ECE385
DIGITAL SYSTEMS LABORATORY
UNIVERISTY OF ILLINOIS AT URBANA-CHAMPAIGN

FALL2015
SECTIONAB2
SHUBHAM PAHADIA AND WYATT MCALLISTER

FINAL PROJECT
MARIO IN SYSTEM VERILOG

INTRODUCTION:

For our final project we decided to create a System Verilog Implementation of Mario. We wanted to design a game which allowed us to showcase the skills we learned this semester by creating hardware architecture which drew high quality graphics and performed at speeds above a software implementation on a comparable platform. We designed our own custom graphics and went through a long design process to draw dynamic sprites on top of custom background images with very little memory usage and high graphical processing speeds. We implemented all the game mechanics in hardware in order to ensure faster performance.

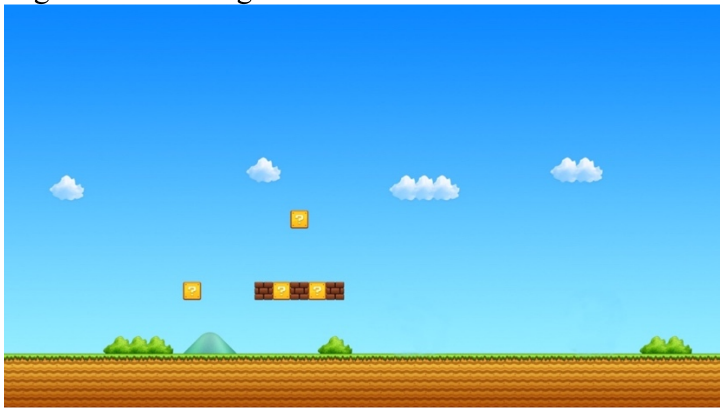
OVERVIEW OF GAMEPLAY:

This implementation of Mario has one level. Mario can collect coins by bumping into the bottom of a coin block and then colliding with the coin which then appears on top of this block. Mario must also avoid two types of monsters, Flying Koopas, which move up and down, and Goombas which move side to side along the ground. Mario will die if he collides with the monsters in any other fashion than by jumping on top of them, in which case the monster will die. The game ends when Mario Jumps on the pipe, in which case a second level could be added. This game was implemented in such a manner as to provide scalability, using custom graphics and an efficient all hardware implementation for higher performance. The reasoning behind using only one level is that we designed our code in such a way as to be scalable. We minimized the usage of memory and maximized processing efficiency. Focusing on only one level allowed us to perfect our game mechanics, and as extension to multiple levels is trivial, we decided this was more important than having a game with larger scale. The design process for two levels has been completed and shown below. The code mechanics remain the same. A star sprite was added to the design in the case that an end game state is desired.

GRAPHICAL DESIGN PROCESS AND GRAPHICAL PROCESSING METHOD:

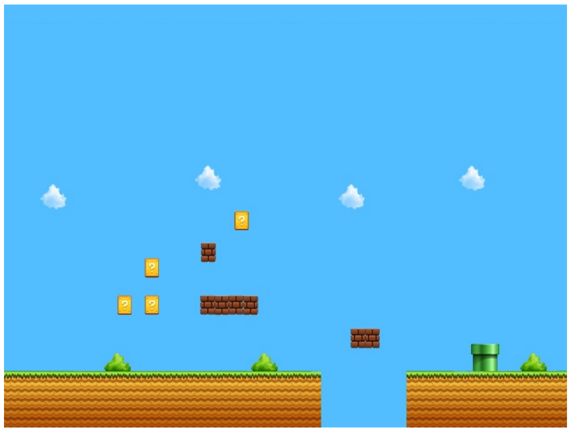
Part of our goal for designing this game was to be able to display custom graphics on an arbitrary background image. We first realized that in order to do this we would have to load and display full color images on the FPGA. We realized early on that we would have to use 8-bit color to do this. In order not to sacrifice graphical quality we decided to build our game with images which we could easily store as a 256 color dithered BMPs, an image format which processes an image as a color table of 256, 24-bit colors, with each pixel stored as an index into this color table. We first found a background image, shown below, that was clearly designed to be drawn using sprites and matched the contrast requirements to look satisfactory in a 256 dithered format.

Original Level Image Used to Generate Our Game Levels

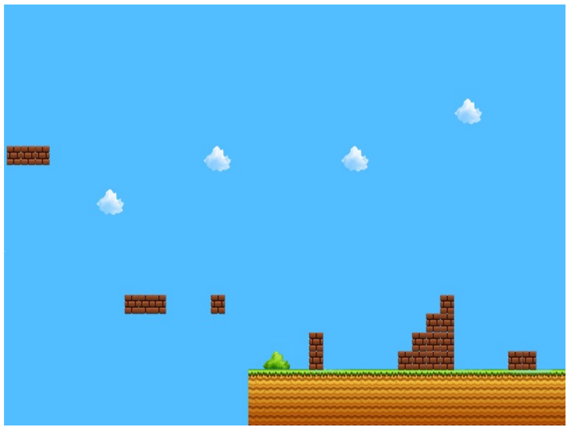


From this we created two levels which matched our specifications exactly.

Level 1

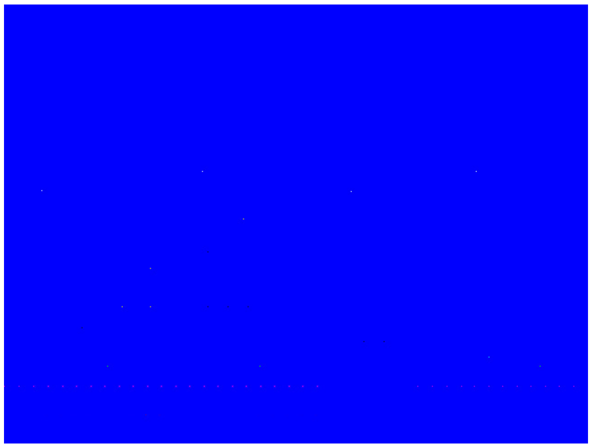


Level 2

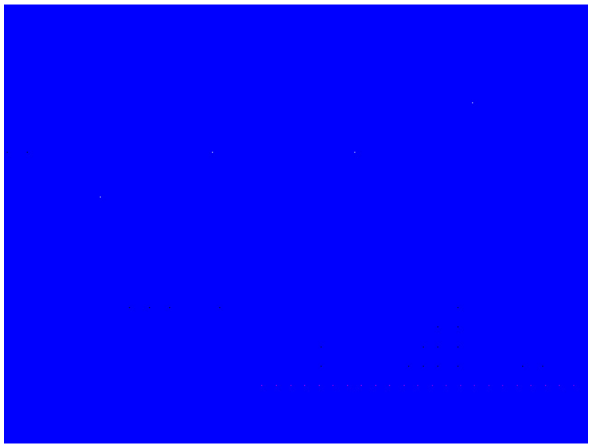


We initially thought of just storing the background images as raw index arrays and color tables inside the on chip memory of the FPGA. This would actually be feasible for a small scale game. However, we decided it would be more effective to draw the images as arrays of tiles on a background color. We created two structure files which indexed 256 color images into only a three-bit array of seven sprites plus the background color, shown below.

