**ECE 385**

Spring 2023

Final Experiment

**NES Super Mario Bros**

Jason Wright, Noah DuVal

DJ (JZ) / Friday 5:20 pm

Dohun Jeong

**Introduction**

When it came to doing the last and final project for the semester we followed our initial idea of making some sort of a video game as they are easy to scale once base functionality is achieved. Thus, allowing us to pick and choose what we wanted to add to the game as we went along and had time for. Following choosing a video game, the choice of creating *NES Super Mario Bros* was made. The idea being that it would be relatively simple to implement base features and then we can add the challenging features as we saw fit with time left, but would still be well off if we were only able to get basic functionality. Given that we are both EE and have minimal coding experience outside of ECE 220. We knew while still simple it was going to still be a bit of a challenge.

The original plan was to have the ability to draw on the screen and create in a sense a frozen screen of the game by the midpoint check-in for the project. Then add the rest of it such as, game logic and movement to the sprites. However, the first two weeks we both had exams and did not get as much done as we would have liked. Thus, we were a little behind and when it actually came around to the mid-point check-in we did not yet have anything to display for the TA, but we were close as the frame buffer was almost complete and all of the sprite work and logic was complete to the best of our knowledge at the time. In the weeks to come we went to the background to display and started to place objects on the screen and began working on the movement aspect of the game. However, we ran into multiple problems along the way that started to get harder and harder to diagnose as we went along. For example, we had issues with writing to and from the frame buffer at first as some of our math was off in our calculations. In addition a large portion of our issues came from movement and collision as those had a lot of moving parts in the code and took large amounts of time to fix frustratingly simple problems.

All in all, while a stressful and time consuming project we were able to complete the project and get most of the base functionality of the game working properly. The following sections will explain how we were able to produce what we showed in the demo as well as how our thought process changed along the way based on what problems we had run into.

**Written Description**

Our rendition of the 1985 Super Mario Bros game by Nintendo included the basic functions to run the games. We had implemented drawing of the sprites on the screen to create a level background, the level floor, and the characters of the level. We included Mario, Goombas, and Koopa Shells. We unfortunately were unable to get enemy collision done in time as it started to break our working program when implementing it at the end so we had to scratch it last minute. However, the collision between Mario and the floor and blocks placed throughout the level was fully functional with a minor glitch which appeared minutes before the demo and we ran out of time to fix as he would bounce up and down a little bit. We presume that this was due to the range of values we used to which he was not allowed to go was too big and his height was just constantly resetting as he would fall down a pixel or two. In addition we were able to get Mario to be able to move around the screen while staying in bounds and jump around to the different blocks climbing up to the top floor. The original idea and or plan was to create a camera register that would allow us to scroll throughout the level from start to finish just as you see in the real game. However, we ran into an issue where past a certain point in the level it would break the game and just start to repeat the starting point. Thus, we had to scratch it at the end in order to maintain basic functionality for the demo. However, we were still able to have a frame buffer that worked properly to draw the screen in the blanking period to allow for constant updating of the screen and ultimately movement of mario and the other objects.

We barely used any C code as we decided it would be simpler to keep everything in SystemVerilog to avoid having to move information between the two. That way we were able to use local registers for everything, such as storing the level design and the color palettes for the sprites. In order to get the color palettes for the sprites we used Ian’s helper tools as it palletized the sprite images for us and allowed us to easily implement each and every sprite.

In order to draw the sprites and or objects on the screen we had an object register which held the sprite ID, position, and whether it was alive or dead. When drawing to the screen the sprite ID is used to grab the correct color palette and character to put on the screen as well as the X and Y coordinates to tell the system where to place it on the screen. Finally the last bit was used to check whether or not to print the character as it would either be dead or alive. The way we printed the objects was through a massive state machine that had a state for each object drawn on the screen. In between each state there was a wait state that would decide whether or not the next object should be drawn and if not it would skip the following state and go to the next wait state deciding if the next character should be drawn and so on. We realized later when the level design started to grow that this was probably not the best way to go about, however, it was too late in the game to change it. Thus, this leads to a lot of copy and pasting of a repetitive process that ideally coils have been done in a much better way. Although as I stated earlier with both of us not having the coding and COMPE background we did not think far ahead enough and or did not know a better way to go about in the beginning. We just kind of followed what we knew from earlier in the class and knew it would in theory work so at first there was not much thought about it causing issues later.

