if we're done with the
                        // READ_REF step, and enabled

                        if (y[1:0] != 3 || x == 510) begin
                                // if we're not on the bottom row of the 4x4 block (y[1:0] != 3)
                                // or if we're at the last x of the image, we should move down
                                // one row
                                y <= y + 1;
                        end else begin
                                // otherwise we're on the bottom row, so we should move up 3
                                // rows to get to the top row of the next block
                                y <= y - 3;
                        end
                end else begin
                        y <= y;
                end

                // update the position delay registers
                x_d[2] <= x_d[1];
                x_d[1] <= x_d[0];
                x_d[0] <= x;

                y_d[2] <= y_d[1];
                y_d[1] <= y_d[0];
                y_d[0] <= y;

                if (step == READ_CUR) begin
                        current_two_pixels <= zbt_read_data;
                end else if (step == READ_REF && enable) begin

                        if (x_d[1][1:0] == 0 && y_d[1][1:0] == 0) begin
                                // we just finished a block, so reset the block diff accumulator
                                block_diff_0 <= 0;
                                block_diff_1 <= 0;

                                sil_wdata <= block_diff_0 + block_diff_1 > block_diff_threshold;

                                if (block_diff_0 + block_diff_1 > block_diff_threshold &&
                                                y_d[2][8:2] < 127) begin // slight hack here to
ignore the bottom row of the silhouette
                                // figure out which accumulator to use based on the
quadrant of the
                                // image we were in (use upper 2 bits of delayed x and y)
                                occupied_count[{y_d[2][8:7], x_d[2][8:7]}] <=
occupied_count[{y_d[2][8:7], x_d[2][8:7]}] + 1;
                                end
                        end else begin
                                // accumulate the total pixel difference
                                block_diff_0 <= diff_0;
                                block_diff_1 <= diff_1;
                                sil_waddr <= {y_d[1][8:2], x_d[1][8:2]};
                        end
                end

                sil_we <= enable && (x_d[1][1:0] == 0 && y_d[1][1:0] == 0);
         end else begin
                // not running
                sil_we <= 0;
                step <= 0;
                x <= 0;
                y <= 0;
                x_d[0] <= 0;
                x_d[1] <= 0;
                y_d[0] <= 0;
                y_d[1] <= 0;

                if (was_running) begin
                        // compare the accumulators to the threshold, and put that
                        // value into the 12-bit block shape registers
                        cur_block_shape[ 0] <= occupied_count[ 0] > occupied_threshold;
                        cur_block_shape[ 1] <= occupied_count[ 1] > occupied_threshold;
                        cur_block_shape[ 2] <= occupied_count[ 2] > occupied_threshold;
                        cur_block_shape[ 3] <= occupied_count[ 4] > occupied_threshold;
                        cur_block_shape[ 4] <= occupied_count[ 5] > occupied_threshold;
                        cur_block_shape[ 5] <= occupied_count[ 6] > occupied_threshold;
                        cur_block_shape[ 6] <= occupied_count[ 8] > occupied_threshold;
                        cur_block_shape[ 7] <= occupied_count[ 9] > occupied_threshold;
                        cur_block_shape[ 8] <= occupied_count[10] > occupied_threshold;
                        cur_block_shape[ 9] <= occupied_count[12] > occupied_threshold;
                        cur_block_shape[10] <= occupied_count[13] > occupied_threshold;
                        cur_block_shape[11] <= occupied_count[14] > occupied_threshold;

                        // reset the accumulators
                        occupied_count[ 0] <= 0;
                        occupied_count[ 1] <= 0;
                        occupied_count[ 2] <= 0;
                        occupied_count[ 3] <= 0;
                        occupied_count[ 4] <= 0;
                        occupied_count[ 5] <= 0;
                        occupied_count[ 6] <= 0;
                        occupied_count[ 7] <= 0;
                        occupied_count[ 8] <= 0;
                        occupied_count[ 9] <= 0;
                        occupied_count[10] <= 0;
                        occupied_count[11] <= 0;
                        occupied_count[12] <= 0;
                        occupied_count[13] <= 0;
                        occupied_count[14] <= 0;
                        occupied_count[15] <= 0;
                end
         end
     end

     assign zbt_read_addr = {step, 1'b0, y[8:0], x[8:1]}; //MSB is reference flag = step

endmodule
```

display.v

```
module display (
   input wire vclock,    // 65MHz clock
   input wire reset,           // 1 to initialize module
   input wire [10:0] hcount,    // horizontal index of current pixel (0..1023)
   input wire [9:0]     vcount, // vertical index of current pixel (0..767)
   input wire hsync,           // XVGA horizontal sync signal (active low)
   input wire vsync,           // XVGA vertical sync signal (active low)
   input wire blank,           // XVGA blanking (1 means output black pixel)
    input wire [9:0] score,
   input wire [2:0] boardColor,
    input wire sil_r,
    input wire sil_g,
    input wire sil_b,
    input wire preview,
    input wire [7:0] preview_r,
    input wire [7:0] preview_g,
    input wire [7:0] preview_b,
   output reg [7:0] r,
   output reg [7:0] g,
   output reg [7:0] b,
    output wire [8:0] coord,
    input wire isFallingPiece,
    input wire rowFilled,
    input wire gameOver
   );

    reg isFallingPiece_d;
    reg [2:0] boardColor_d;
    reg rowFilled_d;

    /*
     *   Tile ROM setup
     */
    wire [12:0] tile_addr;
    wire [23:0] tile_data;
    tile_rom tiles (.clka(vclock), .addra(tile_addr), .douta(tile_data));

    wire [3:0] x;
    wire [4:0] y;

    wire [10:0] hcount_f = hcount + 11'd1;

    assign x = hcount_f[8:5] - 6;
    assign y = vcount[9:5] - 2;
    assign coord = {x,y};

    assign tile_addr = {boardColor[2:0], vcount[4:0], hcount_f[4:0]};



    /*
     *  Logo ROM setup
     */
    wire [15:0] logo_addr;
    wire [23:0] logo_data;
    logo_rom logo (.clka(vclock), .addra(logo_addr), .douta(logo_data));

    parameter LOGO_W = 512;
    parameter LOGO_H = 96;
    parameter LOGO_START_X = 1024 - LOGO_W;
    wire [8:0] logo_x = hcount[8:0];
    wire [6:0] logo_y = vcount[6:0];
    assign logo_addr = {logo_y, logo_x};
    wire [3:0] score_ones;
    wire [3:0] score_tens;
    wire [1:0] score_hundreds;
    binary2bcd bcd(.A(score),.ONES(score_ones),
         .TENS(score_tens), .HUNDREDS(score_hundreds));

    reg [3:0] disp_digit_val;
    wire [13:0] num_addr;
    wire [7:0] num_data;
    font_num_rom nums (.clka(vclock), .addra(num_addr), .douta(num_data));

    parameter SCORE_START_X = 768;
    parameter SCORE_END_X = SCORE_START_X + 3*32;
    parameter SCORE_START_Y = 256;
    parameter SCORE_END_Y = SCORE_START_Y + 32;

    always @(*) begin
         case (hcount[6:5])
                2'd0: disp_digit_val = score_hundreds;
                2'd1: disp_digit_val = score_tens;
                2'd2: disp_digit_val = score_ones;
                2'd3: disp_digit_val = 0;
         endcase
    end

    assign num_addr = {disp_digit_val, vcount[4:0], hcount_f[4:0]};




