ImprovTetris Proposal

Scott Bezek
Ray Li

Overview

The game of Tetris has entertained people for ages. The classic implementation features randomly picked falling blocks that the user must orient and piece together like a jig-saw puzzle to complete and clear as many contiguous rows of blocks on the bottom as fast as possible before everything piles up above the top. ImprovTetris aims to make the classic Tetris game even more fun on the FPGA by taking in controller input from real-time video-streaming of the player instead of traditional key presses. In this game, the shape of the falling block corresponds to the quantized shape of the player's silhouette captured on video and its left/right movement corresponds to left/right button presses on the FPGA. We also give the user visual feedback of his calculated-silhouette as well as the quantized shape for the falling block on the screen.

The implementation of ImprovTetris is broken down into two major subsystems: the image-processing subsystem and the Tetris game logic. The following sections describe how these subsystems function and interact.

Image-processing Subsystem (Scott)

The image-processing subsystem handles 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. 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 5x5 grid - this is the custom Tetris piece that falls in the game.

Modules:

1. **video_decoder** - [Provided by 6.111 staff] Inputs signals from a 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** - Converts the 30 bit YCrCb data into an 18 bit RGB value.
3. **ntsc_to_zbt** - Stores each frame of NTSC video (as 18-bit RGB values) to ZBT memory
4. **image_processing** - Reads the current video frame out of memory and compares it to the reference frame. Calculates the silhouette and outputs this as a 5x5 Tetris piece

Game-Logic (Ray)

1. **Major FSM** - FSM that governs the tetris game logic
2. **Display** - Display logic that pieces together the game board as well as the control panel on the side
3. **BRAM** - Stores 40 32-bit cells that represent two game boards. One board holds the dropping block and the other the fallen blocks on the bottom. Each 32-bit represents each row of the playing board (3-bit color for each of 10 blocks with 2-bit at the end to keep track of animation and clearing of the row).