Level 1 Structure



Level 2 Structure

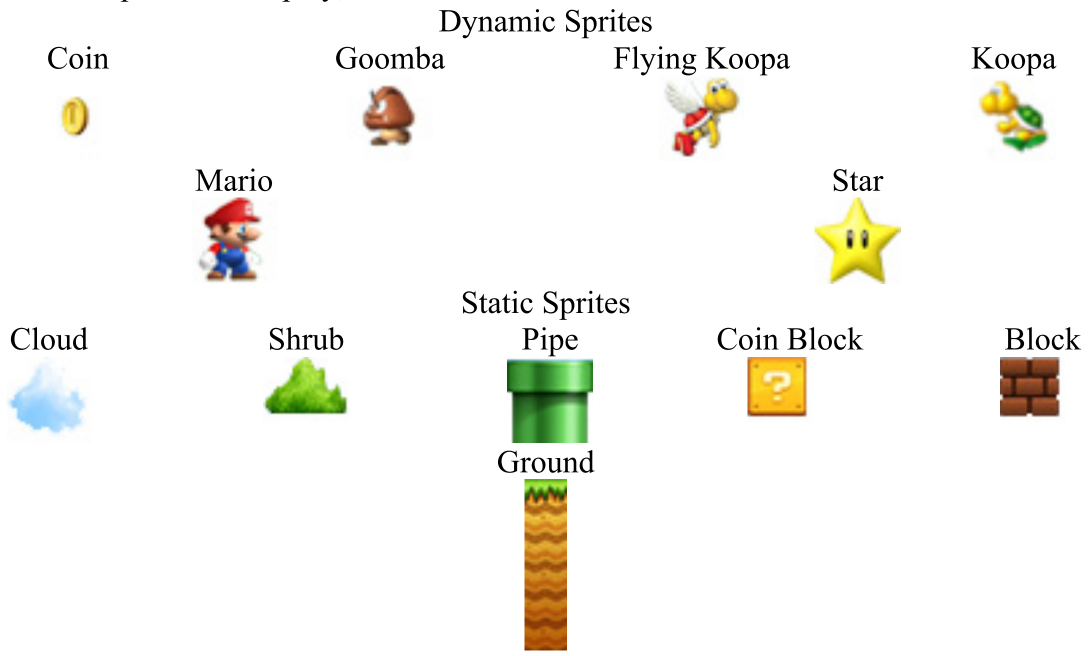


Carful inspection of these images will show that they consist of seven pixel colors interspersed on a blue background, following the conventions detailed in the table below. The blue background only provides a soft backdrop that is more pleasing to the eye when changing pixel locations for long periods of time.

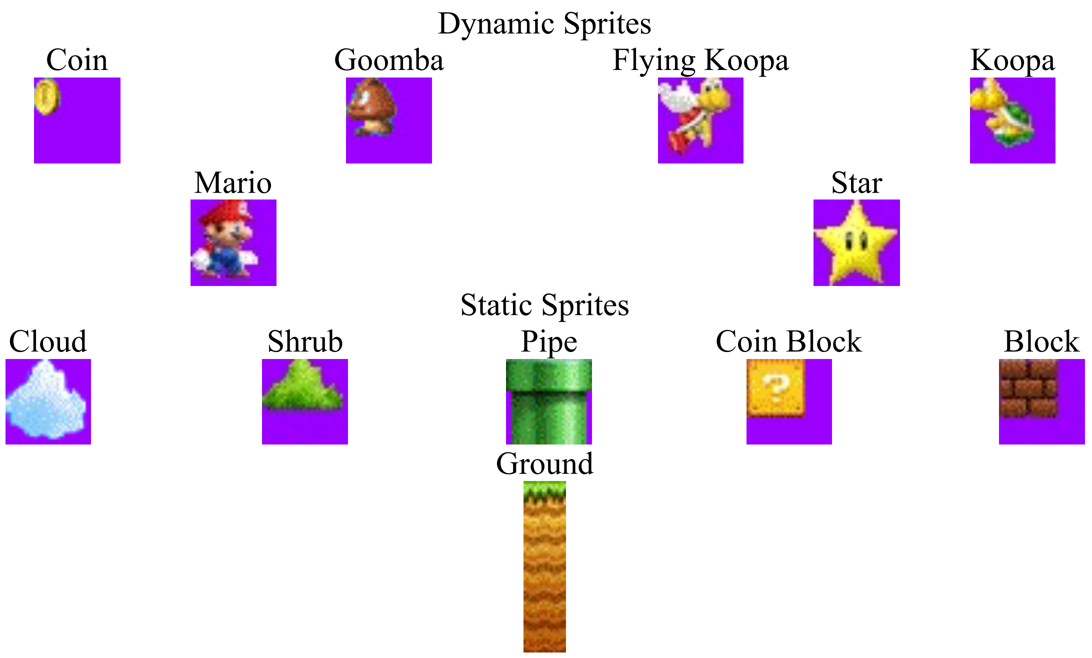
Sprite	Color	Red	Green	Blue
Cloud	White	255	255	255
Unassigned	Red	255	0	0
Shrub	Green	0	255	0
Background	Blue	0	0	255
Cyan	Pipe	0	255	255
Magenta	Ground	255	0	255
Yellow	Coin Block	255	255	0
Black	Block	0	0	0

This allowed us to create a more compact representation of the background which allowed for better scalability. We wrote C++ code using the Easy BMP library to process these images into an array of three bit values, which will be discussed in detail in the following section. We then thought of storing these arrays in memory and using them to populate the background. However, we realized that this would still require significant hardware overhead to locate the specific pixel locations within the image array. We realized that we could extend our C++ code to create an array of X and Y position values for all the background sprites for each level, which would allow us to effectively pre process a large quantity of image structures at scale and give us the capability of extending our game to an arbitrary size. We use two levels in this implementation for simplicity, choosing to store all the location arrays in register memory for ease of access. However, our method as well as the C++ code we wrote allows for ROM files to be created with ease. As the on chip memory of the Cyclone IV FPGA is thousands of times larger than the register memory, our game could be extended to an almost infinite number of levels, allowing for scalable game implementation on this hardware platform.