    wire [11:0] score_text_addr;
    wire [7:0] score_text_data;
    score_rom score_text (.clka(vclock), .addra(score_text_addr), .douta(score_text_data));
    parameter TEXT_START_X = 640;
    parameter TEXT_END_X = TEXT_START_X + 128;
    parameter TEXT_START_Y = SCORE_START_Y;
    parameter TEXT_END_Y = SCORE_END_Y;

    assign score_text_addr = {vcount[4:0], hcount[6:0]};

    reg flip = 0;
    reg [31:0] counter;

    always @ (posedge vclock) begin
         if(counter == 139999) begin
                counter <= 0;
                flip <= !flip;
         end else begin
                counter <= counter + 1;
         end

         rowFilled_d <= rowFilled;
         isFallingPiece_d <= isFallingPiece;
         boardColor_d <= boardColor;

         if (preview) begin
                r <= preview_r;
                g <= preview_g;
                b <= preview_b;
         end else begin
                if(hcount[8:5] >= 6 && hcount[10:5] < 16 && vcount[9:5] >= 2 && vcount[9:5] < 22)
begin
                       // If we're within the playing field section:
                       if(rowFilled) begin
                              r <= (flip ? 8'b00000000 : 8'b11111111);
                              g <= (flip ? 8'b00000000 : 8'b11111111);
                              b <= (flip ? 8'b00000000 : 8'b11111111);
                       end else begin
                              if (isFallingPiece_d && boardColor_d == 3'd0) begin
                                     // This is part of the falling piece - highlight it by
```

```
                                        // setting the lower bits to 1
                                        r <= {tile_data[23:22], 6'b111111};
                                        g <= {tile_data[15:14], 6'b111111};
                                        b <= {tile_data[7:6], 6'b111111};
                                end else begin
                                        // Display the tile normally (or with a red hue if game
                                        // is over
                                        r <= (gameOver ? (tile_data[23:16] | 7'b1111111) :
tile_data[23:16]);
                                        g <= tile_data[15:8];
                                        b <= tile_data[7:0];
                                end
                        end
                end else if (hcount >= LOGO_START_X && vcount < LOGO_H) begin
                        // we're in the logo region of the screen (upper right corner)
                        r <= logo_data[23:16];
                        g <= logo_data[15:8];
                        b <= logo_data[7:0];
                end else if (hcount >= SCORE_START_X && hcount < SCORE_END_X &&
                                        vcount >= SCORE_START_Y && vcount < SCORE_END_Y)
begin
                        // we're in the score region of the screen
                        r <= 0;
                        g <= {1'b0, num_data[6:0]};
                        b <= num_data;
                end else if (hcount >= TEXT_START_X && hcount < TEXT_END_X &&
                                        vcount >= TEXT_START_Y && vcount < TEXT_END_Y)
begin
                        // display the text "Score:"
                        r <= 0;
                        g <= {1'b0, score_text_data[6:0]};
                        b <= score_text_data;
                end else begin
                        // fill the rest of the area with whatever the silhouette gives us
                        r <= {8{sil_r}};
                        g <= {8{sil_g}};
                        b <= {8{sil_b}};
                end
        end
    end

endmodule



/*
From:
http://www.ee.duke.edu/~dwyer/courses/ece52/Binary_to_BCD_Converter.pdf
*/
module binary2bcd(A,ONES,TENS,HUNDREDS);
input [7:0] A;
output [3:0] ONES, TENS;
output [1:0] HUNDREDS;
wire [3:0] c1,c2,c3,c4,c5,c6,c7;
wire [3:0] d1,d2,d3,d4,d5,d6,d7;
assign d1 = {1'b0,A[7:5]};
assign d2 = {c1[2:0],A[4]};
assign d3 = {c2[2:0],A[3]};
assign d4 = {c3[2:0],A[2]};
assign d5 = {c4[2:0],A[1]};
assign d6 = {1'b0,c1[3],c2[3],c3[3]};
assign d7 = {c6[2:0],c4[3]};
add3 m1(d1,c1);
add3 m2(d2,c2);
add3 m3(d3,c3);
add3 m4(d4,c4);
add3 m5(d5,c5);
add3 m6(d6,c6);
add3 m7(d7,c7);
assign ONES = {c5[2:0],A[0]};
assign TENS = {c7[2:0],c5[3]};
assign HUNDREDS = {c6[3],c7[3]};
endmodule

module add3(in,out);
input [3:0] in;
output [3:0] out;
reg [3:0] out;
always @ (in)
case (in)
4'b0000: out<=4'b0000;
4'b0001: out<=4'b0001;
4'b0010: out<=4'b0010;
4'b0011: out<=4'b0011;
4'b0100: out<=4'b0100;
4'b0101: out<=4'b1000;
4'b0110: out<=4'b1001;
4'b0111: out<=4'b1010;
4'b1000: out<=4'b1011;
4'b1001: out<=4'b1100;
default: out<=4'b0000;
endcase
endmodule
```

memory_manager.v

```
module memory_manager(
       input clk,
     input blank,
       input switch,
       input preview,

       input [18:0] ntsc_waddr,
       input [35:0] ntsc_wdata,
     input ntsc_we,

       input [18:0] disp_raddr,
       output [35:0] disp_rdata,

       input [18:0] proc_raddr,
       output [35:0] proc_rdata,

       output zbt_0_we,
       output [18:0] zbt_0_addr,
       output [35:0] zbt_0_wdata,
       input [35:0] zbt_0_rdata,

       output zbt_1_we,
       output [18:0] zbt_1_addr,
       output [35:0] zbt_1_wdata,
       input [35:0] zbt_1_rdata
       );
    assign zbt_0_we = blank && ntsc_we;
    assign zbt_0_addr = blank ? ntsc_waddr : (preview ? disp_raddr : proc_raddr);
    assign zbt_0_wdata = blank ? ntsc_wdata : 0;

    assign disp_rdata = zbt_0_rdata;
    assign proc_rdata = zbt_0_rdata;

    assign zbt_1_we = 0;
    assign zbt_1_addr = 0;
    assign zbt_1_wdata = 0;
endmodule
```

ntsc2zbt.v

```
//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.
//
// Bug fix: Jonathan P. Mailoa <jpmailoa@mit.edu>
// Date   : 11-May-09  // gph mod 11/3/2011
//
// Significantly modified by sbezek
//
///////////////////////////////////////////////////////////////////////////////
// Prepare data and address values to fill ZBT memory with NTSC data

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

   input      clk;    // system clock
   input      vclk;    // video clock from camera
   input [2:0]      fvh;
   input      dv;
   input [17:0]      din;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output reg     ntsc_we;    // write enable for NTSC data
   input      ref_frame;       // switch which determines mode (for debugging)
    output new_frame;

   parameter      COL_START = 0;//10'd30;
   parameter      ROW_START = 0;//10'd30;

   // here put the luminance data from the ntsc decoder into the ram
   // this is for 800 * 600 XGA display

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

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

    wire valid_edge = dv & ~old_dv;

    reg old_h;

    assign new_frame = col == 1 && row == 1 && frame == 1;

   always @ (posedge vclk) begin //LLC1 is reference
         old_dv <= dv;
         old_h <= fvh[0];

         vwe <= dv && col >= 32 && col < 544 && row < 272; // if data valid, write it
         old_frame <= frame;
         even_odd <= valid_edge ? frame : even_odd;// frame_edge ? ~even_odd : even_odd;

        col <= fvh[0] ? COL_START : (valid_edge && col < 1000 ? col+1 : col);
        row <= fvh[1] ? ROW_START : (fvh[0] & ~old_h ? row + 1 : row);
        vdata <= valid_edge ? din : vdata;

   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];

   always @(posedge clk)
        begin
     {x[1],x[0]} <= {x[0]-32,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];


   // compute address to store data in

    wire [7:0] x_addr = 8'd255 - x[1][8:1];           // Subtract from 255 to mirror left-to-right
    wire [8:0] y_addr = {y[1][7:0], eo[1]};

   wire [18:0] myaddr = {ref_frame, 1'd0, y_addr, x_addr};

   // Now address (0,0,0) contains pixel data(0,0) etc.