The last part that gave us issues implementing was using the keycodes from the keyboard. When trying to implement this feature we were able to follow the same process as in previous labs, but we tried to do it through platform designers and were unable to get it to work with our code at first. However, it was only able to read to the FPGA as it would display on the hex displays, but would not affect the code we have written for it. This is where we reached out to a TA that worked on it with us for a long while and we ended up getting it to work through changing a few things in the platform designer and the way we connected it. Ultimately, none of us had any idea why it was not working and or what was changed that truly fixed it. As it was one of the moments where everything looks right and should work the way it should have, but did not. At the end of the day this was one thing that was fully implemented in the end and worked just as planned.

**Final Implementation and Features**

Object Attribute Registers: These registers stored information about every object that needed to be drawn on screen. Each is a 40 bit register using 32 bits to store coordinate information for the object, 6 bits to store the type of object (used for accessing our sprite rom), and 2 bits to store status information about it (i.e. whether an object has been broken or killed).

Frame Buffer: This is a 241,920 bit RAM unit with two port access. This was used to store pixel information for the screen. The sizing was determined by using our resolution (252x240) and using 4 bits per pixel to store the index of our 16 color palette. We determined that as long as there were less than 100 objects on screen at once and we utilized two-port writing during the blank cycle, we only needed a single frame buffer to run the game at 60 Hz. It is written during the blank cycle based on the data stored in the OARs.

Sprite Rom: Our ROM is a collection of 16x16 ROMs that are independently instantiated, one for each type of block, enemy, or sprite that is implemented in the game. Larger sprites were broken up into 16x16 blocks for simplicity. The sprite ROM module is accessed with a single sprite index (chooses which sprite to reference) and rom address (chooses which pixel in the sprite). The floor sprite is the only exception to this because it must be drawn more than any other sprite, so we gave it its own access.

Movement: Movement was accomplished by changing the coordinates stored in the player character’s object register. We used constant speed vectors for the x and y coordinates that were added to the object’s x and y coordinates whenever the corresponding keycode was detected. Jumping could only occur when colliding with the ground to avoid flying. When not colliding with the ground, gravity was implemented by adding the same constant value to the object’s y coordinate until the player stood on an object. Additionally, two button presses were usable at any time allowing the player to jump and strafe independently.

Collision: For demo purposes, collision detection was created by checking the coordinates of the player and comparing it to each of the on-screen objects. When the player coordinates were directly above any object or the floor, a collision down flag was raised that stopped the player from moving down any farther. Collision up was similarly created. Left and right collisions were not implemented due to time constraints.