Once we had created the background structure for our game we set about completing the graphical design and processing phase by creating the necessary structures for the display of an arbitrary number of sprites on our background images. We decided to make almost all of our sprites 32 by 32, to allow for ease of processing and storage. We came up with thumbnails of all of our desired sprites for display, as shown below

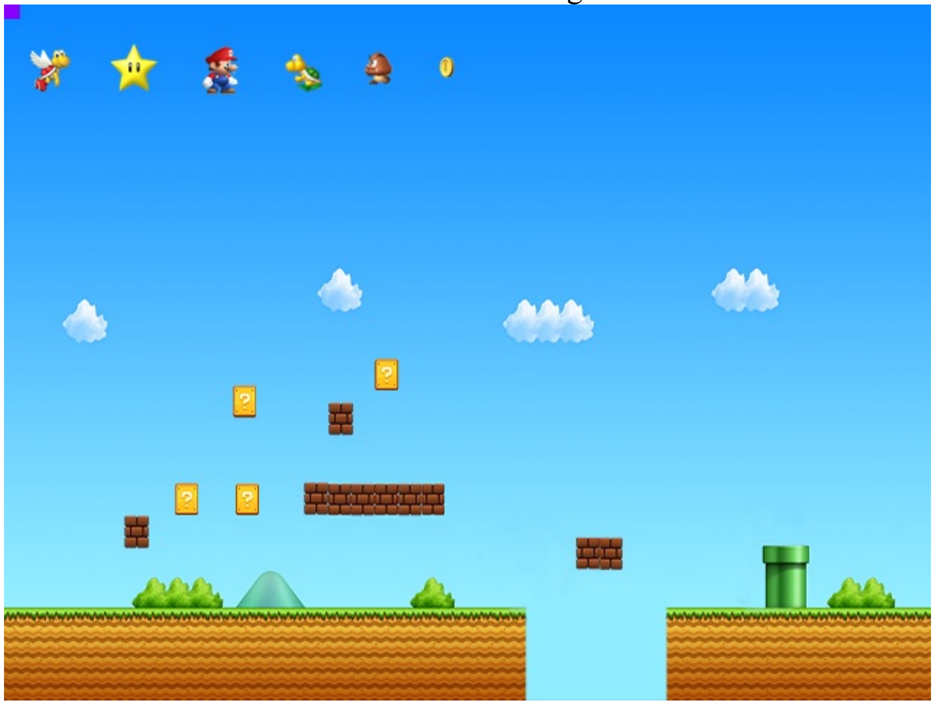


After this point, it was only a matter of processing our sprites with the Easy BMP library. With this library we were able to load BMP images which had been processed as 256 dithered BMPs. We generated these images by finding a Linux program that could perform the processing and running it in a sandbox. We decided to use a purple background color for all the sprites in order to ensure that we could recognize this distinct background color and have our logic draw other sprites behind it if needed. Our processed sprites are shown below.



Our C++ code then loaded the color table a master image, shown below, which contained all our sprites on top of one of our background images, plus a thumbnail of our purple transparent color. This allowed it to create a color pallet of 256, 24-bit entries which fully encapsulated the color spectrum of all our graphical structures.

Master Color Image

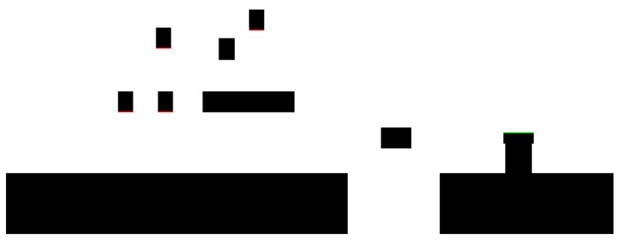


It was then a simple matter of using the BMP library to generate sprite tables for all our images which we could store on the On Chip Memory of the FPGA. We simply looped through each image and looked up each pixel color in the color palette and generated an array of indexes which we stored in an array, like the one shown below for the star sprite

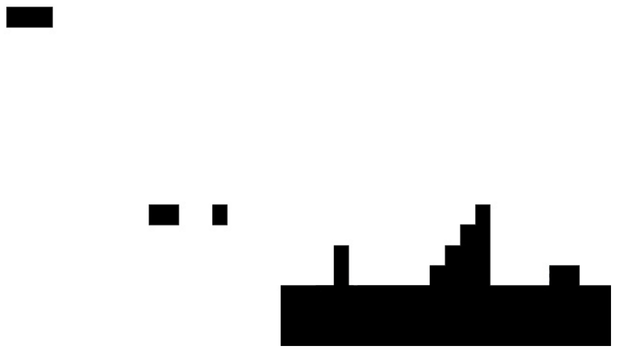
```
sprite5 <=
'{
'{215,215,215,215,215,215,215,215,215,215,215,215,215,215,215,212,170,215,215,215,215,215,215,215,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,215,215,215,215,215,215,215,213,212,205,178,215,215,215,215,215,215,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,215,215,215,215,215,215,205,212,211,170,215,215,215,215,215,215,215,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,215,215,215,215,215,215,177,212,213,205,169,214,215,215,215,215,215,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,215,215,215,215,215,215,211,206,213,212,204,171,215,215,215,215,215,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,215,215,215,215,215,207,211,212,213,212,211,162,215,215,215,215,215,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,215,215,215,215,215,175,205,213,213,212,211,168,164,215,215,215,215,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,215,215,215,215,208,211,212,213,213,212,211,211,126,215,215,215,215,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,215,215,215,215,205,211,213,213,213,212,212,211,168,170,215,215,215,215,215,209,215,215,215,215,215,},
'{215,215,215,215,215,209,215,214,172,214,170,211,212,213,207,212,212,212,211,205,169,171,208,178,215,215,215,215,215,215,215,},
'{215,215,208,214,171,212,176,205,211,211,212,212,213,213,213,213,206,212,212,211,211,210,205,211,169,206,177,207,178,208,215,215,},
'{212,176,212,211,212,212,212,212,213,212,213,213,213,177,213,212,212,212,169,212,211,211,211,211,211,211,211,211,211,211,169,171,},
'{206,211,211,206,212,212,206,213,213,213,213,213,128,79,170,213,212,176,79,169,212,211,212,211,211,211,211,211,205,205,168,171,},
'{215,169,210,211,211,212,212,212,213,207,213,176,122,129,170,212,212,128,129,127,212,211,211,212,211,205,211,168,168,126,171,215,},
'{215,215,170,168,211,211,211,212,212,212,212,206,128,129,170,212,212,127,129,85,212,211,205,211,211,204,169,168,162,171,215,215,},
'{215,215,215,169,204,205,211,211,211,212,212,176,43,42,170,212,212,163,42,85,211,211,211,169,204,169,168,126,171,215,215,215,},
'{215,215,215,215,169,168,174,205,211,211,211,212,42,255,170,211,212,127,255,84,211,211,204,169,168,162,126,171,215,215,215,215,},
'{215,215,215,215,208,169,204,210,211,205,212,211,42,255,205,211,211,169,255,85,211,205,168,169,168,126,170,215,215,215,215,215,},
'{215,215,215,215,215,215,169,168,211,211,211,211,127,85,212,211,211,169,85,168,205,210,169,168,162,127,215,215,215,215,215,215,},
'{215,215,215,215,215,215,172,168,205,211,211,211,211,211,211,211,211,211,211,169,204,169,162,128,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,206,210,210,205,211,211,211,211,211,211,211,210,205,210,169,168,162,171,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,211,211,211,211,211,211,211,204,211,204,205,168,211,168,205,168,126,171,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,215,169,211,211,211,211,204,211,168,205,174,205,210,169,204,169,168,162,170,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,214,210,211,205,210,211,169,210,169,168,168,169,168,205,168,169,168,162,134,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,207,211,210,211,175,205,204,168,168,168,168,168,168,168,204,168,169,168,127,208,215,215,215,215,215,215,},
'{215,215,215,215,179,215,177,210,211,205,204,168,168,163,162,126,162,162,168,163,168,168,162,168,126,215,215,215,215,215,215,215,},
'{215,215,215,215,215,215,207,210,211,168,168,168,126,126,126,170,127,126,126,168,126,168,162,169,162,172,215,215,215,215,215,215,},
'{215,215,215,215,215,215,176,204,205,168,168,126,126,164,172,215,215,171,163,126,162,126,168,126,168,171,215,215,215,215,215,215,},
'{215,215,215,215,215,215,205,210,168,126,126,163,172,215,215,215,215,215,215,171,127,126,132,162,162,170,215,215,215,215,215,215,},
'{215,215,215,215,215,214,211,168,126,163,172,215,215,215,215,215,215,215,215,215,215,171,163,126,132,164,215,215,215,215,215,215,},
'{215,215,215,215,215,208,168,127,171,215,215,215,215,215,215,215,215,215,215,215,215,178,171,162,127,214,215,215,215,215,215,},
'{215,215,215,215,215,178,171,214,215,215,215,215,215,215,215,215,215,215,215,215,215,215,215,172,171,215,215,215,215,215,215,},
};
```