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


   always @(posedge clk) begin
         if (we_edge) begin
                ntsc_addr <= myaddr;
                // shift each of 2 pixels into a large register for the ZBT
                // (new pixel is high-order instead of low-order because we're mirroring the
image)
                ntsc_data <= { data[1], ntsc_data[35:18]};
         end

         if ( (we_edge) && (x[1][0]==1)) begin
                ntsc_we <= 1;
         end else begin
                ntsc_we <= 0;
         end
   end

endmodule
```

sil_display.v

```
module sil_display(
       input wire clk,
       input wire [10:0] hcount,
       input wire [9:0] vcount,
       output wire [13:0] sil_read_addr,
       input wire sil_read_data,
       output reg r,
     output reg g,
     output reg b
       );

    parameter SIZE = 256;

    parameter X_OFF = 640;
    parameter Y_OFF = 768 - SIZE - 64;

    wire [7:0] raw_x;
    wire [7:0] raw_y;

    assign raw_x = hcount - X_OFF;
    assign raw_y = vcount - Y_OFF;

    wire [6:0] x_addr;
    wire [6:0] y_addr;

    assign x_addr = raw_x[7:1];
    assign y_addr = raw_y[7:1];

    assign sil_read_addr = {y_addr,x_addr};

    always @ (posedge clk) begin
         if (X_OFF <= hcount && hcount < X_OFF+SIZE &&
               Y_OFF <= vcount && vcount < Y_OFF+SIZE) begin
                if (x_addr[4:0] == 0 || y_addr[4:0] == 0) begin
                       r <= 1;
                       g <= 0;
                       b <= 0;
                end else if (x_addr > 95 && y_addr[1]) begin
                       r <= 0;
                       g <= 0;
                       b <= 1;
                end else begin

                       r <= sil_read_data;
                       g <= sil_read_data;
                       b <= sil_read_data;
                end
         end else begin
                r <= 0;
                g <= 0;
                b <= 0;
         end
    end

