# Contra

**Rishi Thakkar**

## 1. Introduction

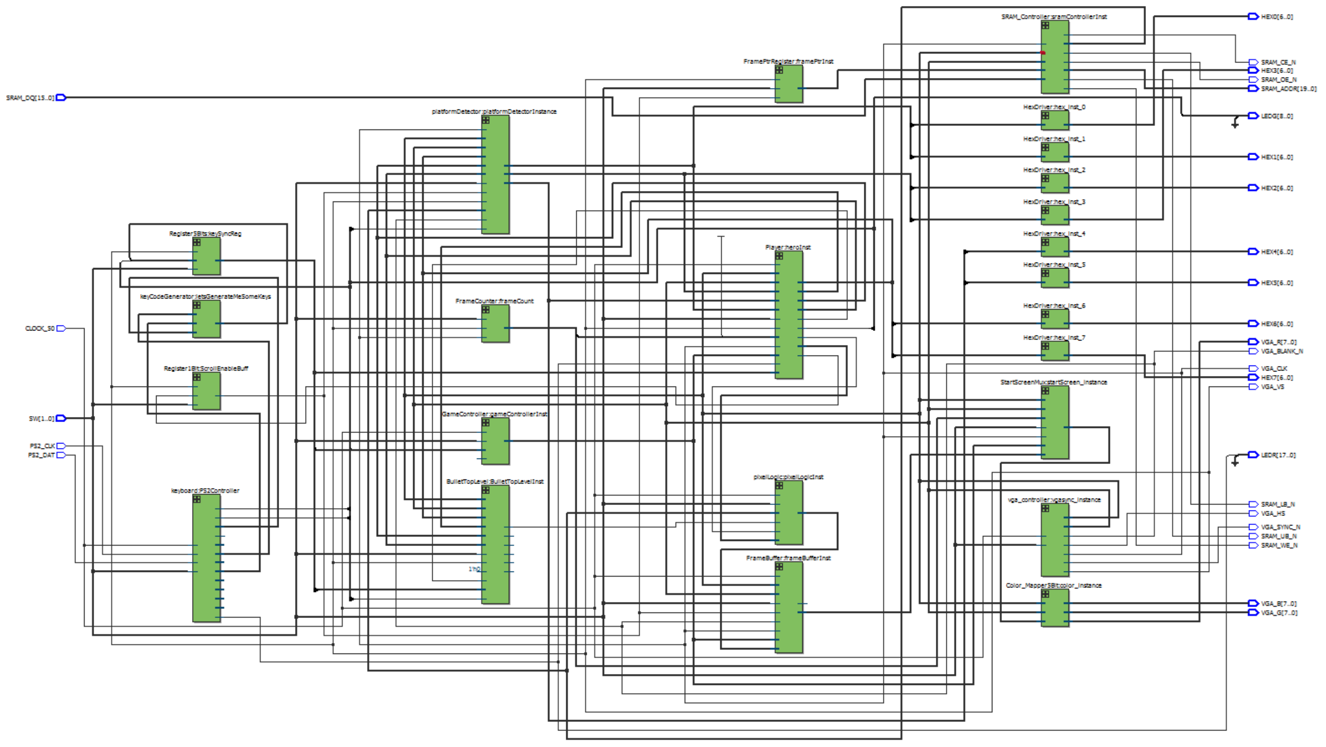
This project incorporates a level of Contra, an old-school NES game, entirely in hardware on the FPGA. Implementation of this game incorporated many features from a frame buffer to an intricate platform detection engine. Through the use of our overall design, we were able to get an entire level of Contra, minus the enemies, up and running on the FPGA.

## 2. Circuit Description

We implemented a old school platformer called Contra completely in hardware. This implementation of the first level of Contra required many different components to work correctly. Since everything was done in hardware, we needed to build comprehensive state machines that dealt with the different states of the game. The main flow of data in our design was controlled by an overarching module called the game controller. The game controller contained a state machine that took into account 3 different states. These states were START, PLAY, and GAME OVER. Based on these three states, the rest of the circuit would rearrange the internal flow of data and make sure that the design was performing the designated tasks for those three states. More specifics on how the game state controller works are provided in section 3. In the START state, access to the frame buffer is shut off. Since our frame buffer is preloaded with the image of the start screen, we do not want to overwrite this image with the game data. Due to this, during the START state the frame buffer just constantly copies itself into itself. Also, during this state, we have enabled blinking of the pixels in the bottom left portion of the screen. This emulates simple animation for the text blinking on the screen. More information about the start screen animation is provided in section 11. The last important thing that occurs in the START state is the fact that all of the PLAY state components are initialized to their starting values. The player is placed on the first platform, all of the pointers are reset, and the actual game is set up. Once a key is pressed, the game state controller transitions over to the PLAY state. In this state, the flow of data transitions from the blocking to the accepting state. What this means is that the frame buffer is enabled and the game data is allowed to be written into it. We use a pixel logic module to choose which pixel to put in which memory location. More information on how this happens is provided in the sections below. The overall flow of data once the game was in the PLAY state was dictated by the user. Based on the inputs from the user through the use of the PS/2 keyboard, we were able to send signals to the player module to create changes in the player position and animations. These changes were then visualized through the use of the frame buffer. The frame buffer was then connected to the color mapper which then put the image we were trying to visualize onto the screen. The basic concept was that when the user pressed a key, a series of tasks were executed and then the change specified by the user was visualized onto the screen. This defined the general flow of data in our design.

## 3. Project Top Level Module

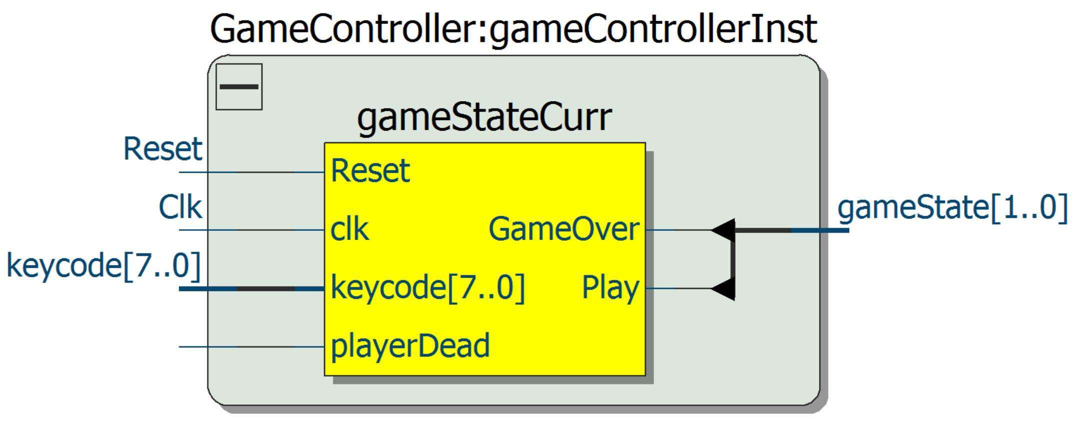
The top level of this project basically served as a way to connect all of the modularly designed components of this project to each other. It allowed us to organize our thoughts as designers and helped us make sure that no mistakes were being made. The inputs of this module were the Clk, PS2_CLK, PS2_DAT, KEY, SW, and the SRAM_DQ. The outputs from this module were all of the HEX displays, the green and red LEDS, the VGA control signals, and the SRAM control signals. These inputs and outputs allowed our design to communicate with external hardware and thus visual and play an entire level on contra. The image below shows the schematic block diagram of the project's top level module.



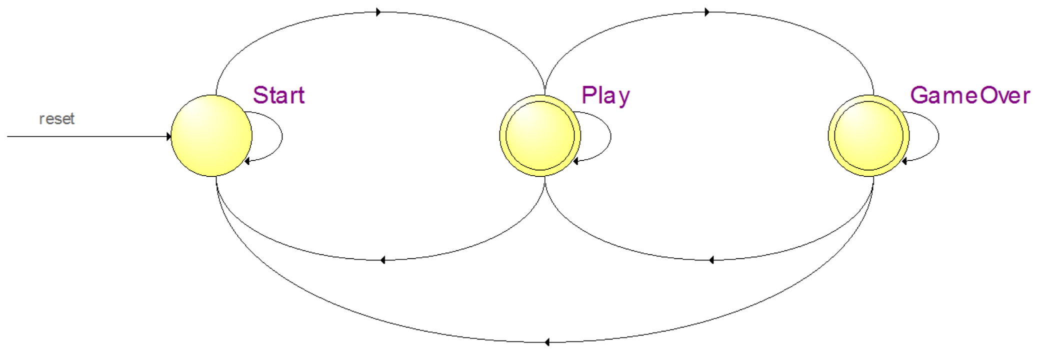
## 4. Game Controller Module

The game controller is the control unit for our entire game. It is a very simple module, but it plays a very important role in setting up the entire game and making sure that the rest of the state machines get initialized to the correct values. The inputs into this module are Reset, Clk, playerDead, and keycode. The output from this module is the state of the internal state machine. Through its states, it basically controls the data flow of the entire circuit. There are three states through which it transitions. These states are Start, Play, and Game Over. If you are in the Start state, all the components related to the game are turned off. During this state, the circuit is instructed to output the title screen to the display with blinking text. If any key is pressed, then we transition to the Play state. In this state, the game has started and all of the other state machines related to the game are initialized as such. When the player dies, the state machine transitions to the Game Over state. In this state, there is a end screen which is displayed to the monitor. If the user presses any key, then the player is respawned and the game starts again. The block diagram and next state diagram of this circuit are presented in the sections below.