Once we had completed this process all we had to do was figure out how to generate the required structures to implement the physics of our game. Initially we had thought of creating ghost images like the ones shown below to tell our physics engine where exactly the sprite was in relation to objects of interest.

Level 1 Ghost



Level 2 Ghost



We generated these images with the following four colors to tell our physics engine where space, and solid objects existed. We also added the capability to turn on coin blocks and find the pipe with these images. We used C++ code to generate a two-bit array.

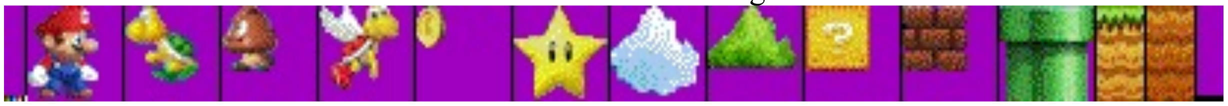
Object	Meaning	Color	Red	Green	Blue
Space	Sprite can move	White	255	255	255
Wall	Sprite Cannot Move	Black	0	0	0
Coin Block Bottom	Turn Coin On	Red	255	0	0
Pipe Top	Move to Next Level	Green	0	255	0

After doing all this, we realized that we would run into the same issue that we had before with the background drawing image structures. We would be using more memory than we could use with a more efficient and scalable implementation. We decided to just check the position of each moving sprite against all the positions of our known static objects from the sprite position array generated from the image structure. This gave us the capability of handling the physics for an arbitrary number of objects with much less memory use and processing overhead.

Finally, we made a major design decision. We realized that if we did all the game mechanics in software, we would decrease the performance of the game overall due to handshaking time more than we were improving it due to the CPU processing capabilities. The number of sprites we had was simply large enough to make a full hardware implementation a valid consideration. We realized that with the initial position of all of the sprites, both static and dynamic, stored in memory, we could easily create simple hardware logic to update the position of the dynamic sprites during gameplay, and account for the game mechanics when these dynamic sprites came into contact with one another. We created several hardware modules to do this and eventually came up with a game design which was both more scalable than anything we had originally envisioned, and had far greater performance than we had hoped.

Given this decision, we scaled up the complexity of our level structures and scaled down the complexity of our master color image. With our new structures we only needed sprite graphics, so we created a final master color image, shown below, with our sprite images and all the colors used in our structure images.

Final Master Color Image

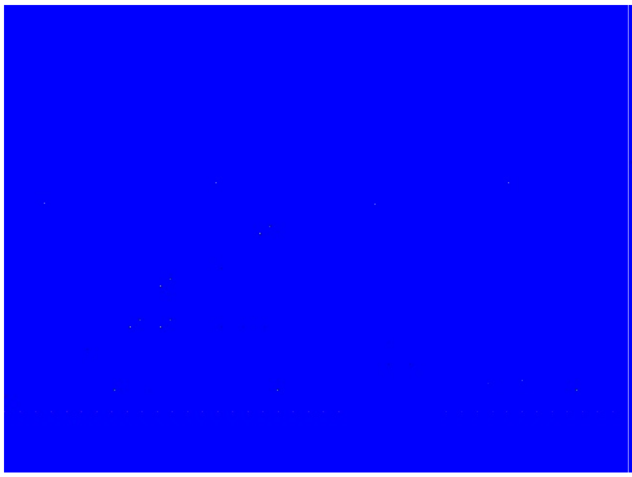


We then used our expanded structure to populate the initial positons of our dynamic sprites as well. For convenience we simply cut the RGB values of our previous colors in half and stored them in the master color image as well. The table of color values with its set of associated sprites is shown below.

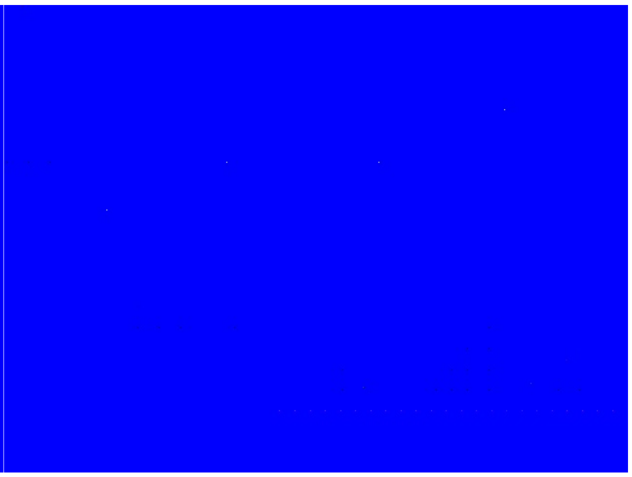
Sprite	Color	Red	Green	Blue
Cloud	White	255	255	255
Unassigned	Red	255	0	0
Shrub	Green	0	255	0
Background	Blue	0	0	255
Pipe	Cyan	0	255	255
Ground	Magenta	255	0	255
Coin Block	Yellow	255	255	0
Block	Black	0	0	0
Mario	Light Red	128	0	0
Koopa	Light Green	0	128	0
Goomba	Light Blue	0	0	128
Flying Koopa	Light Cyan	0	128	128
Star	Light Magenta	128	0	128
Coin	Light Yellow	128	128	0

Finally, we generated our set of expanded structures and used our C++ code to populate an array of initial positions for every single one of our sprites.