endmodule
```

zbt_6111_sample.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2009-May-11: Fixed memory management bug by 8 clock cycle forecast.
//              Changed resolution to 1024 * 786 was ... 800 * 600.
//              Reduced clock speed to 40MHz.
//              Disconnected zbt_6111's ram_clk signal.
//              Added ramclock to control RAM.
//              Added notes about ram1 default values.
//              Commented out clock_feedback_out assignment.
//              Removed delayN modules because ZBT's latency has no more effect.
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
```

```
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

module zbt_6111_sample(beep, audio_reset_b,
		ac97_sdata_out, ac97_sdata_in, ac97_synch,
	ac97_bit_clock,

	vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
	vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
	vga_out_vsync,

	tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
	tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
	tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

	tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
	tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
	tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
	tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

	ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
	ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

	ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
	ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

	clock_feedback_out, clock_feedback_in,

	flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
	flash_reset_b, flash_sts, flash_byte_b,

	rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

	mouse_clock, mouse_data, keyboard_clock, keyboard_data,

	clock_27mhz, clock1, clock2,

	disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
	disp_reset_b, disp_data_in,

	button0, button1, button2, button3, button_enter, button_right,
	button_left, button_down, button_up,

	switch,

	led,

	user1, user2, user3, user4,

	daughtercard,

	systemace_data, systemace_address, systemace_ce_b,
	systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

	analyzer1_data, analyzer1_clock,
		analyzer2_data, analyzer2_clock,
		analyzer3_data, analyzer3_clock,
		analyzer4_data, analyzer4_clock);

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
      vga_out_hsync, vga_out_vsync;

   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
      tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
      tv_out_subcar_reset;

   input  [19:0] tv_in_ycrcb;
   input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
      tv_in_hff, tv_in_aff;
   output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
      tv_in_reset_b, tv_in_clock;
   inout  tv_in_i2c_data;

   inout  [35:0] ram0_data;
   output [18:0] ram0_address;
   output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
   output [3:0] ram0_bwe_b;

   inout  [35:0] ram1_data;
   output [18:0] ram1_address;
   output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
   output [3:0] ram1_bwe_b;

   input  clock_feedback_in;
   output clock_feedback_out;

   inout  [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
   input  flash_sts;

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
   output  disp_data_out;

   input  button0, button1, button2, button3, button_enter, button_right,
      button_left, button_down, button_up;
   input  [7:0] switch;
   output [7:0] led;

   inout [31:0] user1, user2, user3, user4;

   inout [43:0] daughtercard;

   inout  [15:0] systemace_data;
   output [6:0]  systemace_address;
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
         analyzer4_data;
   output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

   ////////////////////////////////////////////////////////////////////////////
   //
   // I/O Assignments
   //
   ////////////////////////////////////////////////////////////////////////////

   // Audio Input and Output
   assign beep= 1'b0;
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;
/*
*/
   // ac97_sdata_in is an input
   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

   // Video Input
   //assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b1;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b1;
   //assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = clock_27mhz;//1'b0;
   //assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs

/* change lines below to enable ZBT RAM bank0 */

/*
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;    // clock enable
*/

/* enable RAM pins */

   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_bwe_b = 4'h0;

/**********/
/*
   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_clk = 1'b0;
   assign ram1_we_b = 1'b1;
   assign ram1_cen_b = 1'b1;
  */
   //These values has to be set to 0 like ram0 if ram1 is used.
   assign ram1_ce_b = 1'b0;
   assign ram1_oe_b = 1'b0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_bwe_b = 4'h0;

   //clock_feedback_out will be assigned by ramclock
   //assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

   // Flash ROM
   assign flash_data = 16'hZ;
   assign flash_address = 24'h0;
   assign flash_ce_b = 1'b1;
   assign flash_oe_b = 1'b1;
   assign flash_we_b = 1'b1;
   assign flash_reset_b = 1'b0;
   assign flash_byte_b = 1'b1;
   // flash_sts is an input

   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

   // LED Displays
/*
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
*/
   // disp_data_in is an input

   // Buttons, Switches, and Individual LEDs
   //lab3 assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
   assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   assign user3 = 32'hZ;
   assign user4[31:12] = 20'hZ;

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

   // Logic Analyzer
   assign analyzer1_clock = 1'b1;
   assign analyzer2_clock = 1'b1;
   assign analyzer4_clock = 1'b1;

   ////////////////////////////////////////////////////////////////////////////
   // Demonstration of ZBT RAM as video memory

   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));



    wire [11:0] hex_in = user1[11:0];


    wire clk;
    wire locked;

   ramclock rc(.ref_clock(clock_65mhz), .fpga_clock(clk),
                            .ram0_clock(ram0_clk),
                            .ram1_clock(ram1_clk),   //uncomment if ram1 is used
                            .clock_feedback_in(clock_feedback_in),
                            .clock_feedback_out(clock_feedback_out), .locked(locked));
   // power-on reset generation
   wire power_on_reset;       // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(power_on_reset, clk, ~button_enter, user_reset);
   assign reset = user_reset | power_on_reset;

   // display module for debugging

   wire [63:0] dispdata;
    assign dispdata = {54'hFFFFFFFFFFFFFF, hex_in};
   display_16hex hexdisp1(reset, clock_27mhz, dispdata,
                  disp_blank, disp_clock, disp_rs, disp_ce_b,
                  disp_reset_b, disp_data_out);

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank,next_vblank;
   xvga xvga1(clk,hcount,vcount,hsync,vsync,blank,next_vblank);

   // wire up to ZBT ram
   wire [35:0] zbt_0_wdata;
   wire [35:0] zbt_0_rdata;
   wire [18:0] zbt_0_addr;
   wire        zbt_0_we;
    wire useless_clk_1;

   zbt_6111 zbt0(clk, 1'b1, zbt_0_we, zbt_0_addr,
               zbt_0_wdata, zbt_0_rdata,
               useless_clk_1,//ram0_clk,                    //to get good timing, don't connect
ram_clk to zbt_6111
               ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

   // wire up to ZBT ram
   wire [35:0] zbt_1_wdata;
   wire [35:0] zbt_1_rdata;
   wire [18:0] zbt_1_addr;
   wire        zbt_1_we;
    wire useless_clk_2;

   zbt_6111 zbt1(clk, 1'b1, zbt_1_we, zbt_1_addr,
               zbt_1_wdata, zbt_1_rdata,
               useless_clk_2,//ram0_clk,                    //to get good timing, don't connect
ram_clk to zbt_6111
               ram1_we_b, ram1_address, ram1_data, ram1_cen_b);


   // generate pixel value from reading ZBT memory
   wire [7:0]     vr_pixel_r;
   wire [7:0]     vr_pixel_g;
   wire [7:0]     vr_pixel_b;
   wire [18:0]     disp_raddr;
    wire [35:0]   disp_rdata;

   vram_display vd1(0,clk,hcount,vcount,vr_pixel_r,vr_pixel_g,vr_pixel_b,
               disp_raddr,disp_rdata);



    // Silhouette BRAM
    wire sil_write_data;
    wire [13:0] sil_write_addr;
    wire sil_write_enable;
    wire [13:0] sil_read_addr;
```

```
 wire sil_read_data;
 sil_ram sil_ram1(    .clka(clk),.dina(sil_write_data),
                                   .addra(sil_write_addr), .wea(sil_write_enable),
                                   .clkb(clk), .addrb(sil_read_addr),
                                   .doutb(sil_read_data));


 wire step_dbg;
 wire [8:0] x_dbg;
 wire [8:0] y_dbg;
 wire [11:0] diff_dbg;

 wire [18:0] proc_raddr;
 wire [35:0] proc_rdata;

 wire [11:0] cur_block_shape;

 image_analysis ia1 (.clk(clk), .enable(~blank), .restart({vcount == 0}),
      .zbt_read_addr(proc_raddr), .zbt_read_data(proc_rdata),
      .cur_block_shape(cur_block_shape),
      .occupied_threshold({3'd0,hex_in[11:8], 4'd0}),
      .block_diff_threshold({4'b0000, hex_in[7:0]}),