### 4.1 Schematic Block Diagram

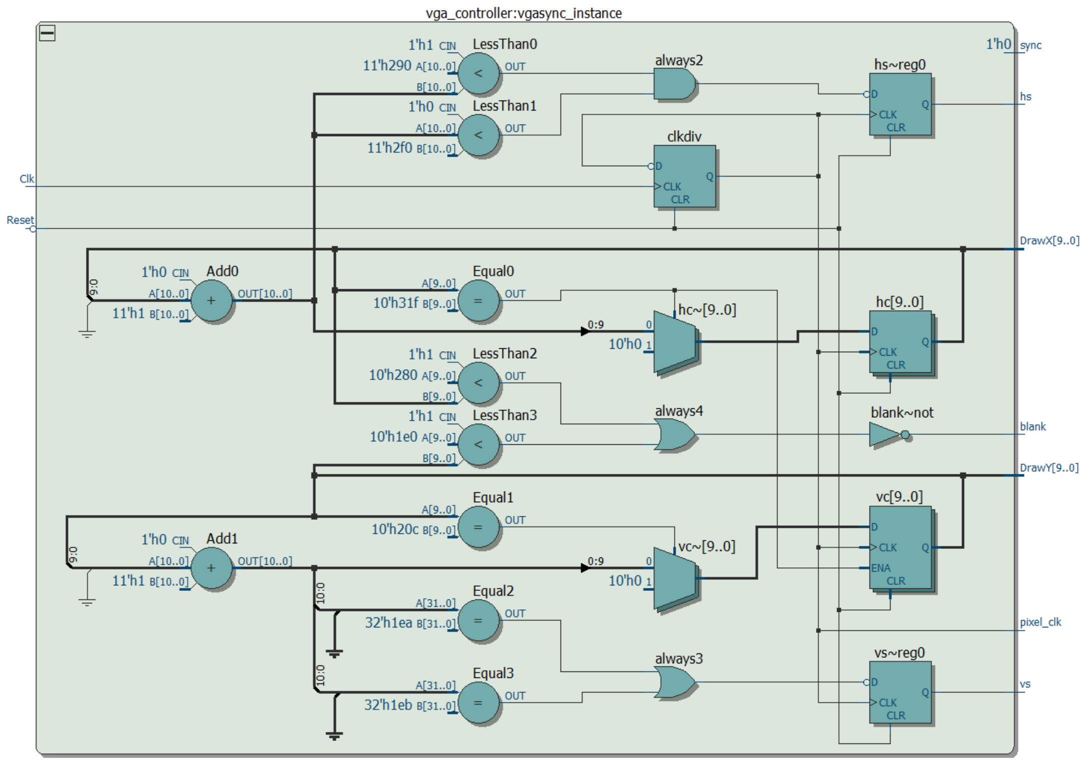


### 4.2 Next State Diagram



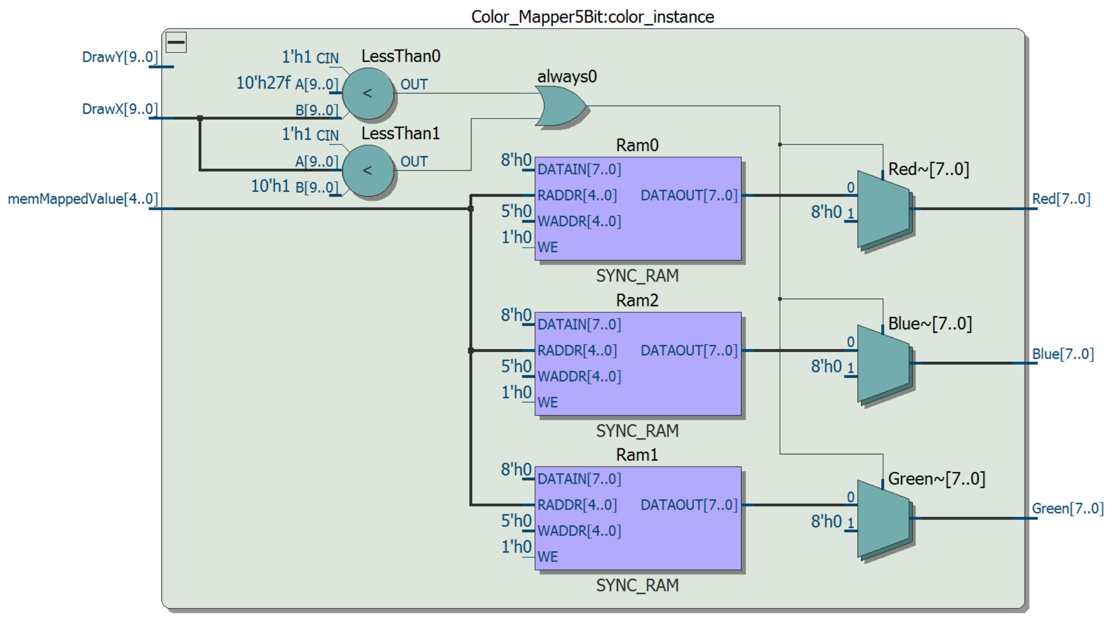
## 5. VGA Controller Module

The VGA controller was used to setup the control signals for the display to which our design was outputting. The inputs into this module were clock and reset. The outputs were DrawX, DrawY, blank, hs, vs, pixel_clk, and sync. The DrawX and DrawY signals were connected to the all of the modules related to sprites and background production. They gave these modules a reference to the position at which the electron beam was drawing. The vs and blank signals were used by many of the modules to help make sure that the drawing, motion, and collision detection were occurring synchronously with the refresh rate of the monitor. The rest of the output signals were directly connected to the display. The hs or horizontal sync was a short pulse at the end of a row of transmissions that kept the horizontal scanning of the screen exactly in sync with the transmission of the next row. This basically means that the horizontal sync was telling to screen that it needs to transition to outputting to the next line of pixels so that it doesn't miss any pixels. The vs or vertical sync was also a short pulse except it was transmitted at end of each frame transmission instead of at the end of each row transmission. The purpose of the vertical sync was to help retain the frame by frame synchronization of the outputs. This basically means that the vertical sync told the screen to transition to outputting the first row and first column when it had reached the end of the last row. To make sure that the electron beam is not corrupting the image during these transition periods, a blank signal is used by the VGA controller to tell the electron beam to output nothing. This short period of time is known as the blanking period. Both the horizontal sync and the vertical sync are very important to the integrity of the image. The pixel_clk was used to set the refresh rate of the screen. In the case of our design, we decided to set our pixel_clk to 25 MHz. This allowed us to work at a rate of 60 frames per second and was fast enough such that the user would see any changes to be seamless. The last signal we outputted was the sync signal. This signal was disabled because it is not relevant to outputting our image. The image below shows the schematic block diagram of the VGA controller module.



## 6. 5-Bit Color Mapper Module

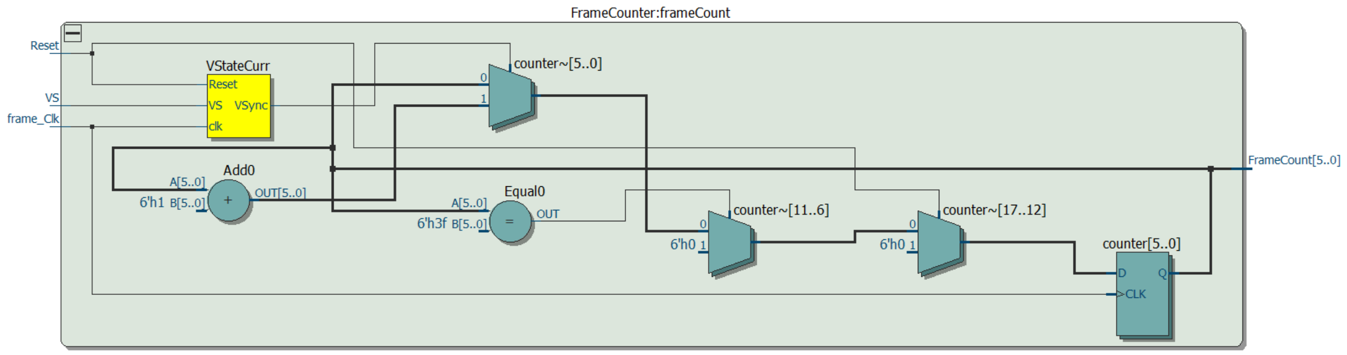
The color mapper played a very important role in our design. Through its use, we were able to more efficiently use memory and make sure that there was enough room to store all of our sprites and background images. Our original images had 32 bit color depth. With this depth, it would have been impossible to fit most of the things that we needed in the memory we have allocated on the board. To fix this issue, we decided to compress our images through the use of a color palette. We realized that we only needed 21 total colors in our game and so we only truly needed a 5-bit value to describe the color of each pixel. We included a 22nd color which accounted for transparency and allowed us to display images that had partial transparency in them. Though the use of a python script that we found, we were able to compress each image and convert it to a format that was recognizable by the memory initializer. The python script basically traversed through each pixel and mapped each pixel's RGB values to its closest match in our predefined palette. If the transparency level for any pixel was less than 200, then that pixel was automatically mapped to our transparency color. The color mapper basically served as a decoder for this color palette. The input into this module was a 5-bit pixel value. The color mapper took this value and used an internal lookup table to map this value to its corresponding RGB value. These RGB values were then exported from this module and were directly connected to the display. Through this setup, were able to visualize all the graphics that corresponded to our game. The image below shows the block diagram of the color mapper module.



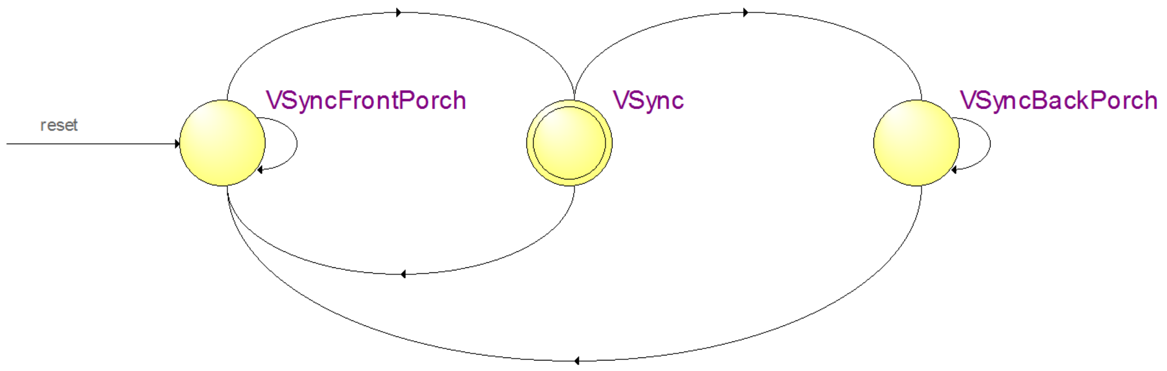
## 7. Frame Counter Module

The frame counter module served a very general purpose in the design of our project. It served the role of counting the number of frames that had been displayed to the screen at any given time. This played a very important role in our design because many of the state machines that we designed were based on the this count. The animation engine used this count to keep track of the number of frames that needed to pass before switching to a new image. The start screen mux module used this count to create blinking text on the screen. Basically, without this module, many of our core design components would not have worked. The inputs into this module were Reset, VS, and frame_Clk. The output from this module was the FrameCount. Through the use of a simple state machine, the frame counter module incremented the value of a count register every time a VS was seen. This allowed us to count the number of times a vertical sync has been seen and thus the number of frames that have been outputted at any given time. The sections below show the block diagram and next state diagrams for the frame counter module.