ImprovTetris Block Diagram

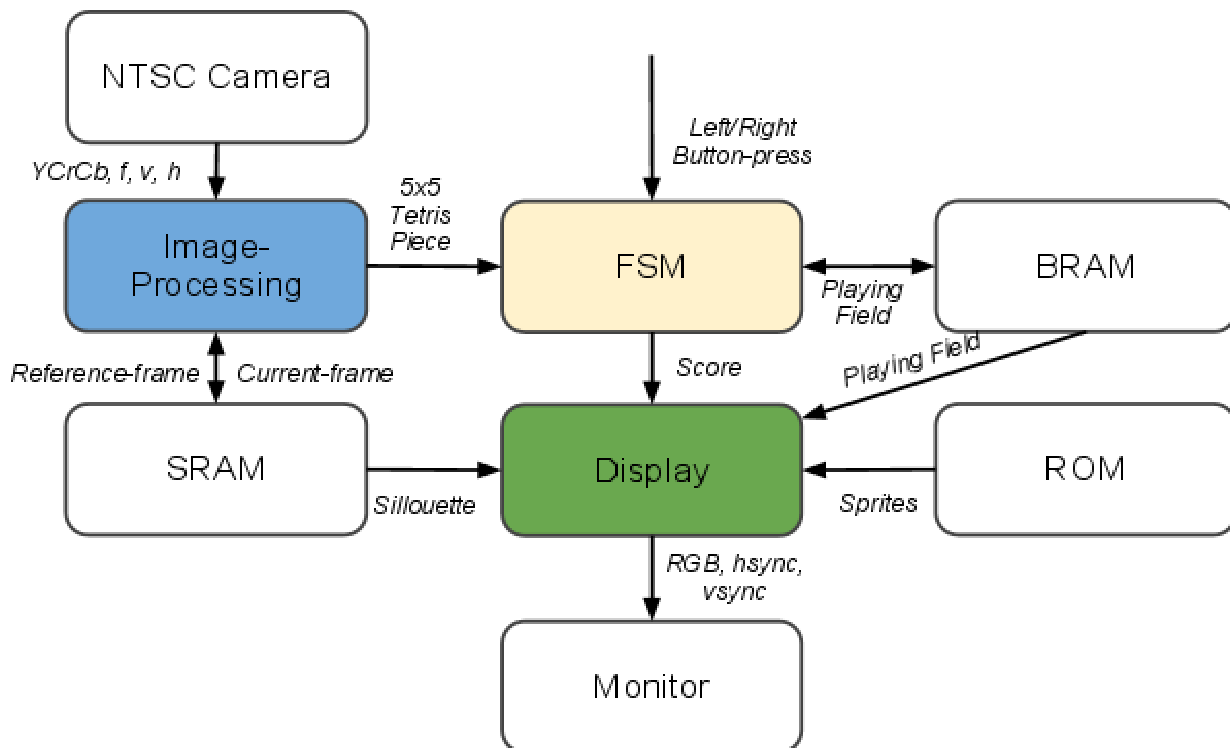
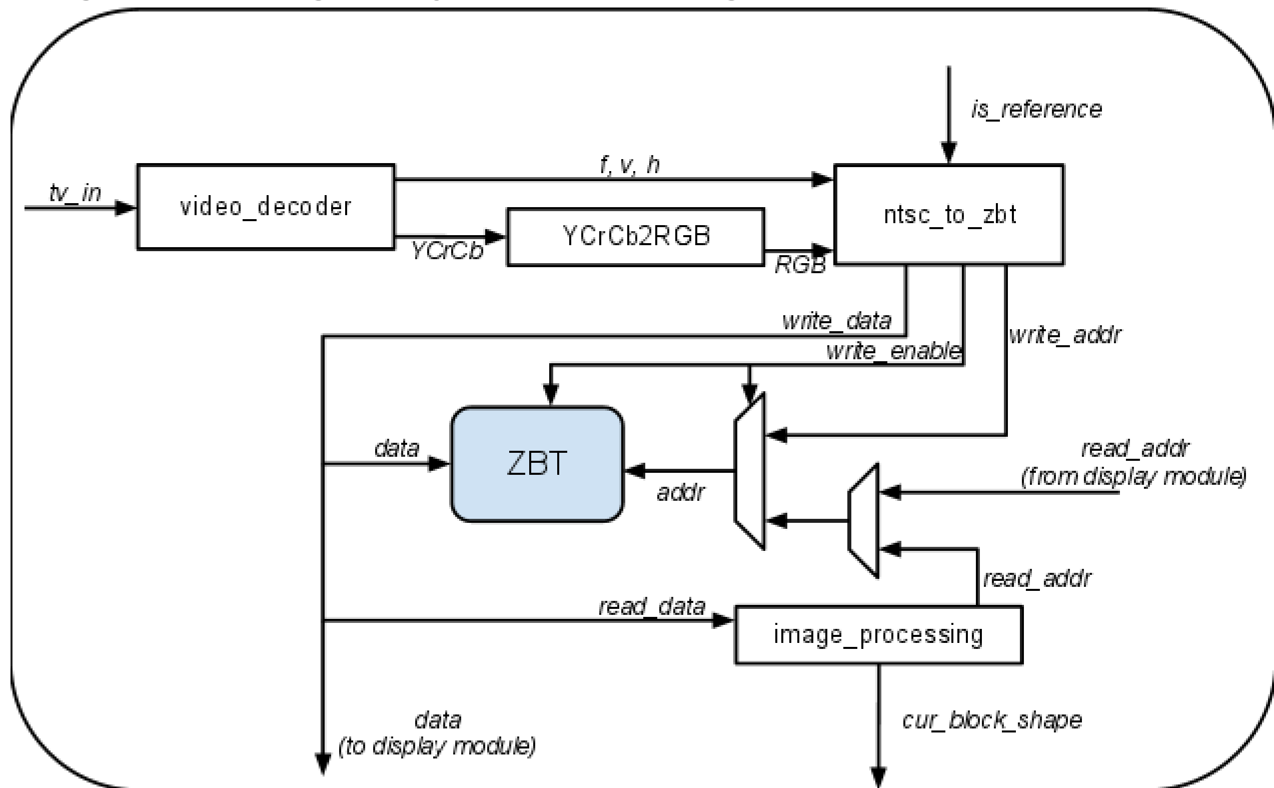


Image-processing Subsystem Block Diagram:



Detailed Description of Major Blocks

ntsc_to_zbt

One of the major challenges of the image-processing subsystem is storing and accessing the camera images in memory. ZBT memory cannot simultaneously read and write at different addresses, so we need to carefully specify when image data is being stored vs. accessed. There are also constraints on the amount of data that the ZBT memory can hold.

This 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. Since each pixel takes up 18 bits, we will store 2 pixels per memory address. With ~400x400 pixels of video data, this will use 80,000 addresses in the ZBT memory chip, which has a total of 512K addresses (~16% usage). However, since we will be storing a reference frame along with the current frame, this will use a total of 160,000 addresses.

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. write_addr - The address to write in ZBT

2. write_data - The data to write in ZBT (2 pixels of RGB data)
3. write_enable - Controls write access to the ZBT

Testing Strategy:
This module can be tested in simulation by supplying some fake NTSC data and making sure that it outputs the correct memory addresses and combined data for each pixel that is input.

image_processing

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 5x5 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, which should offer sufficient time. It will take on the order of 160,000 cycles to process the images (there are 400*400 pixels). Rendering the VGA frame takes ~600,000 cycles which should provide plenty of time for the processing to take place in parallel.