      .sil_waddr(sil_write_addr), .sil_wdata(sil_write_data),
      .sil_we(sil_write_enable),
      .x_dbg(x_dbg), .y_dbg(y_dbg), .step_dbg(step_dbg),
      .diff_dbg(diff_dbg)
     );


 // 3 bit color VGA output for silhouette display
 wire sil_disp_r;
 wire sil_disp_g;
 wire sil_disp_b;

 sil_display sd1 (.clk(clk), .hcount(hcount), .vcount(vcount),
      .sil_read_addr(sil_read_addr), .sil_read_data(sil_read_data),
      .r(sil_disp_r), .g(sil_disp_g), .b(sil_disp_b));


// ADV7185 NTSC decoder interface code
// adv7185 initialization module
adv7185init adv7185(.reset(0), .clock_27mhz(clock_27mhz),
            .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
            .tv_in_i2c_clock(tv_in_i2c_clock),
            .tv_in_i2c_data(tv_in_i2c_data));

wire [29:0] ycrcb;    // video data (luminance, chrominance)
wire [2:0] fvh;    // sync for field, vertical, horizontal
wire        dv;    // data valid

ntsc_decode decode (.clk(tv_in_line_clock1), .reset(0),
            .tv_in_ycrcb(tv_in_ycrcb[19:10]),
            .ycrcb(ycrcb), .f(fvh[2]),
            .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

 wire [7:0] red;
 wire [7:0] green;
 wire [7:0] blue;
 wire [18:0] rgb = {red[7:2], green[7:2], blue[7:2]};

 YCrCb2RGB convert (.R(red), .G(green), .B(blue),
                             .clk(tv_in_line_clock1), .rst(0),
                             .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]), .Cb(ycrcb[9:0]));

// code to write NTSC data to video memory

wire [18:0] ntsc_waddr;
wire [35:0] ntsc_wdata;
wire        ntsc_we;
 wire new_frame;
ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, rgb,
               ntsc_waddr, ntsc_wdata, ntsc_we, switch[6], new_frame);

   wire [7:0]  vga_red;
        wire[7:0] vga_green;
        wire [7:0] vga_blue;
   reg         b,hs,vs;



        reg [7:0] vga_r_old;
        reg [7:0] vga_g_old;
        reg [7:0] vga_b_old;
        reg clk_half;
        reg clk_out;

        always @ (posedge clk) begin
               clk_half <= ~clk_half;
               clk_out <= clk_half && !b;
               vga_r_old <= vga_red;
               vga_g_old <= vga_green;
               vga_b_old <= vga_blue;
        end

        assign analyzer3_clock = clk_out;
        assign analyzer1_data = {vga_red, vga_green};
        assign analyzer3_data = {b,hs,vs, vcount[9:5], vga_blue};

        assign analyzer2_data = {vga_r_old, vga_g_old};
        assign analyzer4_data = {hcount[10:3], vga_b_old};

/*
    wire [18:0] pattern_addr;
    wire [35:0] pattern_data;
    zbt_pattern pt1(.clk(clk), .write_addr(pattern_addr),
                                          .write_data(pattern_data));
*/

    wire preview = switch[2];

    // wire up the memory manager
    memory_manager mm1(.clk(clk), .blank(blank), .switch(new_frame), .preview(preview),
         .ntsc_waddr(ntsc_waddr), .ntsc_wdata(ntsc_wdata), .ntsc_we(ntsc_we),
         .disp_raddr(disp_raddr), .disp_rdata(disp_rdata),
         .proc_raddr(proc_raddr), .proc_rdata(proc_rdata),
         .zbt_0_we(zbt_0_we), .zbt_0_addr(zbt_0_addr),
         .zbt_0_wdata(zbt_0_wdata), .zbt_0_rdata(zbt_0_rdata),
         .zbt_1_we(zbt_1_we), .zbt_1_addr(zbt_1_addr),
         .zbt_1_wdata(zbt_1_wdata), .zbt_1_rdata(zbt_1_rdata)
        );


   // select output pixel data

    wire [9:0] score;
    wire [2:0] boardColor;
    wire [8:0] coord;
    wire rowFilled;
    wire isFallingPiece;
    wire gameOver;


    display ds1(.vclock(clk), .reset(0), .hcount(hcount),
         .vcount(vcount), .hsync(hsync), .vsync(vsync), .blank(blank),
         .score(score),
         .boardColor(boardColor),
         .sil_r(sil_disp_r),    .sil_g(sil_disp_g), .sil_b(sil_disp_b),
         .preview(preview),
         .preview_r(vr_pixel_r), .preview_g(vr_pixel_g), .preview_b(vr_pixel_b),
         .r(vga_red), .g(vga_green), .b(vga_blue),
         .coord(coord), .isFallingPiece(isFallingPiece),
         .rowFilled(rowFilled),
      .gameOver(gameOver));
```

```
// UP and DOWN buttons
wire up,down,left,right;
debounce db2(.reset(reset),.clk(clock_65mhz),.noisy(~button_up),.clean(up));
debounce db3(.reset(reset),.clk(clock_65mhz),.noisy(~button_down),.clean(down));
debounce db4(.reset(reset),.clk(clock_65mhz),.noisy(~button_left),.clean(left));
debounce db5(.reset(reset),.clk(clock_65mhz),.noisy(~button_right),.clean(right));

 wire left_switch, right_switch;
debounce db6(.reset(reset),.clk(clock_65mhz),.noisy(user3[1]),.clean(left_switch));
debounce db7(.reset(reset),.clk(clock_65mhz),.noisy(user3[0]),.clean(right_switch));

 wire real_left = left | left_switch;
 wire real_right = right | right_switch;


 wire [11:0] rev_block;
 assign rev_block[0] = cur_block_shape[11];
 assign rev_block[1] = cur_block_shape[10];
 assign rev_block[2] = cur_block_shape[9];
 assign rev_block[3] = cur_block_shape[8];
 assign rev_block[4] = cur_block_shape[7];
 assign rev_block[5] = cur_block_shape[6];
 assign rev_block[6] = cur_block_shape[5];
 assign rev_block[7] = cur_block_shape[4];
 assign rev_block[8] = cur_block_shape[3];
 assign rev_block[9] = cur_block_shape[2];
 assign rev_block[10] = cur_block_shape[1];
 assign rev_block[11] = cur_block_shape[0];


FSM fsm1(.clk(clock_65mhz),
      .reset(reset),
      .blanking(next_vblank),
      .FP(rev_block),
      .left(real_left),
      .right(real_right),
      .coord(coord),
      .score(score),
      .blockColor(boardColor),
      .up(up),
      .down(down),
      .startLevel({switch[1],switch[0]}),
      .rowFilled(rowFilled),
      .isFallingPiece(isFallingPiece),
      .gameOver(gameOver));




always @(posedge clk)
     begin
 b <= blank;
 hs <= hsync;
 vs <= vsync;
     end

// VGA Output.  In order to meet the setup and hold times of the
// AD7125, we send it ~clk.
```

```
   assign vga_out_red = vga_red;
   assign vga_out_green = vga_green;
   assign vga_out_blue = vga_blue;
   assign vga_out_sync_b = 1'b1;      // not used
   assign vga_out_pixel_clock = ~clk;
   assign vga_out_blank_b = ~b;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

   // debugging

   assign led = ~{6'b0, real_left, real_right};

    assign user4[11:0] = ~cur_block_shape;

endmodule

///////////////////////////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)

module xvga(vclock,hcount,vcount,hsync,vsync,blank,next_vblank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output     vsync;
   output     hsync;
   output     blank;
    output    next_vblank;

   reg       hsync,vsync,hblank,vblank,blank;
   reg [10:0]      hcount;    // pixel number on current line
   reg [9:0] vcount;     // line number

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire        hsyncon,hsyncoff,hreset,hblankon;
   assign      hblankon = (hcount == 1023);
   assign      hsyncon = (hcount == 1047);
   assign      hsyncoff = (hcount == 1183);
   assign      hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire        vsyncon,vsyncoff,vreset,vblankon;
   assign      vblankon = hreset & (vcount == 767);
   assign      vsyncon = hreset & (vcount == 776);
   assign      vsyncoff = hreset & (vcount == 782);
   assign      vreset = hreset & (vcount == 805);

   // sync and blanking
   wire        next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
        hcount <= hreset ? 0 : hcount + 1;
        hblank <= next_hblank;
        hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

        vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
        vblank <= next_vblank;
        vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

        blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule

module vram_display(reset,clk,hcount,vcount,vr_pixel_r,vr_pixel_g,vr_pixel_b,
               vram_addr,vram_read_data);

   input reset, clk;
   input [10:0] hcount;
   input [9:0]     vcount;
   output [7:0] vr_pixel_r;
   output [7:0] vr_pixel_g;
   output [7:0] vr_pixel_b;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

    parameter MAX_HCOUNT = 1344;
    parameter LOOKAHEAD = 6;
    parameter WRAP_HCOUNT = MAX_HCOUNT - LOOKAHEAD;

   //forecast hcount & vcount 8 clock cycles ahead to get data from ZBT
   wire[10:0] hcount_f = (hcount >= WRAP_HCOUNT) ? (hcount - WRAP_HCOUNT) : (hcount + LOOKAHEAD);
    wire [9:0] vcount_f = (hcount >= WRAP_HCOUNT) ? ((vcount == 805) ? 0 : vcount+1) : vcount;


   reg [18:0]      vram_addr;
   reg [7:0]      vr_pixel_r;
   reg [7:0]      vr_pixel_g;
   reg [7:0]      vr_pixel_b;
   reg [35:0]      vr_data_latched;
   reg [35:0]      last_vr_data;


   always @(posedge clk) begin

        vram_addr <= {hcount_f[10:9] != 0, 1'd0, vcount_f[8:0], hcount_f[8:1]};


       last_vr_data <= (hcount_f[0]==1) ? vram_read_data : last_vr_data;

       //vr_data_latched <= (hc4==2'd1) ? vram_read_data : vr_data_latched;
        // each 36-bit word from RAM is decoded to 4 bytes
        if (hcount_f[0]==1) begin
               vr_pixel_r <= {last_vr_data[17:12],2'b00};
               vr_pixel_g <= {last_vr_data[11:6],2'b00};
               vr_pixel_b <= {last_vr_data[5:0],2'b00};
        end else begin
               vr_pixel_r <= {last_vr_data[17+18:12+18],2'b00};
               vr_pixel_g <= {last_vr_data[11+18:6+18],2'b00};
               vr_pixel_b <= {last_vr_data[5+18:0+18],2'b00};
        end
    end

endmodule // vram_display
```