### 7.1 Schematic Block Diagram



### 7.2 Next State Diagram



## 8. Background Image Display

The SRAM was used in our design to store the entire background of our game. This consisted of a very large image that was 4114 pixels in width and 480 pixels in height. This means that there were a total of 1974720 pixels that we need to account for. This is a clear problem since there are only 1 million address spaces that can be accounted for in the SRAM. While looking for a way to compress our image to fit into this memory space, we discovered a very interesting feature of some of these old school NES game. Since the NES was not that powerful, the graphics of some of these games had to be optimized so that limitations in the NES design would not be noticed during run time. One of these optimizations was to only store the address of every other pixel in memory. What this means is that every pixel would be output to the screen twice. Since the graphics of the NES are already fairly pixelated, this type of optimization would not affect visual experience of the user in anyway. Based on this, we were able to reduce our total needed memory size to exactly 987360 address spaces and thus fit everything on the SRAM. To get this image into the SRAM, we used the same method as we did for the sprites. We first used the python script to map each pixel value to our palette. Then we took the the outputted file and used a C program to convert this .txt file to a .ram file. The C program used a well known function called fwrite() to convert our file to a file that would be recognizable by the DE2-115 control panel. Once the image had been flashed onto to the DE2-115 board, it was ready to be used by the rest of our design.

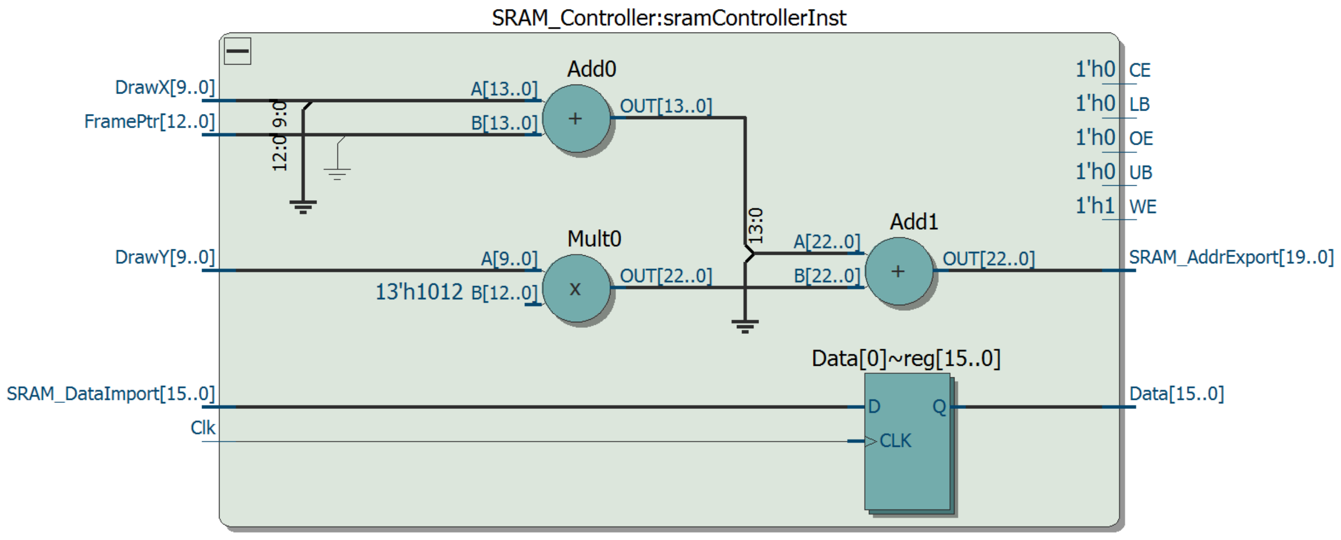
## 9. SRAM Controller Module

Since the background of our entire level was stored in the SRAM, an SRAM controller was integral to the functionality of our design. The SRAM controller took in inputs of Clk, DrawX, DrawY, FramePtr, and SRAM_DataImport. The outputs were SRAM_AddrExport, Data, CE, UB, LB, OE, and WE. A combination of these control signals allowed us to access memory as we pleased and grab the pixel data for any pixel. Since we were essentially treating this memory as a ROM, the control signals for the SRAM were hardwired to specific values. We pulled WE high and OE low so that the SRAM will always be outputting data. The addresses from which the pixel data was being read was calculated using the mathematically formula:

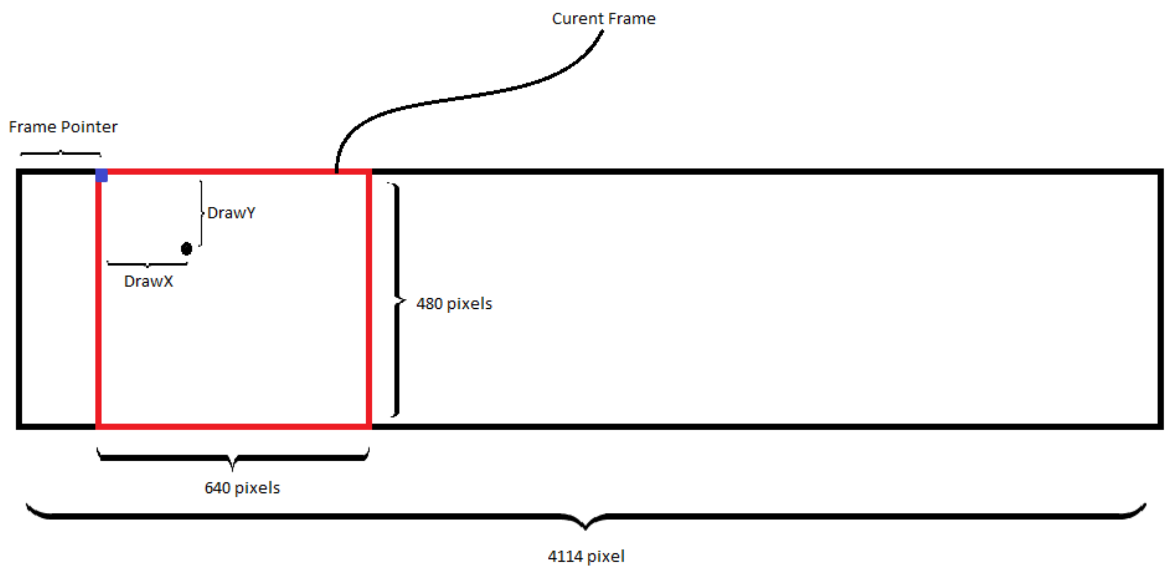
$$SRAM\_AddrExport = (DrawX + FramePtr) + (DrawY * 4114)$$

DrawX is the X position on the screen that we want to draw to, FramePtr points to the current frame that we have scrolled to, DrawY is the Y position on the screen that we want to draw to, and 4114 is the pixel width of the entire level. Since the image has been stored in row-major order, this formula allows us to access any pixel in any given frame. More information on how the FramePtr is generated is provided in section 10. The image in the section below gives a visual representation of this formula and how it is accessing the different memory locations.

### 9.1 Schematic Block Diagram

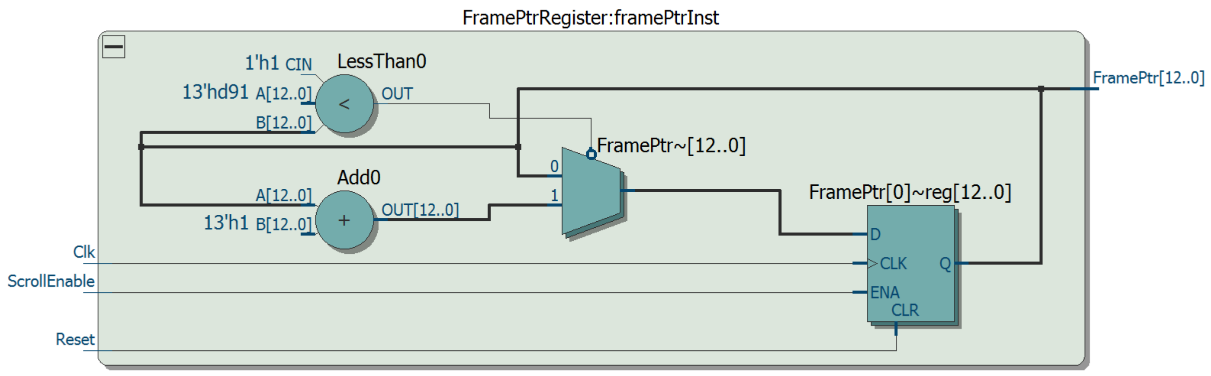


### 9.2 Visual Representation of SRAM Access Formula



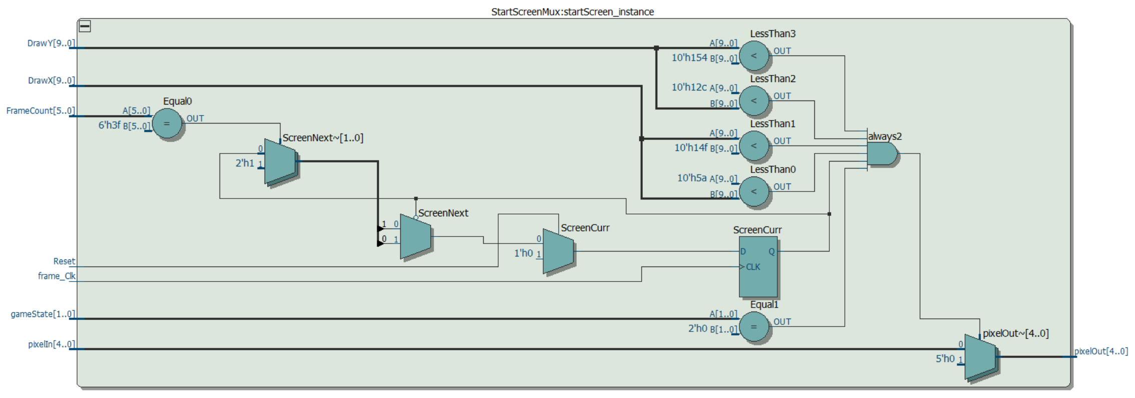
## 10. Frame Pointer Register Module

The frame pointer register served the purpose of keeping track of which frame the game is currently on. The inputs into this module were Clk, Reset, and ScrollEnable. The output from this module was the FramePtr value. Basically, what this module did is it took the current frame pointer value and incremented it at the positive edge of the clock if scroll enable was high. Scroll enable was determined using a very simple checking mechanism. If the player is at the center of the screen and the right movement key is being pressed, then scroll enable is high. We used the VGA_VS to clock this module. This is because we did not want the frame pointer to be changing during a display cycle of the monitor. This could cause some real problems with the display because if half the screen has been drawn and the frame pointer suddenly changes, then the second half of the screen would be using a different frame pointer to access the SRAM. This would causes distortions in the image being displayed and thus be very bad for our project. The image below below shows the schematic block diagram for the frame pointer register module.