Level 1 Structure Final



Level 2 Structure Final



When we tried to implement the 256 color scheme described above, we had a lot of problems with the implementation. We first tried storing all the sprite tables in ROM files. We got a static background image to display. However, we had a lot of timing issues which prevented us from drawing dynamic sprites. We decided to try storing all the sprite tables in registers. However, we were unable to access all the register index values and the color table color values for our sprites within one clock cycle, since the registers that were synthesized were combinational.

We decided to extend our 256 color scheme to 8-bit color. We used the exact same implementation that is described so far. However, we added another part into the C++ code which scanned the sprite images above using the same color table from the master color image. Then, instead of outputting the index of the color into the master color image color table, we parsed the Red, Green, and Blue values from the image. We then assigned three-bit values for Red and Green, and a two-bit value for Blue using the scheme summarized in the table below. We checked the range of the Red, Green and Blue values. For Red and Green, we assign the value to seven if the color was in the range from 255 to 223, six if it was in the range from 222 to 190 and so on. For blue we assigned three if it was in the range from 255-191, two if it was in the range from 190 to 126 and so on. We then used this to create a three-bit array for red and green and a two-bit array for blue. We used these arrays in our graphics engine as summarized below to draw a set of different colors evenly distributed throughout the spectrum. This significantly reduced the resolution of our images but it decreased memory usage and processing time significantly and allowed us to display the sprites with ease.

Red/Green/Blue Range	255-223	222-191	191-159	158-127	126-95	94-63	62-31	30-0
Red/Green/Blue Index	111	110	101	100	011	010	001	000
Red/Green/Blue Color	255	222	191	158	126	94	62	30

We then ran into the problem of using our transparent color we needed some way to differentiate the purple background from another color in the same range, due to the decreased resolution. So, we added one bit to blue so the blue table looked like the following. We decided to fully utilize the last bit of blue so we could spread about the blue color spectrum as well and cut down on contrast issues.

Red/Green Range	255-223	222-191	191-159	158-127	126-95	94-63	62-31	30-0	RGB=Purple
Red/Green Index	111	110	101	100	011	010	001	000	N/A
Red/Green Color	255	222	191	158	126	94	62	30	Transparent

Red/Green/Blue Range	255-219	218-183	182-147	146-111	100-75	74-39	38-0	RGB=Purpose
Red/Green/Blue Index	110	101	100	011	010	001	000	111
Red/Green/Blue Color	255	222	191	158	126	94	62	N/A

This final image structure allowed us to successfully display eight-bit color images dynamically with far less memory usage than the previous 256-color scheme, and it speeded up our design significantly.

IMMPLEMENTATION OF GRAPHICAL PROCESSING WITH EASY BMP (C++):

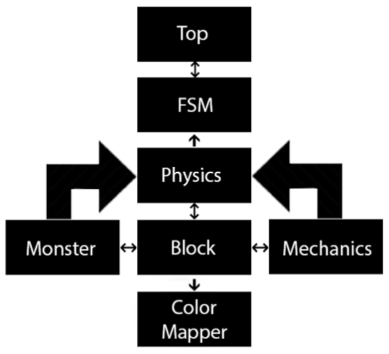
In order to process the above images to compile our game structure, we used the Easy BMP C++ library. This library has the capability to read the color palette from a BMP image and also read off the RGB values for an arbitrary pixel using the images own color table. In order to parse our images to create the proper structure, we first loaded the final master color image shown above and parsed the color table as a 256 entry, 24-bit array. We then used this color table to generate index arrays for all the sprite images of the form of the example shown above for the star sprite. Finally, we used the color pallet shown in the bottom left corner of the master color image to parse the structure images shown directly above for the X and Y values of the initial position of all our sprites. We then simply assigned the initial X and Y values of our position array for each set of sprites and for each level.

For our eight-bit color implementation, we simply scanned the BMP sprite images for the RGB values of each pixel and assigned the index values as describe above. We created three arrays for each sprite, one for Red, one for Green, and one for Blue. Finally, we processed these arrays to make register files in System Verilog in order to successfully implement the graphics scheme in hardware.

The code for our C++ implementation and the final images used are attached to this report. This code is highly scalable and can be used to generate an arbitrary number of levels of this type, if a larger game is to be created in the future.

HARDWARE IMMPLEMETATION AND DESIGN:

Top Level Block Diagram



In order to implement our design effectively in hardware, we created a two tiered hardware structure detailed in the top level block diagram shown above. Our top level module drove a finite state machine whose inputs were updated by our physics engine. This physics engine took in the position of our sprite and determined its relative position to all the other dynamic sprites. For all our sprites, we simply created an On Vector to turn each and every one of the sprites of the same type on or off during gameplay.

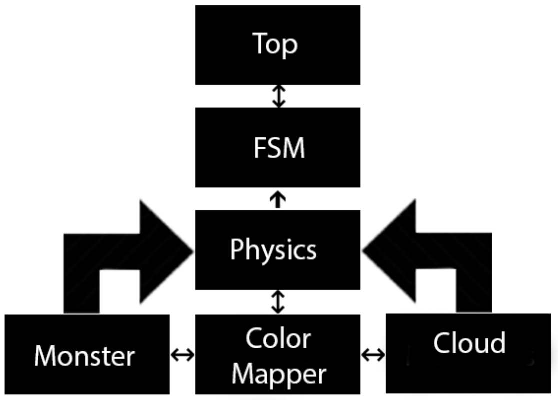
Our original design simply included one physics engine which used a motion vector to account for the physics of Mario and the monster sprites in one module. We did this by creating a motion vector for each monster whose mechanics is summarized in the following table. When controlling Mario, we simply asserted zero for motion which caused our physics engine to take input from the keyboard. When using a Flying Koopa, we simply pulled the motion vector high which caused our physics engine to have the Flying Koopa continuously jump after falling back to the ground. For the lateral motion of the Goombas we simply had two other values of the vector for left and right. We then had another module called Monster which set the motion vectors of each Goomba to a leftwards or rightwards motion. We simply had a counter which counted the distance of the Koopa or Goombas lateral motion and flipped the motion vector after this distance had been expended. We then created an entirely separate mechanics module for collisions between Mario and and the dynamic sprites such as the monsters and the coins and star. However, we ended up changing this design significantly

Motion	Meaning	Value
None	Motion for Mario	00
Left	Leftwards Motion for Koopas and Goombas	01
Right	Rightwards Motion for Koopas Goombas	10
Up	Upwards motion for Flying Koopas	11

We that our previous design wasted a lot of computation time as some of the computations for Mario were unnecessary for the monsters. After we simplified our design, our graphics engine ran a lot faster since it only needed to take input from one large physics block. We also simplified the IO between all our modules. We originally had all the updates of the position vectors and on vectors synchronized through a Block module which stored these values and sent them back to the other hardware modules for updates at each clock cycle. However, we saved hardware overhead by keeping all the position values in the top level module and only passing the values to the few hardware modules which needed them.