FSM.v

```
/////////////////////////////////////////////////////////////////////////////////
//
//           Module FSM (Controls Tetris Game Logic)
//
/////////////////////////////////////////////////////////////////////////////////

module FSM(    input wire clk,
                input wire reset,
                input wire blanking,
                input wire [0:11] FP,
                input wire left,
                input wire right,
                input wire [8:0] coord,
                input wire up,
                input wire down,
                input wire [1:0] startLevel,
                output reg [9:0] score,
                output reg [2:0] blockColor,
                output reg rowFilled,
                output reg isFallingPiece,
                output wire gameOver
    );

  wire cross = (FP == 12'b000010111010);
  assign gameOver = (state == GAME_OVER);

  parameter GAME_OVER = 0;
  parameter FALLING = 1;
  parameter CLEARING = 2;
  parameter FLASH = 3;
  parameter COLLAPSE = 4;
  parameter RESET = 5;

  //in .1 seconds
  parameter FLASH_SPEED = 1;
  reg [3:0] falling_speed = 5;

  parameter LOGSIZE = 5;
  parameter WIDTH = 32;
  //Falling Piece (FP) indices:
  /*
    0 1 2
    3 4 5
    6 7 8
    9 10 11
  */
  // # # # [# # #] # # # #
  reg [LOGSIZE-1:0] a1 = 0;
  reg [LOGSIZE-1:0] a2 = 0;
  reg [LOGSIZE-1:0] aR = 0;
  reg [LOGSIZE-1:0] aW = 0;

  reg we1 = 0;
  reg we2 = 0;
  reg we = 0;

  reg [WIDTH-1:0] mIn1;
  reg [WIDTH-1:0] mIn2;
  reg [WIDTH-1:0] mInR;
  reg [WIDTH-1:0] mInW;

  wire [WIDTH-1:0] mOut1;
  wire [WIDTH-1:0] mOut2;
  wire [WIDTH-1:0] mOutR;
  wire [WIDTH-1:0] mOutW;
  //playing field - Double buffer
  //Falling piece 32 x 32bit memory block
  bram #(.LOGSIZE(LOGSIZE),.WIDTH(WIDTH))
          PF(.addr(a1),.clk(clk),.we(we1),.din(mIn1),.dout(mOut1));
  bram #(.LOGSIZE(LOGSIZE),.WIDTH(WIDTH))
          PF1(.addr(a2),.clk(clk),.we(we2),.din(mIn2),.dout(mOut2));

  //write switch (ws BRAM is writing)
  reg ws = 0;
  assign mOutW = (ws ? mOut2 : mOut1);
  assign mOutR = (ws ? mOut1 : mOut2);

  //Switching logic that abstracts out the double-switching-buffers into two Read and Write BRAMs
  always @(*) begin
    we1 = (ws ? 0 : we);
    we2 = (!ws ? 0 : we);
    mIn1 = (ws ? mInR : mInW);
    mIn2 = (!ws ? mInR : mInW);
    a1 = (ws ? aR : aW);
    a2 = (!ws ? aR : aW);
  end

  //counter to properly setup collision reading address
  reg [1:0] waitCo = 2;

  //index for falling block
  reg [4:0] fi = 0;
  //indicate the position of falling piece at any given time.
  reg [3:0] FP_x = 3;
  //FP_x counting by 3's
  reg [5:0] P_x = 9;
  reg [4:0] FP_y = 0;

  reg [2:0] state = 0;
  reg startClearing = 0;
  //color of current falling piece
  reg [2:0] color;

  reg go, co;

  //test if the falling piece is empty and need to pause the game
  wire empty = !(|FP[0:11]);
  //check if the top row is nonempty
  //speeds of flashing and falling piece
  reg [3:0] speed = 5;
  reg [2:0] flash_speed = FLASH_SPEED;
  //All the counters to keep track of left/right/up/down button presses
  reg rpressed = 0;
  reg lpressed = 0;
  reg upressed = 0;
  reg dpressed = 0;
  reg ry = 0;
  reg ly = 0;
  reg uy = 0;
  reg dy = 0;
  //bool to check if any rows were cleared
  reg clr = 0;
  //row keeping track the row # to collapse
  reg [4:0] corow = 21;
  //checks to see if row is complete
  wire complete = !((mOutR[2:0] == 3'b000) || (mOutR[5:3] == 3'b000) || (mOutR[8:6] == 3'b000) ||
(mOutR[11:9] == 3'b000) || (mOutR[14:12] == 3'b000) || (mOutR[17:15] == 3'b000) || (mOutR[20:18]
== 3'b000) || (mOutR[23:21] == 3'b000) || (mOutR[26:24] == 3'b000) || (mOutR[29:27] == 3'b000));

  reg [31:0] counter = 0;

  always @(posedge clk) begin

    if(reset) begin
      state <= RESET;
    end

    //only step through FSM if monitor is blanking. (do I even need to do this?)
    if(blanking && !reset) begin
      //Counter that determines the speed of falling block.
      if(counter == 139999 || reset) begin
        counter <= 0;
        ena <= !ena;
      end
      else begin
        counter <= counter + 1;
      end

      //Logic that handles up/down button presses to increase and decrease block falling speed
      if(up) begin
        if(!upressed) begin
          upressed <= 1;
          uy <= 1;
        end
      end else begin
        upressed <= 0;
        uy <= 0;
      end
      if(down) begin
        if(!dpressed) begin
          dpressed <= 1;
          dy <= 1;
        end
      end else begin
        dpressed <= 0;
        dy <= 0;
      end

      if(uy && !dy && (falling_speed > 3)) begin
        falling_speed <= falling_speed - 1;
        uy <= 0;
      end
      if(dy && !uy && (falling_speed < 15)) begin
        falling_speed <= falling_speed + 1;
        dy <= 0;
      end


      //--------------------------State Machine--------------------------


      case(state)
        GAME_OVER: begin
                  //keep old playing field and look for either a reset
                  //or cross shaped player indicating a new game
                  if(cross) begin
                    state <= RESET;
                    startClearing <= 0;
                  end
                  else begin
                    state <= GAME_OVER;
                  end
                end
        FALLING:   begin
                  if(counter == 0)begin
                    //make sure the piece is not empty (user paused) and on .1 second pulse steps
we calculate logic.
                    if(speed > 0) begin
                      speed <= speed - 1;
                    end
                  end


                  if(speed >= 1) begin
                    //handle left/right movement separately.
                    if(left) begin
```

```
                      if(!lpressed) begin
                        lpressed <= 1;
                        ly <= 1;
                      end
                    end else begin
                      lpressed <= 0;
                      ly <= 0;
                    end
                    if(right) begin
                      if(!rpressed) begin
                        rpressed <= 1;
                        rpressed <= 1;
                        ry <= 1;
                      end
                    end else begin
                      rpressed <= 0;
                      ry <= 0;
                    end

                    if(ly && !ry && FP_x > 0) begin

                      if(waitCo == 2) begin
                        //start reading memory from where falling piece is currently
                        aR <= FP_y;
                        fi <= 0;
                        co <= 0;
                        we <= 0;
                        state <= FALLING;
                        waitCo <= 1;
                      end else if(waitCo == 1) begin
                        //iteratively checking for collision
                        if(aR >= (FP_y+1)) begin
                          if(((aR-1) >= FP_y) && ((aR-1) <= FP_y + 3)) begin
                            if(fi <= 9) begin
  //
                              co <= (co ||    (FP[fi] && (mOutR[P_x-3] || mOutR[P_x-3+1] ||
mOutR[P_x-3+2])) ||
                                  (FP[fi+1] && (mOutR[P_x-3+3] || mOutR[P_x-3+4] || mOutR[P_x-
3+5])) ||
                                  (FP[fi+2] && (mOutR[P_x-3+6] || mOutR[P_x-3+7] || mOutR[P_x-
3+8])));
  //
                            end
                            fi <= ((fi < 15) ? fi + 3 : 15);
                          end
                        end


                        if(fi < 15) begin
                          aR <= aR + 1;
                        end else begin
                          waitCo <= 0;
                          startClearing <= 0;
                          aR <= 0;
                          aW <= 5'b11111;
                          we <= 0;
                        end
                        state <= FALLING;
                      end else begin
                        //update coords only if no collision detected
                        if(!co) begin
                          FP_x <= FP_x - 1;
                          P_x <= P_x - 3;
                          co <= 0;
                        end
                        ly <= 0;
                        waitCo <= 2;
                      end
                    end
                    if(ry && !ly && FP_x < 7) begin
                      if(waitCo == 2) begin
```

```
      //start reading memory from where falling piece is currently
      aR <= FP_y;
      fi <= 0;
      co <= 0;
      we <= 0;
      state <= FALLING;
      waitCo <= 1;
    end else if(waitCo == 1) begin
      //iteratively checking for collision
      if(aR >= (FP_y+1)) begin
        if(((aR-1) >= FP_y) && ((aR-1) <= FP_y + 3)) begin
          if(fi <= 9) begin
              co <= (co ||
    (FP[fi] && (mOutR[P_x+3] || mOutR[P_x+3+1] || mOutR[P_x+3+2])) ||
  (FP[fi+1] && (mOutR[P_x+3+3] || mOutR[P_x+3+4] || mOutR[P_x+3+5])) ||
   (FP[fi+2] && (mOutR[P_x+3+6] || mOutR[P_x+3+7] || mOutR[P_x+3+8])));
          end
          fi <= ((fi < 15) ? fi + 3 : 15);
        end
      end


      if(fi < 15) begin