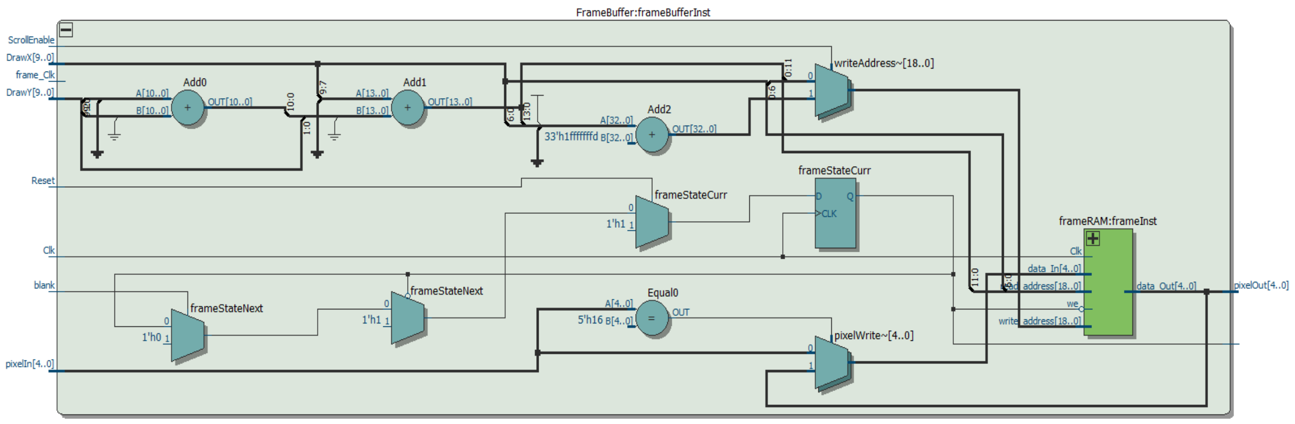
## 11. Start Screen Mux Module

The start screen mux module played an important role during the Start state of our game. This state is defined by the Game Controller module. For more information on the Game Controller visit section 4. The inputs into the start screen mux are DrawX, DrawY, FrameCount, Reset, frame_Clk, gameState, and pixelIn. The output from this module is the pixelOut value. The start screen mux is basically used to create the text blinking animation on the start screen. It uses the gameState input as an enable bit. If the current game state is Start, then modify the pixelIn value such that the blinking occurs. Otherwise, just the the pixelIn value and relay it directly to the output. The modification of the input pixel is done through the use of a simple mathematical formula. If the DrawX is greater than 90 but less than 335 and DrawY is greater than 300 but less than 340, then modify the input pixel. The decision on how to modify the screen was done by checking how many frames have passed at a given time. This is where the frame count input came in handy. Using the count, this module alternated between outputting black and the pixel input value. This gave the effect of blinking the text every 1 second. Though this was a very simple animation, it added to the overall aesthetic of the game and user experience. The image below shows the schematic block diagram of this module.



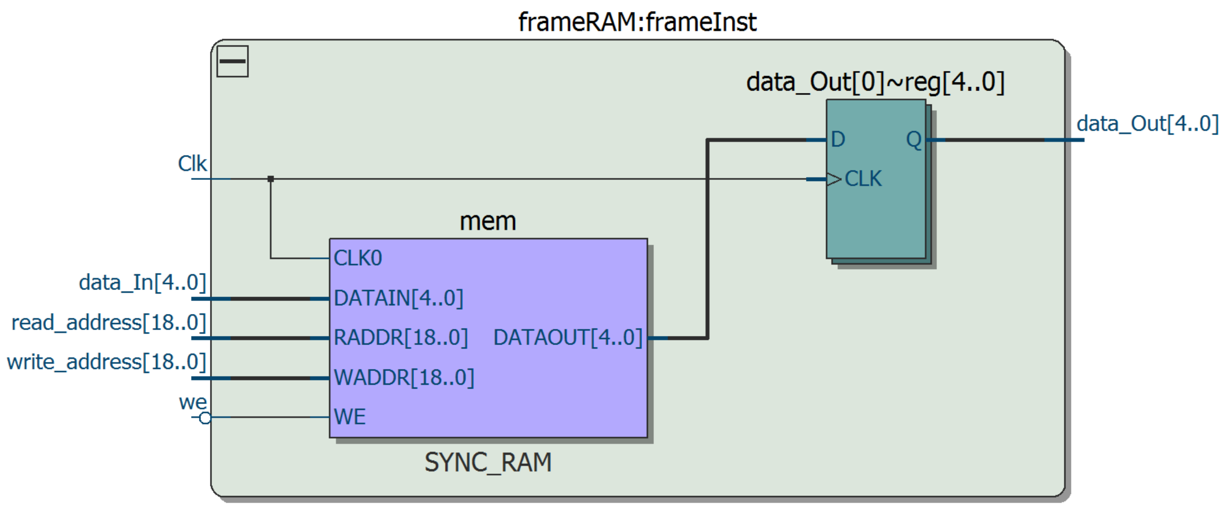
## 12. Single Frame Buffer Controller Module

A frame buffer is a specific portion of memory that is used to store frame data. It allows us to read and write frame data from it and through this mechanism it separates the pixel generation and pixel output portions of our design. Its properties allow for easy scrolling and and image manipulation and due to this we decided to implement a frame buffer to store the values our display. We looked into using a single frame and a dual frame buffer and we came to the conclusion that a single frame buffer would be the best choice. It saves a lot of memory and helps us store everything that we need in on-chip memory. The inputs into this module were Clk, DrawX, DrawY, Reset, ScrollEnable, blank, and pixelIn. The output from this module is a 5 bit pixel value that signifies the color for any given pixel in frame buffer. This module served as a top level for our frame buffer. It set up all of the control signals for the frame RAM and helped select the appropriate read and write addresses, and pixel values. The frame buffer itself was implemented using on chip memory and was a submodule to the Frame Buffer Controller module. More information about the frame buffer module is provided in the section below. The implementation of the frame buffer controller was done through a simple state machine. The whole system revolved around two states. There was a Read/Write state and a Wait state. During the Read/Write state, the current value stored in the frame buffer was read and the pixel value of the next frame was written. Since the memory has an external register that stores the read value, writing into the same address we were reading from did not corrupt the value that was being read. During the Wait state, the value that was read during the last clock cycle is continued to be outputted. Nothing else is done during this state. This state is needed because of two reasons. The first reason is that the state machine is clocked at 50MHz while the output to the display is clocked at 25MHz. Since we do not want to skip random pixels and corrupt the data that is stored in the frame buffer, we need to add a wait state to account for this difference in clock speeds. The second reason is that it allows us to check for the blanking portions of the screen. Since we do not want to be reading or writing to the the frame buffer during the blanking state, the Wait state allows us to wait until the blanking period is over to continue to read and write from the frame buffer. This is also another way that we can make sure that we are not corrupting our frame data. The pixel read and write addresses were assigned in this module and then exported to the frame buffer. They were decided through the use of combinational logic with inputs of DrawX, DrawY, and ScrollEnable. Since the frame buffer was organized in row major order, the read address was consistently assigned the value of DrawX + DrawY*640. The write address was decided through the use of the ScrollEnable signal. If ScrollEnable was hgihg, then the write address was DrawX + DrawY* 640 -1. The minus 1 allowed for scrolling since it shifted all of the values stored in memory over by 1. If ScrollEnable was low, then the write address was equivalent to the read address. Through the use of this simple system, we were able to output to the screen the buffered pixel values in the frame buffer. The schematic block diagram of this module is shown below.



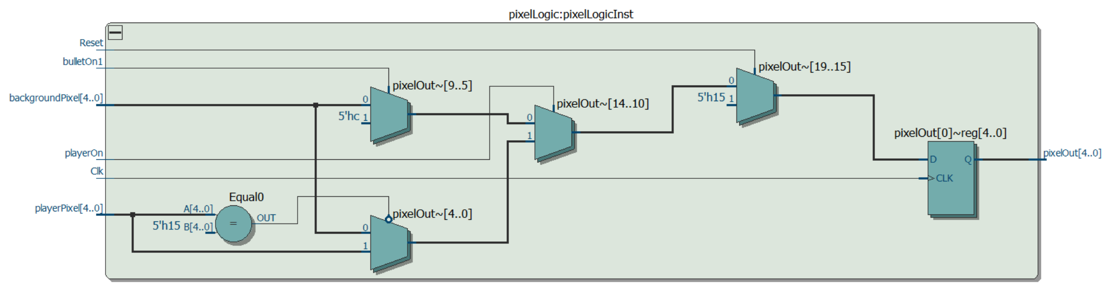
## 13. Single Frame Buffer Module

The frame buffer itself was implemented through the use of on-chip memory. The inputs into this module were Clk, data_In, read_address, write_address, and we. The output from this module was a 5 bit data_Out value. This block of memory allowed us to read and write our frame data and thus allowed us to visualize our entire game. The schematic block diagram of this module is shown in the image below.