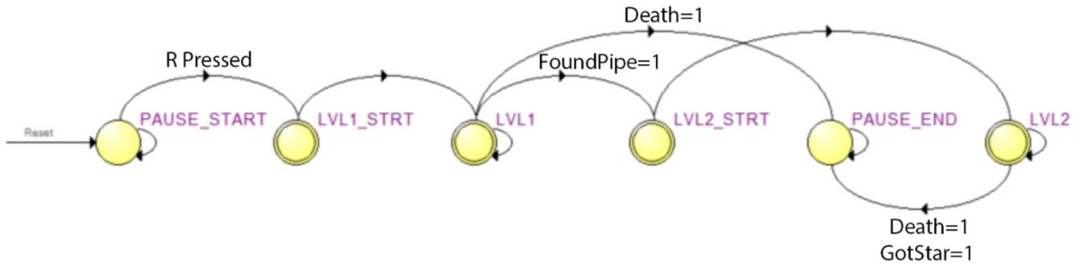
We revised our design and created modules to check the position of Mario against the pipe position to output found pipe, and a monster module to change the position of all our monsters and check them against Mario’s position to output found death. We also created modules to check the position of Mario against an input sprite and to check Mario’s position against the edge of the screen. Our new physics engine then only had to update the position of Mario dynamically in order to account for gravity, collisions, and upwards, rightwards, and leftwards motion commands, with the parameters of whether there was a wall above, below, to the left, or to the right simply taken as input. Having a separate module for the monster mechanics and monster collisions with Mario allowed us to simplify our design a lot and save on logic and memory. Our new block diagram is shown on the following page and greatly increases performance.

Top Level Block Revised



FINITE STATE MACHINE DESIGN:

FSM Diagram



We started with a two level state machine for a larger. It had a state transition diagram shown above. Its mechanics is as follows and is summarized in the tables below. We started game play in a Start State, which allowed the game to Pause until the R Key, which we used for our Run signal, was pressed. Our game then transitioned to Level 1 Start, the state where all our initial values for Level 1 were loaded and the level one game mechanics prepared. Our game immediately transitioned to Level 1 and stayed there until the found pipe signal was asserted. We then had a similar sequence of transitions through a Level 2 Start State where our level two game mechanics was initialized and into the Level Two State, where it stayed until found star was asserted before transitioning to the End State. If found death was asserted in either of our Level 1 or Level 2 game states, the game immediately transitioned to the end state where our hardware waits until the game restarts.

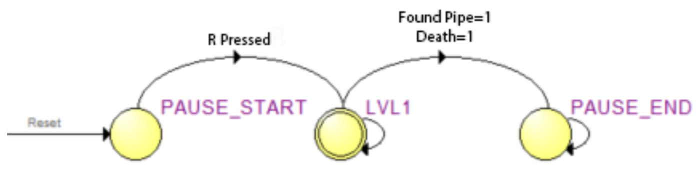
Output Logic Table

State	Level 1 Start	Level 2 Start	Level 1 Not Level 2	Not Pause
Pause Start	0	0	0	0
Level 1 Start	1	0	0	1
Level 1	0	0	1	1
Level 2 Start	0	1	0	1
Level 2	0	0	0	1

Pause End	0	0	0	0

Next State Table					
Current State	Run	Death	Found Pipe	Got Star	Next State
Pause Start	1	X	X	X	Level 1 Start
Pause Start	0	X	X	X	Pause Start
Level 1 Start	X	X	X	X	Level 1
Level 1	X	0	1	X	Level 2 Start
Level 1	X	1	X	X	Pause End
Level 2 Start	X	X	X	X	Level 2
Level 2	X	X	X	1	Pause End
Level 2	X	1	X	X	Pause End
Level 2	X	0	X	0	Level 2

For our scaled down design, our state machine only needed three states. Synchronizing our sprite positions in the top level eliminated the need for a start state for the level. We simply had start and end states as before and a game play state for our level 1 game mechanics. We also added a pause state which outputted a pause signal to stop the game in the middle of game play. This added a useful feature and also helped for debugging purposes. Our final FSM has diagram and state tables shown below.



Output Logic Table	
State	Pause
Pause Start	1
Level 1	0
Pause End	1
Pause	1

	Next State Table					
Current State	Found Pipe	Death	Reset	Run	Pause	Next State
Pause Start	X	X	X	1	X	Level 1
Pause Start	X	X	X	0	X	Pause Start
Level 1	X	X	X	X	1	Pause
Level 1	1	X	X	X	X	Pause End
Level 1	X	1	X	X	X	Pause End
Pause End	X	X	X	X	X	Pause End
Pause	X	X	X	1	X	Level 1
Pause	X	X	X	0	X	Pause

SYSTEM VERILOG IMMPLEMENTATION:

```
module physics (
        input frame_clk,                        // vsync
        input MarioOn,                          // If Mario is On
        input [4:0] MarioSize,                  // size of sprite we want to control
        input [7:0] keycode,                    // Key Code
        input [9:0] MarioX, MarioY,             // Mario X and Y Position
        input WallL, WallR, WallU, WallD,       // Whether We Have a Wall
        output [9:0] XOUT, YOUT,                // Mario Output Position
);
```

This module implements our improved physics engine. It takes as input the Wall parameters which allow it to give Mario the ability to respond to movement commands, fall with gravity, and only move in the appropriate reginos. We simply continuously append the position of Mario by instantiating the physics engine in the top level. Collisions with the Monsters and Coins are completely accounted for with all of our other logic so all physics needs to do is determine Mario's motion in space. Calling the physics engine with the wall parameters allows us to simplify our design significantly.

```
module bounds (
        input frame_clk,                        // vsync
        input MarioOn,                          // If Mario Is On
        input [4:0] MarioSize,                  // Size of Mario
        input [7:0] keycode,                    // Key Code
        input [9:0] MarioX, MarioY,             // Mario X and Y Positions
        input [9:0] XMax, YMax                  // Bounds of Screen
        output WallL, WallR, WallU, Death,      // If We Have found a Wall or Died
);
```

The above module simply ensures that Mario follows the bounds of the screen. It sets Wall parameters at the top, left, and right edges of the screen and asserts the death signal if Mario falls off the bottom of the screen and dies.

```
module collision (
        input frame_clk,                        // vsync
        input MarioOn,                          // If Mario Is On
        input [4:0] MarioSize,                  // Size of Mario
        input [7:0] keycode,                    // Key Code
        input [9:0] MarioX, MarioY,             // Mario X and Y Positions
        input [9:0] SpriteX, SpiteY,            // X and Y Position of
        input [9:0] SpriteWidth, SpriteHeight,  // Width and Height of Sprite
        output WallL, WallR, WallU, WallD,      // If we have found a wall
);
```