        aR <= aR + 1;
      end else begin
        waitCo <= 0;
        startClearing <= 0;
        aR <= 0;
        aW <= 5'b11111;
        we <= 0;
      end
      state <= FALLING;
    end else begin
      if(!co) begin
        FP_x <= FP_x + 1;
        P_x <= P_x + 3;
        co <= 0;
      end
      ry <= 0;
      waitCo <= 2;
    end
  end
  state <= FALLING;
end

//Make falling piece fall one row if no collision detected
if(speed==0) begin
  //if empty, stop calculating
  if(empty) begin
    speed <= falling_speed;
    startClearing <= 0;
    waitCo <= 2;
    state <= FALLING;
  end else begin
    if(waitCo == 2) begin
      //start reading memory from where falling piece is currently
      aR <= FP_y;
      fi <= 0;
      co <= 0;
      go <= 0;
      we <= 0;
      state <= FALLING;
      waitCo <= 1;
    end else if(waitCo == 1) begin
      //iteratively checking for collision
      if(aR >= (FP_y+1)) begin
        if(((aR-1) >= FP_y) && ((aR-1) <= FP_y + 4)) begin
          if(fi <= 9) begin
            go <= (go ||
                (FP[fi] && (mOutR[P_x] || mOutR[P_x+1] || mOutR[P_x+2])) ||
                (FP[fi+1] && (mOutR[P_x+3] || mOutR[P_x+4] || mOutR[P_x+5])) ||
                                  (FP[fi+2] && (mOutR[P_x+6] || mOutR[P_x+7] || mOutR[P_x+8])));
                            end
                            if(fi > 2 && fi < 15) begin
                                co <= (co ||
                                    (FP[fi-3] && (mOutR[P_x] || mOutR[P_x+1] || mOutR[P_x+2])) ||
                                    (FP[fi-2] && (mOutR[P_x+3] || mOutR[P_x+4] || mOutR[P_x+5]))
||
                                    (FP[fi-1] && (mOutR[P_x+6] || mOutR[P_x+7] || mOutR[P_x+8]))
);
                            end
                            fi <= ((fi < 15) ? fi + 3 : 15);
                          end
                        end


                        if(fi < 15) begin
                          aR <= aR + 1;
                        end else begin
                          waitCo <= 0;
                          startClearing <= 0;
                          aR <= 0;
                          aW <= 5'b11111;
                          we <= 0;
                        end
                        state <= FALLING;
                      end else begin
                        //waitCo = 0, freeze here until falling finishes calculating
                        if(go && (FP_y == 0) && (FP_x == 3)) begin
                          state <= GAME_OVER;
                          startClearing <= 0;
                          speed <= falling_speed;
                        end else begin
                          if(co) begin
                            //don't calculate until collision testing is done calculating
                            if(!startClearing) begin
                              aR <= 0;
                              aW <= 5'b11111;
                              we <= 0;
                              fi <= 0;
                              startClearing <= 1;
                              state <= FALLING;
                            end else begin
                              //only write when aR == 2
                              if(aR >= 1) begin
                                //If collision, just copy over, don't move
                                if(((aR-1) < FP_y) || ((aR-1) > (FP_y+3))) begin
                                  mInW <= mOutR;
                                end else begin
                                  mInW <= ({22'b0, {{3{FP[fi+2]}}, {3{FP[fi+1]}}, {3{FP[fi]}}} &
{3{{color[0], color[1], color[2]}}}
                                        }<<(P_x)) | mOutR;

                                  fi <= ((fi < 11) ? fi + 3 : 12);
                                end
                              end

                              //need to wait 2 more cycles for values to be calulated before
switching
                              if(aR < 23) begin
                                //have not swept through the whole thing yet
                                aR <= aR + 1;
                                if(aR >= 1) begin
                                  aW <= aW + 1;
                                  we <= 1;
                                end
                                state <= FALLING;
                              end else begin
                                //stop writing
                                ws <= !ws;
                                we <= 0;
                                aR <= 0;
                                aW <= 5'b11111;
                                speed <= falling_speed;
                                go <= 0;
                                co <= 0;
                                startClearing <= 0;
                                waitCo <= 2;
                                clr <= 0;
                                state <= CLEARING;
                              end
                            end


                          end else begin
                            //main purpose
                            FP_y <= FP_y + 1;
                            //setup logic
                            we <= 0;
                            speed <= falling_speed;
                            startClearing <= 0;
                            waitCo <= 2;
                            state <= FALLING;
                          end

                        //end of !go
                        end
                      //end of else after co/go calculation
                      end
                    //end of empty else
                    end
                  //end of speed == 0
                  end

                end

        CLEARING:  begin
                //All the buffers should be clean and the switch in the correct place
                //need to clear rows in PF, find all solid rows and mark them for clearing
                if(!startClearing) begin
                  //refresh the addresses as well as clean the pipes
                  aR <= 0;
                  aW <= 5'b11111;
                  we <= 0;
                  startClearing <= 1;
                  state <= CLEARING;
                end else begin
                  if(aR >= 1) begin
                    if(complete && (aR < 22)) begin
                      mInW <= {2'b10, 30'b000000000000000000000000000000};
                      clr <= 1;
                    end else begin
                      mInW <= mOutR;
                    end
                  end

                  if(aR < 22) begin
                    aR <= aR + 1;
                    if(aR >= 1) begin
                      aW <= aW + 1;
                      we <= 1;
                    end
                    state <= CLEARING;
                  end else begin
                    ws <= !ws;
                    aR <= 0;
                    aW <= 5'b11111;
                    we <= 0;
                    flash_speed <= FLASH_SPEED;
                    speed <= falling_speed;
                    state <= FLASH;
                  end
                end
```

```
      end
FLASH:     begin
        //Waits while Display flashes the rows to be cleared
        if(!clr) begin
          state <= COLLAPSE;
        end else begin
          //ONLY SIMULATION
          if(counter == 0) begin
            speed <= ((speed > 0) ? (speed - 1) : speed);
          end else begin
            if(speed == 0) begin
              //finished flashing, next_state is collapse
              flash_speed <= FLASH_SPEED;
              speed <= falling_speed;
              we <= 0;
              aR <= 0;
              startClearing <= 0;
              corow <= 21;
              state <= COLLAPSE;
              aW <= 5'b11111;
            end else begin
              state <= FLASH;

            end

          end

        end
      end
COLLAPSE:  begin
        //0000000000000000000000
        //0000000000000000000000
        //0000000000000000000000
        //000000000000000000000#

        if(!clr) begin
          //the only way out of this state is through here
          we <= 0;
          aR <= 0;
          aW <= 5'b11111;
          speed <= falling_speed;
          go <= 0;
          co <= 0;
          startClearing <= 0;
          waitCo <= 2;
          FP_x <= 3;
          P_x <= 9;
          FP_y <= 0;
          color <= ((color == 3'b111) ? 3'b001 : color+1);
          corow <= 21;
          state <= FALLING;
        end else begin
          //handle general row shifting
          if(corow < 21) begin
            //shift all rows down to corow
            if(!startClearing) begin
              //refresh the addresses as well as clean the pipes
              aR <= 0;
              aW <= 5'b11111;
              we <= 0;
              startClearing <= 1;
            end else begin
              if(aR >= 1) begin
                //override row with the read in prevRow
                if((aR-1) == 21) begin
                  mInW <= {2'b00, 30'b111111111111111111111111111111};
                end else begin
                  mInW <= mOutR;
                end
              end else begin
```