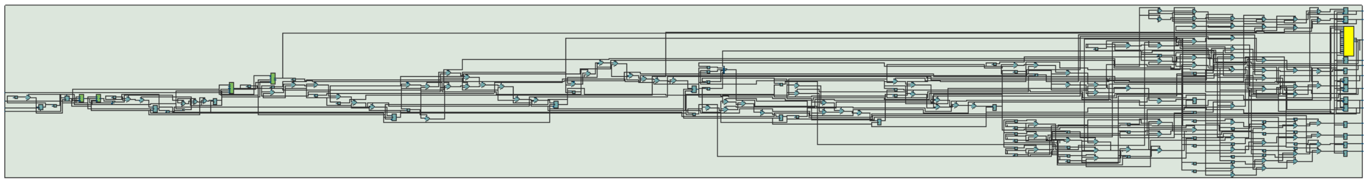
## 14. Pixel Logic Module

The purpose of the pixel logic was to control what was being written into the frame buffer. Due to this, the pixel logic module played a very important role in what was being visualized on the screen. The inputs into this module were Clk, Reset, playerOn, bulletOn1, bulletOn2, bulletOn3, bulletOn4, bulletOn5, bulletOn6, playerPixel, and backgroundPixel. The output from this module was 5 bit pixelOut value. Since the drawing sprites is performed in the order o player, bullet, then background, this module first checks if the playerOn signal is high. If so, then the module does a secondary check to make sure that the input playerPixel value is not transparent. If it is transparent, then the background pixel automatically gets placed onto the frame buffer. If it is not transparent, then the playerPixel value gets placed onto the frame buffer. If playerOn is low, then the module checks to see if bulletOn1 is high. If so, then the bullet gets drawn to the screen as a circle. This same check is done for all of the other bullets and they are drawn to the screen in the same way. If playerOn and all of the bulletOns are low, then the background is placed onto the screen by default. Through the use of this module, we were able to control the flow of data going into the frame buffer and thus visualize the game to the display perfectly. The image below shows the schematic block diagram of the pixel logic module.



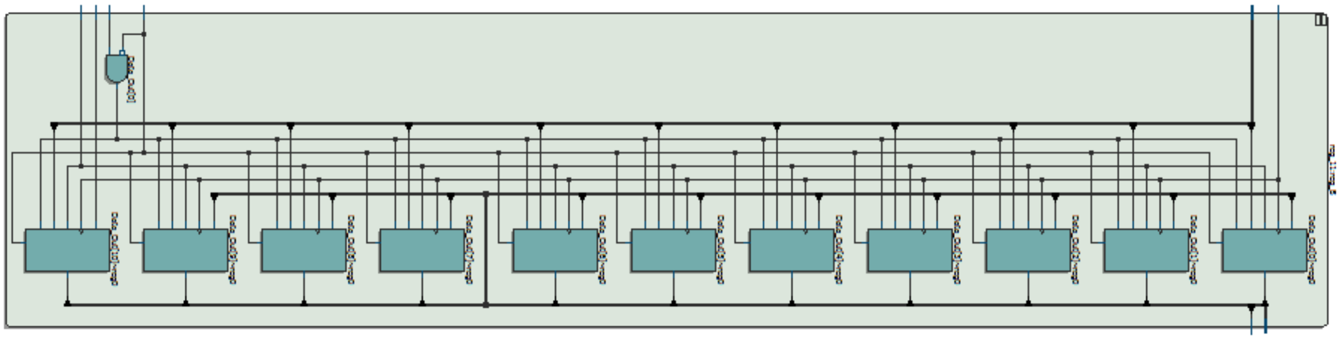
## 15. PS/2 Keyboard Module

Proper user input management is a very important for user experience in a game. Due to this we had to pick the correct method for getting user input. After extensive research, we came to the conclusion that using the PS/2 would be the best choice for our project. One advantage of using a PS/2 over a USB interface is that the PS/2 interface is a lot faster than the USB interface. The reason for this is that the PS/2 interface is interrupt based hardware while USB interface is polling based hardware. What this means in terms of keyboards is that a PS/2 keyboard will generate an interrupt signal immediately when a key is pressed. On the other hand, a USB keyboard will be constantly polling to see if a key is pressed or not. This can cause delays and result is a much slower rate of data transfer. Another positive of using a PS/2 keyboard is that it supports an n key rollover. What this means is that PS/2 keyboard can detect n keys being pressed at once. This features was very important for our game because we needed to detect multiple key presses at once. Our player had the ability to have motion in all four directions and at the same time jump and shoot asynchronously. To account for these inputs along with the priorities associated with them, we used a system of flags and a pseudo linked list. This was needed because the inputs from the user could come in any order. For example, the user could press ‘W’, ‘S’, and then ‘A’. In this type of a situation, it's important to assign priorities to the inputs and then use them according  to these priorities. In the example shown above, ‘W’ would be the head of our linked list. ‘S’ would be connected to ‘W’ and then ‘A’ would be connected to ‘S’. Our code would see that ‘W’ and ‘S’ were the keys that were entered in initially and thus respond to them accordingly. Let's say that the user then decides to release the ‘S’ key but is still holding down ‘W’ and ‘A’. The priorities are then reassigned. The ‘S’ node gets deleted and ‘A’ gets connected to ‘W’. This allows the code to see that ‘W’ and ‘A’ are being entered and then react to this accordingly. While the ‘W’, ‘A’, ‘S’, and ‘D’ inputs have a priority associated with them, the jumping and shooting inputs are accounted for separately using flags. This was done because both of these inputs take priority over all other inputs no matter when they are pressed. Due to this property, they had to be accounted to separately. We received the initial skeleton code for the PS/2 driver from the course website. However, we had to make major changes to get the driver to function the way we wanted it to. The inputs into our driver module were Clk, psClk, psData, and reset. The outputs from this module were key1, key2, key3, key4, keyCount, press, Shooting, and Jumping. At the lowest level, the PS/2 keyboard interface is a serial interface. Due to this, it only has one data line, psData, to perform its communication. It sends packets of 11 bits to perform communication. The packet consists of a start bit that is always 0, the 8-bit scancode, an odd parity bit, and a stop bit that is always 1. Through the use of these packets, the keyboard transfers make and break codes to signify which keys have been pressed and which keys have been released. Since we needed to support at most 6 keys, we had key1, key2, key3, key4, Shooting, and Jumping registers to store these keys in the way that we mentioned above. KeyCount and press were used as bookkeeping information which helped us keep track of how many keys are pressed at any given time and if any keys are even being pressed. Scancodes were internally stored through the use of submodules which contained 11 bit and single bit registers. These modules are discussed in the sections below. Through the use of the PS/2 keyboard and this module, we were able to set up proper user input management and thus enhance the user experience. The image below shows the schematic block diagram of the PS/2 driver module. Due to its complexity, the image isn't really understandable.



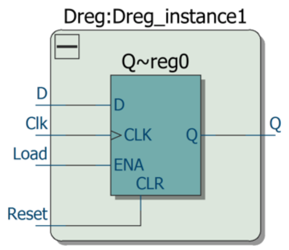
## 15.1 11-Bit Register Module

The 11-Bit Register module was used as a submodule within the PS/2 keyboard driver module. It served the purpose of temporarily storing the make and break packets as they were being transferred from the PS/2 keyboard. Since the data was being transferred through the use of a serial interface, the 11-Bit registers were set up to shift in the values as they were being inputted. This made this module an essential part of our PS/2 driver design. The inputs into this module were Clk, Reset, Shift_in, Load, Shift_En, and D. The outputs from this module were Shift_Out and Data_Out. The image below shows the schematic block diagram of the 11-Bit register module.



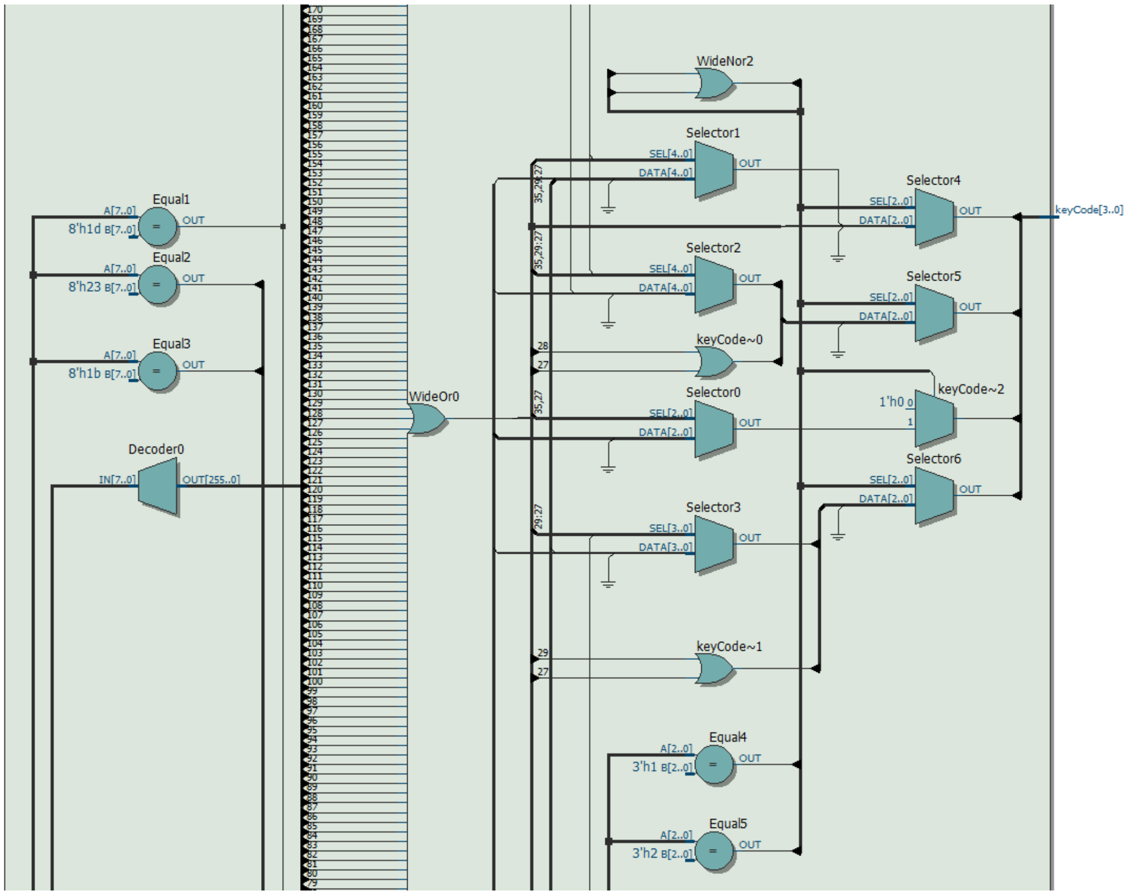
## 15.2 D-Register Module

The D-Register module was also used as a submodule within the PS/2 keyboard driver module. It server the purpose of detecting the edges of the PS/2. To make sure that any disturbance in the PS/2 clock do not affect our design, these registers behaved as modules to buffer the clock and make our design fully synchronous. The inputs into this module were Clk, Load, Reset, and D. The output from this module was Q. The module stored a single bit and made it available for the PS/2 driver to use when it needed to. The schematic block diagram of this module is shown in the image below.