This module is called on Mario with each of the static sprites which Mario cannot enter. We simply check the borders of Mario against the borders of the sprites and output the Wall logic needed to drive our physics engine, allowing us to simplify our design and use less logic to ensure Mario stays within the appropriate boundaries.

```
module monster_mod (
        input Reset, Clk, pause, monsterOn_Cur,
        input [9:0] MarioX, MarioY,
        input [9:0] MonsterWidth, MonsterHeight,
        input [9:0] MonsterX_Init, MonsterYInit,
        input [9:0] MonsterX_Max, MonsterY_Max,
        input [9:0] MonsterX_Min, MonsterY_Min,
        input [9:0] Monster_XStep, Monster_YStep,
        output [9:0] Monster_X, Monster_Y,
        output [9:0] Monster_XstepOUT, Monster_YstepOUT,
        output gameover, monsterOn
    );
```

This module, as described above, takes in the positions of a monster sprite and of Mario. It simply updates the positions of the monster sprite according to the input value and then continuously checks this value against the value of Mario in order to check for death of Mario or Death of the monster. We simply called this module for every monster in our game instead of having a large module for all the monsters. This allows simplification of our design and portability in the graphics engine.

```
    module cloud (
        input Reset,
        input Clk,
        input [9:0] CloudX_InIt
        output [9:0]  cloudX
    );
```

This is a very simple module that handles the positions of the cloud. We realized that since the cloud is the only dynamic sprite and doesn't affect game mechanics, having a module for it in the top level would greatly simplify our drawing of this sprite since it could be directly fed to color mapper.

```
    module coin_on_check( input Reset, Clk, pause, coin_val,
        input [9:0] MarioX, MarioY,
        input[9:0] CoinBlock_X, CoinBlock_Y,
        input[9:0] Mario_Size, CoinBlock_Size
        output Coin_On
    );
```

This module handles all the coin mechanics in one block. It checks Mario's position against the that of the Coin blocks and the coin sprites and turns the coins on when Mario hits the bottom of the coin block and off after Mario then gets the coin from the top of the block.

```
    module pipe_check( input Reset, Clk, pause,
        input [9:0] MarioX, MarioY,
        input [9:0] Pipe_X, Pipe_Y,
        input [9:0] Mario_Size, Pipe_Size
        output Done
    );
```

This pipe checker module was called in our top level and instantiated all the logic to calculate found pipe for the state machine. It simply checked Mario's positions against that of the pipe sprite and made the appropriate calculations of the found pipe signal

Our color mapper module was instantiated in the top level and took all the positions of all of our static and dynamic sprites as input. It simply checked the value of the current pixel to be drawn against the position of all the sprites and generated the appropriate color of that sprite by reading the appropriate Red, Green, and Blue index values from the index arrays for that sprite and choosing the correct color as describe above. Color mapper drew the sprites in order of priority so that Mario would show over a shrub and cloud would behind monsters.

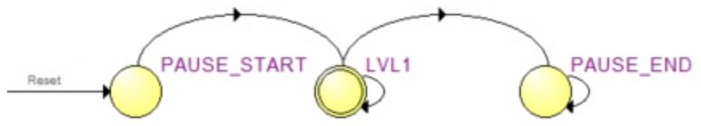
```
module color_mapper (
		input Clk,
		input [9:0] MarioX,
		input [9:0] MarioY,
		input [9:0] DrawX,
		input [9:0] DrawY,
		input [9:0] Mario_Size,
		input [9:0] BlockX [0:7],
		input [9:0] BlockY [0:7],
		input [9:0] Block_size,
		input [9:0] GroundX [0:39],
		input [9:0] GroundY,
		input [9:0] GroundX_Size,
		input [9:0] GroundY_Size,
		input [9:0] PipeX,
		input [9:0] PipeY,
		input [9:0] FlyKoopaX,
		input [9:0] FlyKoopaY,
		input [9:0] Pipe_Size,
		input [9:0] FlyKoopa_Size,
		input [9:0] CoinBlockX_Pos0,
		input [9:0] CoinBlockX_Pos1,
		input [9:0] CoinBlockX_Pos2,
		input [9:0] CoinBlockX_Pos3,
		input [9:0] CoinBlockY_Pos0,
		input [9:0] CoinBlockY_Pos1,
		input [9:0] CoinBlockY_Pos2,
		input [9:0] CoinBlockY_Pos3,
		input [9:0] CoinBlock_size,
		input [9:0] CloudX_Pos0,
		input [9:0] CloudX_Pos1,
		input [9:0] CloudX_Pos2,
		input [9:0] CloudX_Pos3,
		input [9:0] CloudY_Pos0,
		input [9:0] CloudY_Pos1,
		input [9:0] CloudY_Pos2,
		input [9:0] CloudY_Pos3,
		input [9:0] Cloud_size,
		input [9:0] CoinX_Pos0,
		input [9:0] CoinX_Pos1,
		input [9:0] CoinX_Pos2,
		input [9:0] CoinX_Pos3,
		input [9:0] CoinY_Pos0,
		input [9:0] CoinY_Pos1,
		input [9:0] CoinY_Pos2,
		input [9:0] CoinY_Pos3,
		input [9:0] Coin_size,
		input [9:0] ShrubX_Pos0,
		input [9:0] ShrubX_Pos1,
		input [9:0] ShrubX_Pos2,
		input [9:0] ShrubY_Pos0,
		input [9:0] ShrubY_Pos1,
		input [9:0] ShrubY_Pos2,
		input [9:0] ShrubX_size,
		input [9:0] ShrubY_size,
		input [9:0] GoombaX_Pos0,
                input [9:0] GoombaX_Pos1,
                input [9:0] GoombaY_Pos0,
                input [9:0] GoombaY_Pos1,
                input [9:0] Goomba_size,
                input FlyKoopa_Show,
                input Mario_Show,
                input Goomba_Show0,
                input Goomba_Show1,
                input [0:31] [0:31] [0:3] marioredarr,
                input [0:31] [0:31] [0:3] mariobluearr,
                input [0:31] [0:31] [0:3] marioreenarr,
                input [0:31] [0:31] [0:3] blockredarr,
                input [0:31] [0:31] [0:3] blockbluearr,
                input [0:31] [0:31] [0:3] blockgreenarr,
                input [0:31] [0:31] [0:3] groundredarr,
                input [0:31] [0:31] [0:3] groundbluearr,
                input [0:31] [0:31] [0:3] groundgreenarr,
                input [0:31] [0:31] [0:3] piperedarr,
                input [0:31] [0:31] [0:3] pipebluearr,
                input [0:31] [0:31] [0:3] pipegreenarr,
                input [0:31] [0:31] [0:3] flykooparedarr,
                input [0:31] [0:31] [0:3] flukoopabluearr,
                input [0:31] [0:31] [0:3] flykoopagreenarr,
                input [0:31] [0:31] [0:3] coinblockredarr,
                input [0:31] [0:31] [0:3] coinblockbluearr,
                input [0:31] [0:31] [0:3] coinblockgreenarr,
                input [0:31] [0:31] [0:3] cloudredarr,
                input [0:31] [0:31] [0:3] cloudbluearr,
                input [0:31] [0:31] [0:3] cloudgreenarr,
                input [0:31] [0:31] [0:3] coinredarr,
                input [0:31] [0:31] [0:3] coinbluearr,
                input [0:31] [0:31] [0:3] coingreenarr,
                input [0:31] [0:31] [0:3] shrubredarr,
                input [0:31] [0:31] [0:3] shrubbluearr,
                input [0:31] [0:31] [0:3] shrubgreenarr,
                input [0:31] [0:31] [0:3] goombaedarr,
                input [0:31] [0:31] [0:3] goombabluearr,
                input [0:31] [0:31] [0:3] goombagreenarr,
                output logic [7:0] Red, Green, Blue
        );
```