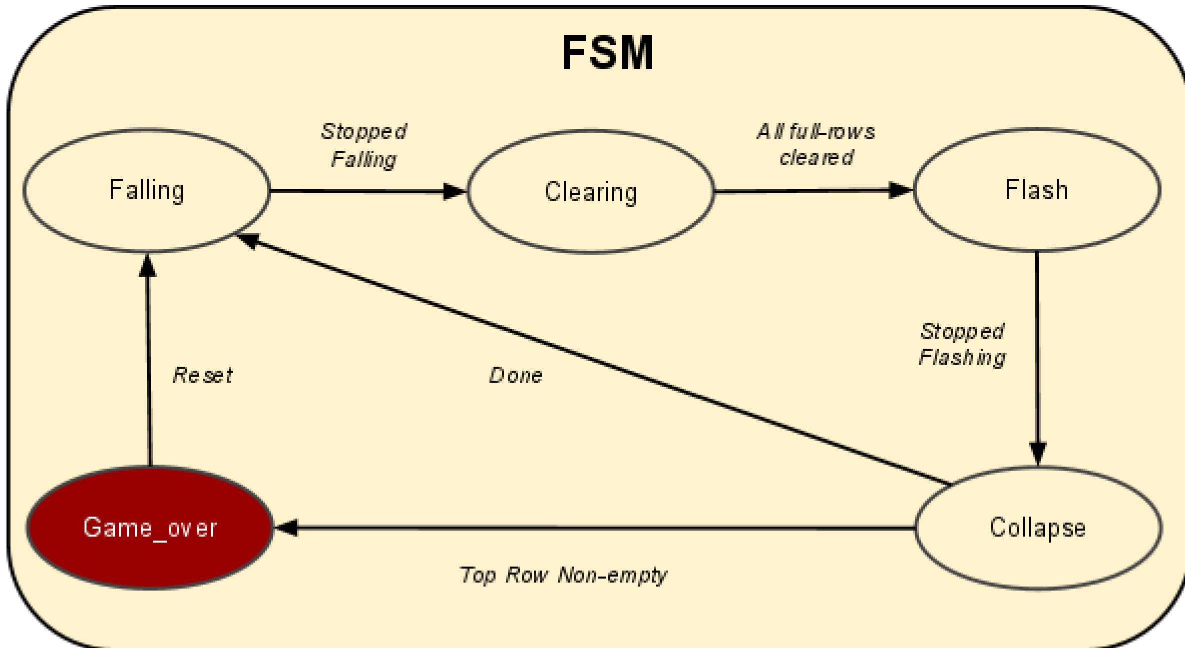
Inputs:
1. read_data - The data being read from ZBT

Outputs:
1. read_addr - The address to read a pixel from the ZBT memory
2. cur_block_shape - The 25-bit representation of the calculated 5x5 Tetris piece

Testing Strategy:
This module will need to be tested on the FPGA due to the complexity of the signals involved. To debug and verify correct operation, it is possible to use the second ZBT on the labkit in order to store the calculated silhouette and output this to the VGA display.

FSM

The FSM handles the general game logic and keeps track fo 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 clear the contiguous rows, then go through a flashing animation in *Flash* state as well as Collapse the blocks in the *Collapse* state to hide the cleared rows. If nothing can be cleared and the last fallen piece hits the top of the game, then the game stops in *Game_over* state or else it starts another round of falling pieces in the Falling state.



States:

1. *Falling* - piece is falling down
2. *Clearing* - after falling piece lands, this marks full rows to be cleared
3. *Flash* - flashing animation for rows marked to be cleared
4. *Collapse* - collapse rows marked as cleared and calculate score
5. *Game_over* - after grid piles up to the top, freeze animation and show score.

Inputs:

1. Shape (5x5 grid) of player's quantized silhouette (Image-Processing)
2. Left/right button presses
3. Game State (BRAM) - location of blocks

Outputs:

1. Score (to display)
2. Game State (BRAM) - updated location of blocks

Testing strategy:

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

Display

The display pieces together the game state in the BRAM, decorations from the ROM, the silhouette from the SRAM, the score from the FSM, and the calculated quantized block shape into the pixels outputted to the player in the monitor.

Inputs:

1. Game State (BRAM) - render the blocks in grid

2. ROM - render fancy images for styling
3. Score (from FSM) - render score
4. SRAM - render silhouette of player
5. Image-Processing - render the quantized shape of the falling block

Outputs:

1. Monitor

Testing strategy:
We can just read out solid blocks of different colors to represent each input component and output it to monitor.

Possible Extensions

Further work include using a gyroscope to detect the player’s motion to replace the left/right buttons for controlling the block, adding interesting sound effects for rowing-clearing or high score/level ups, as well as potentially supporting multi-player mode.