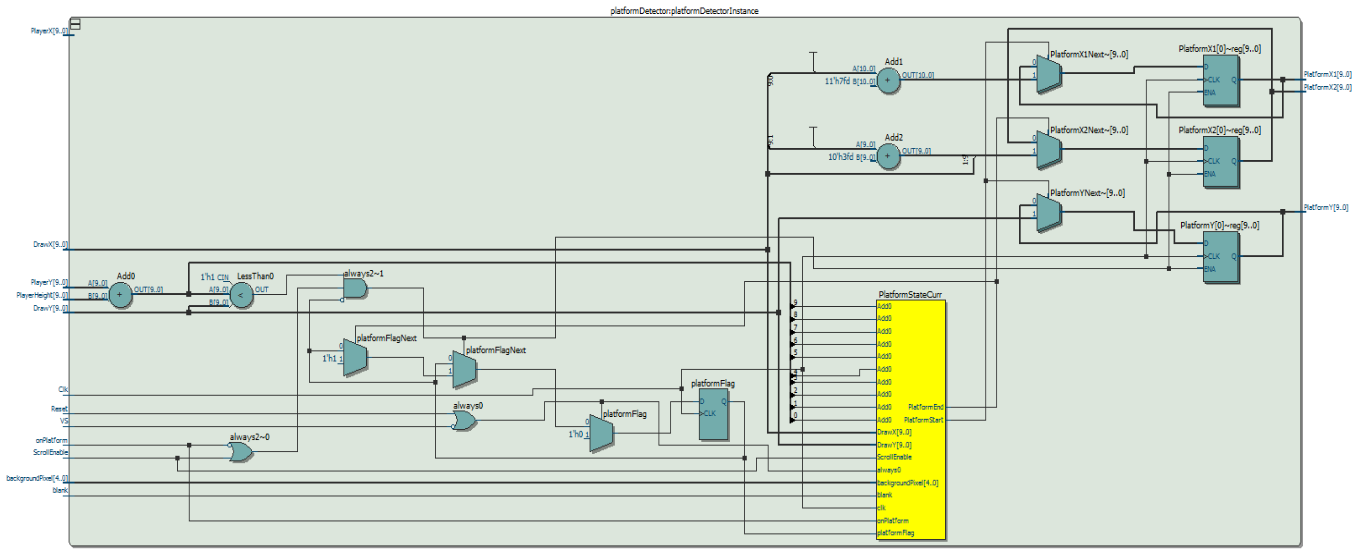
## 16. Keycode Generator Module

The keycode generator module was used to simplify the different possibilities of key presses to ensure that all of the movements of the player, excluding shooting and jumping, would change smoothly. The inputs into this module were the numbers of keys that were being pressed by the user and the two keycode registers that were prioritized in the PS2 Keyboard module. The output is a mapped KeyCode value. The way that the module accomplishes the mapping is to use the number of keys pressed and check the registers in the order of priority. For example, if the user presses 'W' and 'A', this mapped key code value is determined by first checking that 2 keys are pressed and then a case statement that checks the first keycode register value for a W and another case statement that checks the second keycode register for A. By utilizing these mapped behaviors, it became easier to simply determine what the user wants for every keypress. Then this value simply went to a register where it became the outputted value to the Player modules. It was clocked at VGA Vertical Sync to ensure that there were no instances where portions of the player's animations and output pixels were continuously being changed but rather the entire player pixels and aminations were synchronously outputted at each frame. This made sure that the motions were fluid and without any errors. The image below shows the schematic block diagram of this module.



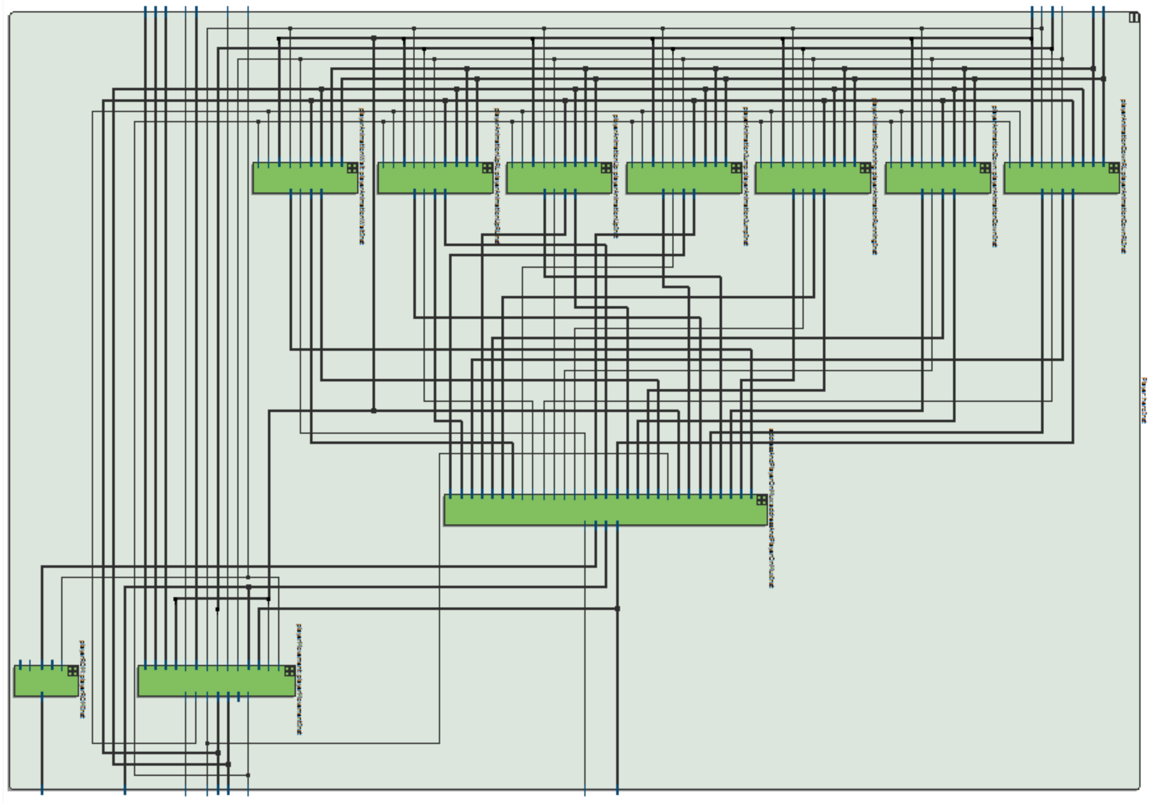
## 17. Platform Detector Module

Since Contra is a classic definition of a platformer game, the platform detector module played a very important role in the functionality of our circuit. The interesting thing about this module is that it performed platform detection in real time. What this means is that the platforms were not hardcoded into the system. Instead, the platform detector module would detect platforms for each frame individually. This was done through system of image manipulation and recognition. As the programmers, we had to take the image of the background and add a random color to mark all of the platforms. We used yellow as our color of choice since it did not appear anywhere else in our palette. In the image above you can see how each of these platforms was colored. The purpose of this was so that the platform detector module would be able to mark the DrawX positions of when it started to see yellow as the starting position of the platform and mark the position when stopped seeing yellow as the end position of the platform. It would also save the DrawY position for the height at which it saw the color yellow. This bassically allowed us to detect the platforms in real time and create a reusable system for platform detection. Through the use of an internal state machine, the module was restricted to finding one platform per frame and it was only allowed to look for platforms underneath the player's y-position. This allowed us to limit our bounds to a single y-bound and 2 x-bounds for any given frame. The inputs into this module were Clk, Reset, blank, VS, onPlatform, ScrollEnable, DrawX, DrawY, PlayerX, PlayerY, PlayerHeight, and the backgroundPixel corresponding with the DrawX and DrawY. The outputs from this module were PlatformX1, PlatformX2, and PlatformY. This module basically detected the platform and then exported the starting and ending X position and Y position of the platform. This was done so that all of the other modules could then modularly use these inputs to define the bounds of the player. The image below shows the schematic block diagram of this module.



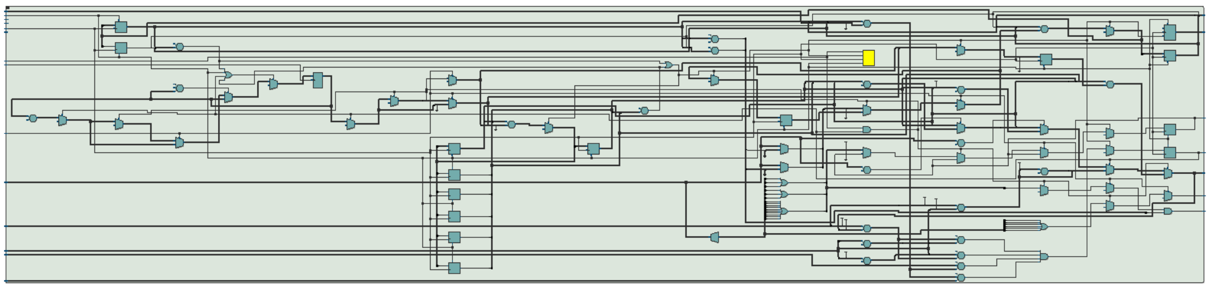
## 18. Player Module

The Player Module was the top level module that determined every single characteristics of the player sprite. The submodules described within the top level determined how the player moved with respect to the platforms and the key presses, as well as the addresses that access the different memory locations of the players movements sprites. The inputs include Reset, Frame Clk, Clk, VGA Vertical Sync, the mapped Key Code, Key Press (checks if any keys are pressed), Game State, DrawX, DrawY, the Frame Counter, and the start and end positions of each platform with the height of each platform. The output is the Pixels of the Player, the Player X and Y positions, Player Height and Width, the player on, of the player is on a platform, the scroll enable, and the direction of the player. The specifics of how each of these inputs are used and how the outputs are determined are described in the submodules below. The image below shows the schematic block diagram of this top level player module.