Our top level module has module descriptor and synthesis diagram detailed below. It is clocked at the FPGAs native clock rate and takes Key Code input from an identical software structure to that of lab eight. It contains the necessary interfaces for the VGA monitor which displays our game, the Easy On The GO (EZOTG) USB Driver Peripheral for the keyboard peripheral, and the SDRAM interface for the NIOS CPU Subsystem which implements our software structure.

```
        module FinalProject (
                input CLOCK_50,                     //
                input [3:0] KEY,                    // bit 0 is set up as Reset
                output [6:0] HEX0, HEX1,            //
                // VGA Interface
                output [7:0] VGA_R,                 // VGA Red
                output [7:0] VGA_G,                 // VGA Green
                output [7:0] VGA_B,                 // VGA Blue
                output VGA_CLK,                     // VGA Clock
                output VGA_SYNC_N,                    // VGA Sync signal
                output VGA_BLANK_N,                   // VGA Blank signal
                output VGA_VS,                        // VGA vertical sync signal
                output VGA_HS,                        // VGA horizontal sync signal
                // CY7C67200 Interface
                inout [15:0] OTG_DATA,                // CY7C67200 Data bus 16 Bits
                output [1:0] OTG_ADDR,                // CY7C67200 Address 2 Bits
                output OTG_CS_N,                      // CY7C67200 Chip Select
                output OTG_RD_N,                      // CY7C67200 Write
                output OTG_WR_N,                      // CY7C67200 Read
                output OTG_RST_N,                     // CY7C67200 Reset
                input     OTG_INT,                    // CY7C67200 Interrupt
                // SDRAM Interface for NIOS II Software
                output [12:0] DRAM_ADDR,              // SDRAM Address 13 Bits
                inout [31:0] DRAM_DQ,                 // SDRAM Data 32 Bits
                output [1:0] DRAM_BA,                 // SDRAM Bank Address 2 Bits
                output [3:0] DRAM_DQM,                // SDRAM Data Mast 4 Bits
                output DRAM_RAS_N,                    // SDRAM Row Address Strobe
                output DRAM_CAS_N,                    // SDRAM Column Address Strobe
                output DRAM_CKE,                      // SDRAM Clock Enable
                output DRAM_WE_N,                     // SDRAM Write Enable
                output DRAM_CS_N,                     // SDRAM Chip Select
                output DRAM_CLK                       // SDRAM Clock
        );
```

Our FSM module has the following module descriptor and synthesis diagram. It is also clocked a FPGA speeds and takes inputs of Got Start, In Pipe, and Game Over from our Physics engine. It outputs the necessary logic: notPause to drive our lower level hardware logic.

```
module Game_FSM (
        input logi Run, Reset, Clk,
        input logic gotStar, inPipe, gameover,
        input logic [7:0] keycode,
        output logic notPause
);
```



The three modules given below were left over from lab 8. They implement the VGA controller, the IO interface with the USB peripheral, and the Hex Driver Display interface with the Altera DE2 Development Board.

```
module vga_controller (
        input Clk,                            // 50 MHz clock
         input Reset,                         // reset signal
        output logic hs,                      // Horizontal sync pulse.  Active low
        output logic vs,                      // Vertical sync pulse.  Active low
        output logic pixel_clk,               // 25 MHz pixel clock output
        output logic blank,                   // Blanking interval indicator.  Active low.
        output logic sync,                    // Composite Sync signal.  Active low
        output [9:0] DrawX,                   // horizontal coordinate
        output [9:0] DrawY                    // vertical coordinate
);
```

```
module hpi_io_intf (
        input [1:0] from_sw_address,
        output [15:0] from_sw_data_in,
        input [15:0] from_sw_data_out,
        input from_sw_r, from_sw_w, from_sw_cs,
        inout [15:0] OTG_DATA,
        output [1:0] OTG_ADDR,
        output OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N,
        input OTG_INT, Clk, Reset
);

module HexDriver (
        input [3:0] In0,
        output logic [6:0] Out0
);
```

CONCLUSION:

In this lab we went through a professional design process of a custom game including graphic design, image processing, sprite design, game mechanics, hardware constrains, memory usage, and hardware architecture comparison for several designs. At each stage in the processes we had at least two design iterations in order to reach the optimal design for performance and quality. We ended up creating a scalable gaming platform that could handle multiple levels and complex dynamics between sprites like object collection and dynamic collisions.

We implemented our entire design in hardware with minimal memory usage and high speed graphics with zero time needed for handshaking with software. Using eight-bit color allowed us to increase the speed of our graphics engine tremendously and allowed us to draw dynamic sprites with ease by accessing only one set of memory arrays. Our entire design was portable and scalable to a large amount of sprites and levels. We made the design decision to use only one level in this implementation in order to focus on improving our graphics. Our image processing code works with any set of BMP images and can generate eight-bit color and 256 color sprite tables, in register or ROM format.

By going through a long design process, we came up with a design which was of far higher quality than any of our original implementations. We used a full hardware implementation to cut down on handshaking time, parsed all the physics and position data in C++ to eliminate the need to process the entire image in hardware and make our design more scalable, stored the sprite data in registers to make our design fully synchronous, created smaller more portable modules to effectively scale our design and make it far more portable, and used an eight-bit color scheme that allowed for faster graphics speed.

This project showcased everything we learned throughout the course of the semester. We created a project architecture that used graphics, user input, hardware and software interfaces for the user interface, memory architecture, and large scale dynamics between modules. We used all our skills at debugging large scale designs to figure out exactly where our design was going wrong, and make intelligent decisions about how to fix it, and whether larger changes to our design would be more effective. Over the course of the semester we learned how to go through a professional design process for a custom hardware design. The game we made was small, but the design we built could easily be extended to any size on this hardware platform. The product we made is something that cannot be reproduced on a software platform, and its original design showcases the depth of intuition for the FPGA hardware architecture we have build over the course of the semester by designing custom hardware from the ground up, in all its forms.