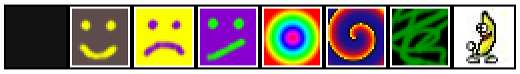
```
                        mInW <= 32'b00000000000000000000000000000000;
                      end

                      if(aR < 23) begin
                        aR <= aR + 1;
                        //if read is ready and current row isn't marked out row, keep going
                        if(((aR-1) != (corow))) begin// || ((aR-1) != (corow+2))) begin
                          aW <= aW + 1;
                          we <= 1;
                        end else begin
                          //else stop writing for one cycle and wait for read address to catch
up.
                          we <= 0;
                        end
                      end else begin
                        //clearing corow, flip memory and start over
                        corow <= 21;
                        score <= score + 1;
                        ws <= !ws;
                        aR <= 0;
                        aW <= 5'b11111;
                        we <= 0;
                        startClearing <= 0;
                      end
                    end
                    state <= COLLAPSE;
                  end else begin
                    //looking for the marked row (corow) for clearing
                    //reading ONLY
                    if(!startClearing) begin
                      //refresh the addresses as well as clean the pipes
                      aR <= 0;
                      we <= 0;
                      startClearing <= 1;
                    end else begin
                      if(aR >= 1) begin
                        //found the row!
                        if(mOutR[31]) begin
                          //save row for collapsin'
                          corow <= aR - 1;
                          startClearing <= 0;
                          aR <= 0;
                          aW <= 0;
                        end
                      end

                      if(aR < 22) begin
                        aR <= aR + 1;
                      end else begin
                        aR <= 0;
                        aW <= 0;
                        we <= 0;
                        //no more rows to collapse
                        clr <= 0;
                      end
                    end
                    state <= COLLAPSE;
                  end


                //end of clr
                end


              end
        RESET:     begin
                if(!startClearing) begin
                  //clear the playing board and falling piece, reset to state FALLING
                  co <= 0;
                  go <= 0;
                  FP_x <= 3;
```

```
  P_x <= 9;
  FP_y <= 0;
  speed <= falling_speed;
  flash_speed <= FLASH_SPEED;
  score <= 0;
  clr <= 0;
  aR <= 0;
  aW <= 5'b11111;
  mInR <= 0;
  mInW <= 0;
  corow <= 21;
  we <= 0;
  falling_speed <= 5;
  rowFilled <= 0;
  color <= 3'b001;
  startClearing <= 1;
  state <= RESET;
end else begin
  //write to the whole board 0's except row 20 where it fills solid
  //(aR - 1) < 19
  if(aR != 21) begin
    //Difficulty levels for starting board
    if (startLevel == 0) begin
      mInW <= 32'b0;
    end else if (startLevel == 1) begin
      case (aR)
        18: mInW <= {2'b00, 30'b000000000000000000000000111000};
        19: mInW <= {2'b00, 30'b000000111111111000000000111111};
        20: mInW <= {2'b00, 30'b000111111000111111000000111111};
        default: mInW <= 32'b0;
      endcase
    end else if (startLevel == 2) begin
      case (aR)
        10: mInW <= {2'b00, 30'b000000111000000000000000000000};
        11: mInW <= {2'b00, 30'b000000111000000000000000000000};
        12: mInW <= {2'b00, 30'b000000111000000000000000000000};
        13: mInW <= {2'b00, 30'b000000111000000000000000000000};
        14: mInW <= {2'b00, 30'b000000111000000000000000000000};
        15: mInW <= {2'b00, 30'b000000111000000000000000000000};
        16: mInW <= {2'b00, 30'b000000111000000000000000000000};
        17: mInW <= {2'b00, 30'b000000111000000000000000000000};
        18: mInW <= {2'b00, 30'b000000111000000111000111000000};
        19: mInW <= {2'b00, 30'b111000111111111000111000111111};
        20: mInW <= {2'b00, 30'b000111000111000111000111000111};
        default: mInW <= 32'b0;
      endcase
    end else if (startLevel == 3) begin
      case (aR)
         8: mInW <= {2'b00, 30'b000000000111111111000000111000};
         9: mInW <= {2'b00, 30'b000000000111111111111000111000};
        10: mInW <= {2'b00, 30'b000000000111111111111000111000};
        11: mInW <= {2'b00, 30'b000000000111111111000000111000};
        12: mInW <= {2'b00, 30'b000000000111111111000111111000};
        13: mInW <= {2'b00, 30'b000000000111111111000111111000};
        14: mInW <= {2'b00, 30'b111111111111111000000000111111};
        15: mInW <= {2'b00, 30'b111111000000000000000000000111};
        16: mInW <= {2'b00, 30'b111111000000000000000000000111};
        17: mInW <= {2'b00, 30'b111111111000000000111000000111};
        18: mInW <= {2'b00, 30'b111111000000000000000000000111};
        19: mInW <= {2'b00, 30'b111111000000000000000000000111};
        20: mInW <= {2'b00, 30'b111111000000000000000000000111};
        default: mInW <= 32'b0;
      endcase
    end
  end else begin
    mInW <= {2'b00, 30'b111111111111111111111111111111};
  end


  if(aR < 30) begin
    aW <= aW + 1;
```

```
                    aR <= aR + 1;
                    we <= 1;
                    state <= RESET;
                  end else begin
                    ws <= !ws;
                    we <= 0;
                    aW <= 0;
                    aR <= 0;
                    state <= FALLING;
                    startClearing <= 0;
                    wait2 <= 2;
                  end


                end

              end
        default: begin
                state <= GAME_OVER;
              end

      endcase

    end else begin
      //When not blanking, handle coord requests from Display and output the blockColor at
coordinate coord
      aR <= (coord[4:0]+1);
      if(mOutR[31]) begin
        blockColor <= 3'b000;
        rowFilled <= 1;
      end else begin
        rowFilled <= 0;
        if(coord[8:5] >= FP_x && coord[8:5] < FP_x+3 &&
          (coord[4:0]+1) >= (FP_y) && (coord[4:0]+1) < (FP_y+4) &&
          {mOutR[3*coord[8:5]], mOutR[3*coord[8:5]+1], mOutR[3*coord[8:5]+2]} == 3'b000) begin
            blockColor <= (FP[3*(coord[4:0]-FP_y+1)+(coord[8:5]-FP_x)] ? color : 3'b000);
            isFallingPiece <= 1;
        end else begin
          isFallingPiece <= 0;
          blockColor <= {mOutR[3*coord[8:5]], mOutR[3*coord[8:5]+1], mOutR[3*coord[8:5]+2]};
        end
      end
    end
  end

endmodule
```

Appendix B: ROM Images

Playing field tiles



Logo

ImprovTetris
By Scott Bezek and Ray Li

Score Text

Score Number