### 18.1 Player Movement

The Player Movement Module is the core of the entire Player. It chooses how the player will react to the platforms, how the player will react to different keycodes, and what the players position. The inputs to this module are the clk, reset, frame clk, Vga vertical sync, the frameCounter[3], jumping, keycodes, keyPressed, the current game state, and the platform start, end, height as well as the players height and width. The output of this module is the Player X and Player Y, the direction of the Player, if the player is on a platform, whether the player is moving or whether scroll should be enabled. Because we tested everything modularly, we found it best to create parameters that could be changed so that each constant could be represented and understood easily. First we have the On Platform signal which has this formula where Player's X positions + half player width is between the platform start and end position and that the platform height - Player's Y position is between the height ± 8 pixels to ensure that there is enough time for the Player sprite to stop and land on a platform.  Next is the scroll enable which is initiated if the Player X position is more than Player Scrolling Max and that the player is moving right. Then we created a gravity function after trial and error to ensure some sort of movement of the Player that increments every 8 frames and moves the player down depending if the player is on platform or not as well as if if the Player is on the screen or not. The movement of the player is determined if any keys are being pressed or not. If there are no keys being pressed, then we simply want gravity to take over where gravity plays a role if the player is off the platform or else the player does not move. However, if keys are pressed, then we first check if the user is jumping and if so the player needs to jump a certain number of pixels and gravity needs to be negative so that a parabolic shape can be seen. If none of these checks pass, the the module checks the keycodes. For example, if the WA mapped keycode is sent into the module then the next direction is left and the player is moving so that the animations can begin. Then if position is less than the minimum position (prevents player sprite from going off screen) the player sprite stops or else moves of the left.  Then check if the player is on the Platform and if so then the player has no Y motion or else the Y position is determined based on a gravity function that was described before. Lastly, to ensure that the drawing is done correctly we change the final output position of the sprite if the player is jumping and on platform as a sum of its position and motion. Or if the player is simply on the platform, we keep that bottom of the player at the platform height - 2 pixels. Lastly we check if the keycode is 4'h1, representing the up position, then the new Player Y position is simply a height shift up or if the keycode is 4'h3 is the down position a height shifts down. This is used to ensure that the player positions are drawn correctly so that expected player sprite movement is as expected. If none of those cases work, then the output Player Y is the Player Y position with no changes. By using all of these complex checks that consider almost every single action of the player sprite and how the player is going to move, we create a dynamic platform game system that provides fluid user directed movement and proper actions depending on the user inputs and interactions with the obstacles and characteristics of the background itself. The image below shows the schematic of this module. Its barely understandable due to the complexity of this module.

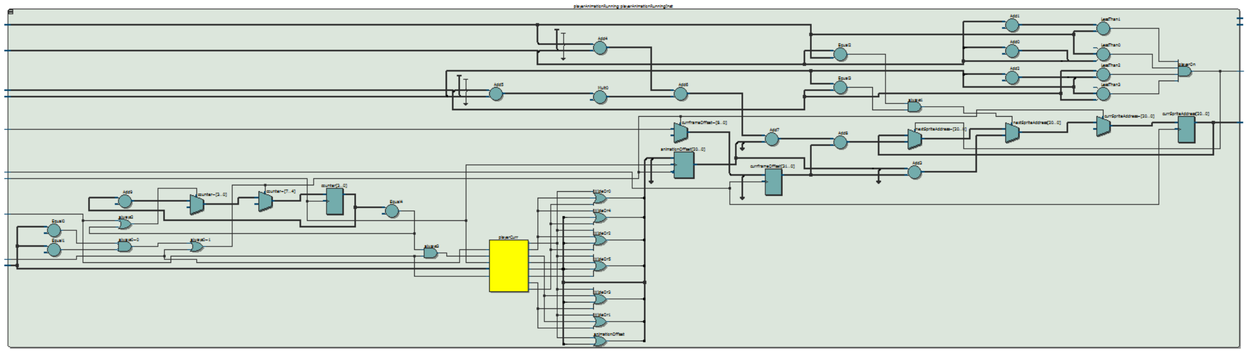


## 18.2 Player Animations

The Player Animations were all repetitions of each other. We created this module to be generalized and expanded to each of the possible movements of the player sprite. The inputs are Reset, if the player is moving, frame clock, player direction, frame counter 0, Draw X and Draw Y, Player X, Player Y, and the keycode. The outputs of these modules are if the player is on (whether the framebuffer should be outputting the players pixels), the sprite address containing the player's current pixels, as well as the player height and width, The way this is generalized system is created begins with parameters that are used to select locations of the sprite addresses. The initial offset chooses where in on chip memory the animation is being stored. The player height and width, which was initially determined by how the images were created, help determine specific memory locations during the animations. Then there is a right offset which is 21'd0 and left offset which is 21'd50620. What these offsets represent is where to find the animation depending on the direction of the player's movement. These offsets are used later on to map the correct address of the player's animation. The next part of this module is to determine when the players pixels need to be sent to the frame buffer so it can be outputted. The modules determines this using this formula: $DrawX \leq PlayerX + playerWidth$ & $DrawX > PlayerX + 1$ & $DrawY < PlayerY + playerHeight$ & $DrawY \geq Player$. This formula ensures that only when the Draw X and Draw Y are within the bounds of the Player, the sprite addresses for the player's pixels are sent to the animation mux so it can be sent to the framebuffer when asked for. Then we have the sprite address and frame offset change every frame clock as long as reset and the keycodes not representing the movement described by the animation are not high. For the sprite address, the next address is determined depending on if the player is on and if so it utilizes this formula: $(DrawX - PlayerX) + (DrawY - PlayerY)*playerWidth + animationOffset + currframeOffset$. What this does is choose the correct pixel and then depending on the current animation state, which is described next, and the direction of the player, the pixel address is sent to the animation mux. The frame offset that is in the formula above helps determine the right or left offset based on player direction. The next part is how the animations are determined. Depending also whether reset and the keycodes representing the movement described by the animation are not high, we also determine the player animation state where the animations are determined by the number of frames. We also used a counter so that we can set a specific number of frames per animation to ensure fluid movement. Then depending on the state, the animation offset described in the formula above is determined a multiple of the player height times the player width plus the initial offset in memory. Then cycling through all the animations gives us the fluid movement that was seen of the player in the demo. We utilized a generalized system to ensure that the player is able to act appropriately depending on the signals and that no interference occurred as the user changed inputs for player movement using the keys. The images in the next two sections show the schematic block diagram of the running animation and running next state diagram. These are good examples of how the rest of the animations were designed.

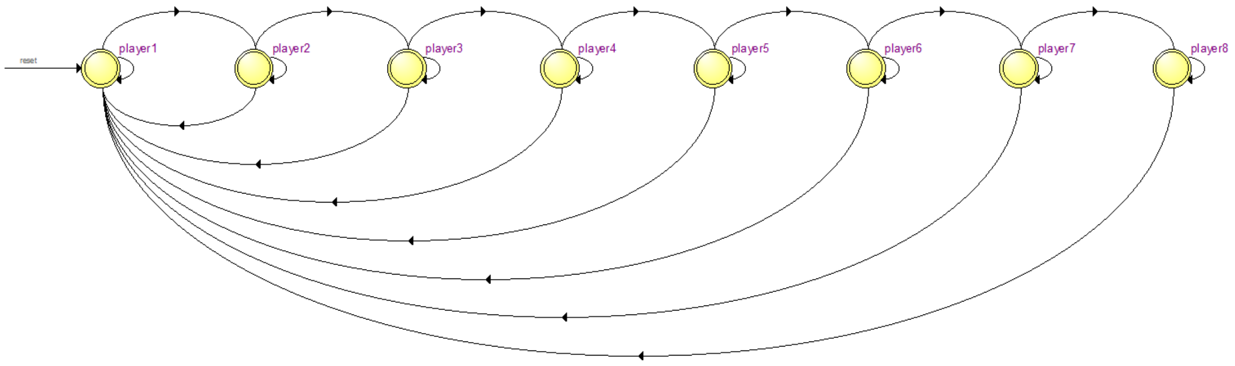
## 18.3 Running Animation Schematic Block Diagram

Due to its complexity, this block diagram is fairly difficult to understand. It is described in the section above.



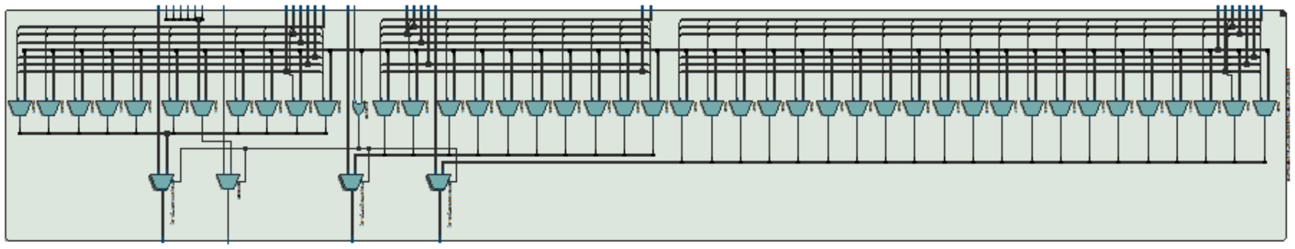
## 18.4 Running Animation Next State Diagram

The image below shows the next state diagram for the running animation. Each state corresponds to seeing a different sprite. More details on how this works are provided in section 18.2.



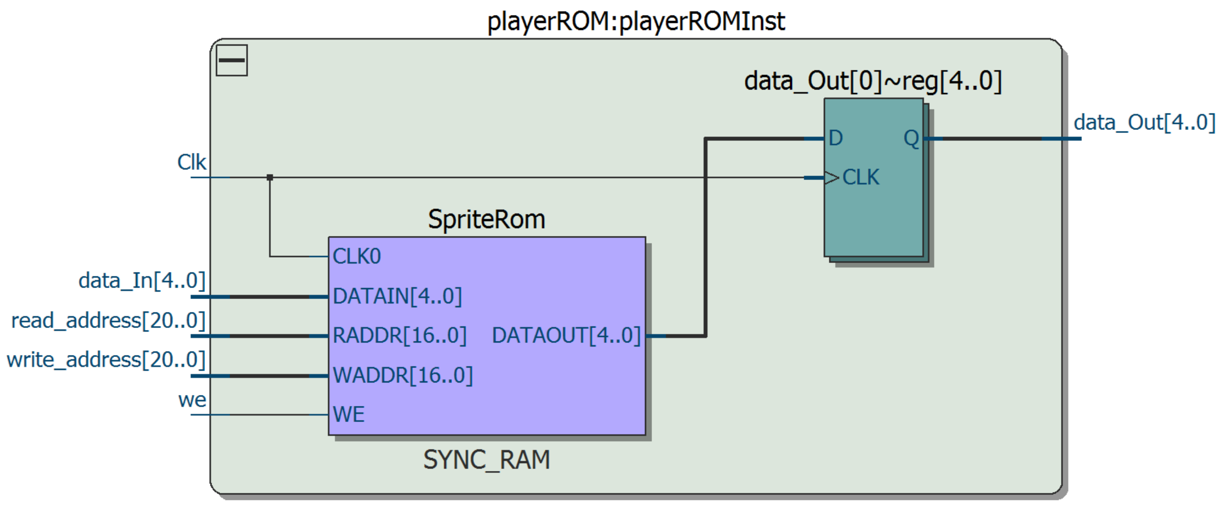
## 18.5 Animation Mux

The Animation Mux module is the module that selects the sprite pertaining to the animation of the Player. It takes in the addresses, player On, player Height, and Player Width from each of the modules for each animation, the keycode, and the on platform signal as inputs and output the correct sprite address, sprite on, and the player height and width. The purpose of the module is exactly like the name. It is a mux that has to different cases. If the player sprite is not on the platform or the user wants the player to jump, then select the sprite address from jump animations or else depending on the mapped key code, choose between the different animations of the player. For example, if the mapped key code is 4’h0 representing the player in the wait state, the animation mux will choose address, the player on, player height, and player width from the player animation representing the wait state for the player. The image on the next page shows the schematic block diagram of this module.



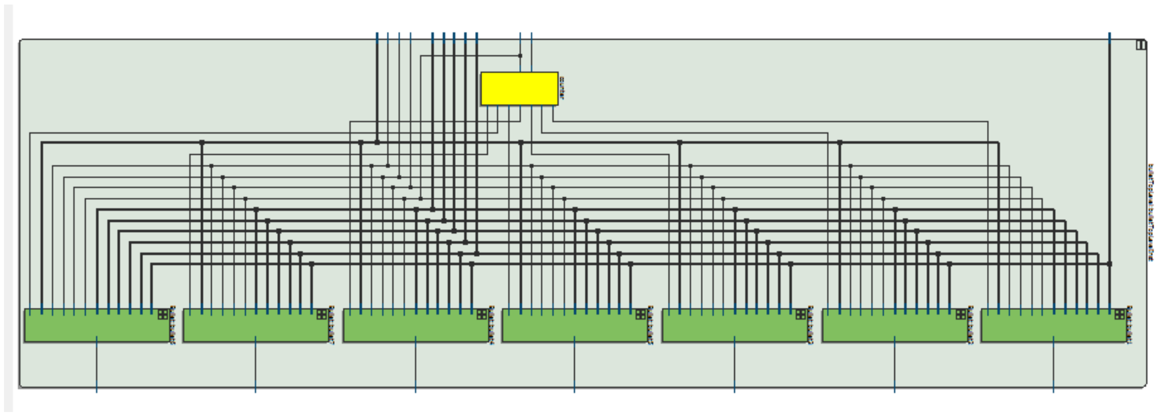
### 18.6 Player ROM

The Player ROM was implemented through the use of on-chip memory. The inputs into this module were Clk, data_In, read_address, write_address, and we. Because it is a ROM, we only utilized the CLK, and the read_address input. The output from this module was a 5 bit data_Out value. This block of memory allowed us to read from our Player Sprites text file and output the pixel at the read_address of the current sprite pertaining to the correct animation of the player. The schematic block diagram of this module is shown in the image below.



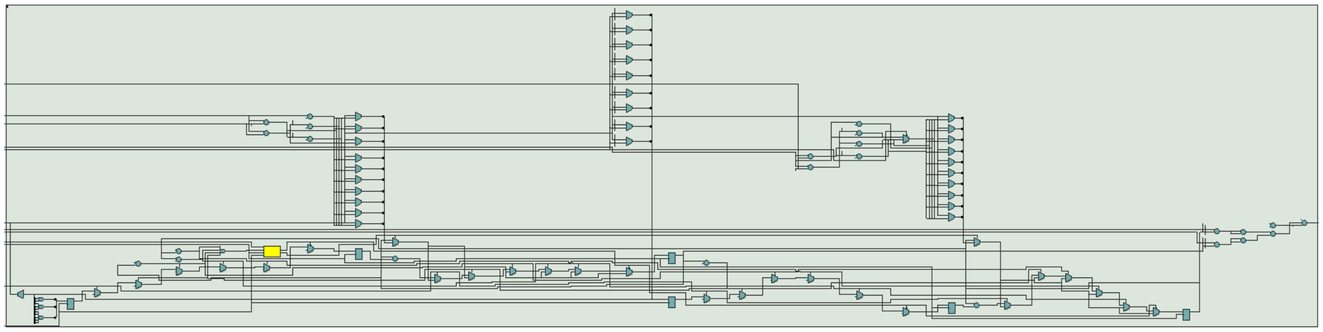
## 19. Bullet Top Level Module

The top level module for the bullet organized all of the bullets. As the game designers, we decided that we would define the max numbers of bullets on the screen at any given time to be equal to 7 bullets. This module bassically controller which bullet would get shot when the user pressed the shooting button. This was done through the use of a simple state machine. Every time the user pressed the shooting key, a counter was incremented. Based on the count of the counter, this module decides which bullet to shoot. It then relayed the control signals to the bullet submodules. Based on these signals, the bullets were either shot or kept hidden behind the player. The inputs into this module were Reset, VS, shootingEnable, collision, direction, keycode, PlayerX, PlayerY, PlayerHeight, PlayerWidth, DrawX, and DrawY. The outputs from this module were bulletOn1, bulletOn2, bulletOn3, bulletOn4, bulletOn5, bulletOn6, and bulletOn7. The inputs were all replayed to the the bullet submodules and based on the DrawX and DrawY positions, the submodules outputted bulletOn signals to signify to the pixel logic that they needed to be drawn. The image below shows the schematic block diagram of the bullet top level module.



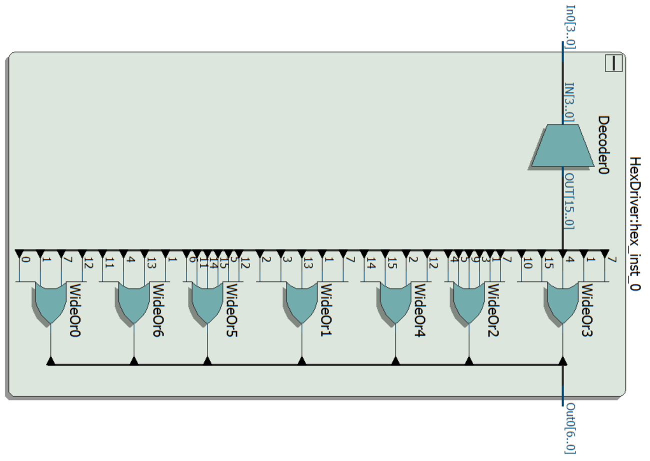
## 19.1 Bullet Module

The bullet module described and controlled all of the characteristics of each of the bullets. Through the hardware described in this module, we were able to initialize, draw, and move the bullet when the shoot key was pressed. The inputs to this module were Reset, VS, shootingEnable, collision, direction, keycode, PlayerX, PlayerY, PlayerHeight, PlayerWidth, DrawX, and DrawY. The output from this module was bullet_on. If the shooting key hadn't been pressed, then we needed to make sure that each of the bullets were being initialized to the correct location. This location was based on the direction of the player and then keycode that they were pressing. For example, if the user was pressing 'W' and 'D', then the user's direction is right and the player's gun is pointing in the top right direction. Based on this, the bullet position would then be changed to the top-right corner of the player sprite. Thus when the shooting key is pressed, the bullet will naturally seem like it is coming out of the gun. This type location and direction assignment was associated with all of the key presses and it allowed for more realistic shooting animations. Another important part of this module was its ability to take a snapshot of the current user direction and angle. Since the direction and angle were being constantly relayed to the module, we ran into some interesting problems where once the bullet had been shot, it would still change direction if the player changed direction. To fix this problem, we used registers to take a snapshot of the current player direction and angle on the rising edge of the shooting signal. This made sure that once the bullet was shot there was no way to change its direction and motion. The only way to reset the bullet was if it went of the screen or if it collided with an enemy. The DrawX and DrawY inputs were important because they allowed us to see if we were drawing at the position of the ball. If we were, then the ball_on signal was pulled to high. Otherwise, it was pulled low. This signal is then exported out of this module to be used by the pixel logic module to decide if we need to draw the ball. The image below shows the schematic block diagram of the bullet module. Due to its complexity, it is hard to decipher.



## 20. Hex Drivers Module

The hex driver modules were mainly used for debugging purposes. Since we had to modify and design our special version of the PS/2 driver, the hex drivers were very useful in visualizing the scan codes that our design was receiving from the keyboard. The driver takes in a 4 bit input and uses a decoder to define the meaning of each of the 16 patterns that might arise. This aided with setting up and making sure that the keyboard driver had been programmed correctly. We also used these modules to perform multiple other modular tests. The schematic of the hex driver module is shown in the image below.



## 21. Simulation Waveforms

Throughout the design process of our project, we did not really make use of simulations. Even though simulations would have been helpful, the scale of our project made it harder to design an overarching testbench that would work in all situations. Due to this, we just used modular design and onboard modular testing to see if everything was working well. For example, to get scrolling to work, we decided to just have the map on the screen by itself and have scroll when a key is pressed. We did not include the player at all and this allowed us to modularly test and make sure that the scrolling was working separately from the all other parts of the project. We use this type of test method throughout the project and so we didn't really need to do any simulations.

## 22. Design Statistics

The table below shows the design resources and statistics for this project. We see how many memory block implementation bits we used because of our instances of our player sprite rom and the frame buffer ram. We also used our DSP blocks to make multipliers that was used in our design.

| ***Resources*** | **Statistics** |
|---|---|
| *LUT* | 2371 |
| *DSP* | 40 |
| *Memory(BRAM)* | 2313216 |
| *Flip-Flop* | 486 |
| *Frequency (MHz)* | 96.29 |
| *Static Power (mW)* | 98.90 |
| *Dynamic Power (mW)* | 0.00 |
| *Total Power (mW)* | 188.64 |

## 23. Conclusion

While a lot of effort needed to be taken to create this system utilizing pure hardware, it was completely worth it to see the final results. Our major tasks included the framebuffer, player sprites, mapping the background, platform detection, and the keyboard. One of the hardest parts of this project was understanding the process of designing and setting up a system that would allow us to realize the vision we had for this project. While we were unable to recreate the entire game, we built a system that could be easily extended to incorporate enemy sprites. This project solidified my idea of modular design and how It can help make the design process faster and more accurate. If I have time in the future, I would love to expand the project and have it include the features that I did not have enough time add.