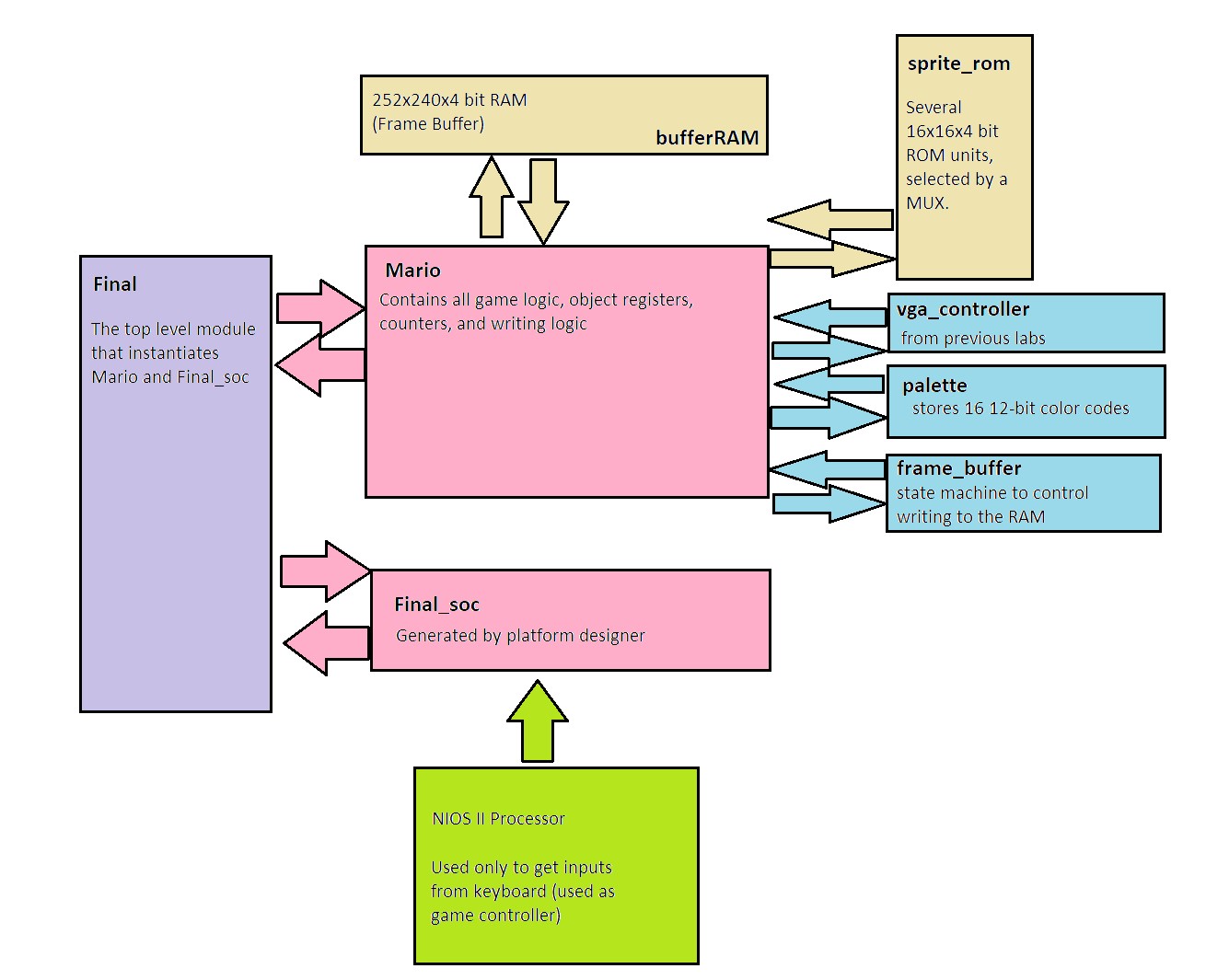
**Partially Implemented Features**

Screen Scrolling: This was only partially working by the end of our project. This was done by utilizing a camera register named cam\_REG that stored the x-coordinate of the screen. When the player moved to the halfway point of the screen and continued moving right, the screen would have the same movement value added to it, allowing it to move with the player. All objects were drawn on screen based on their x coordinate using the camera coordinate as an offset, so that they moved with the camera. If an object moved off screen, it would stop being drawn entirely. Unfortunately, a bug was causing all objects to disappear from the screen once screen scrolled about 126 pixels (half the screen length). For this reason, it was scrapped to focus on other functionality before the demo.

Dynamic Collision: This is a different way to detect collision by only checking the objects shown on screen to save time and resources. The objects are each drawn one after another based on a state machine. The idea was that when each object is drawn (which is only done if the logic detects it will be on screen) the coordinates are compared to the player character coordinates and a collision flag is raised. The collision flags are reset at the beginning of the draw cycle (once per frame) and are otherwise not reset so that the object’s do not overwrite each other. A bug was not stopping the player character once the collision was detected, though, so we reverted to the simpler collision detection method for a lack of time to bugfix the improved version (which would only be needed if scrolling were also implemented).

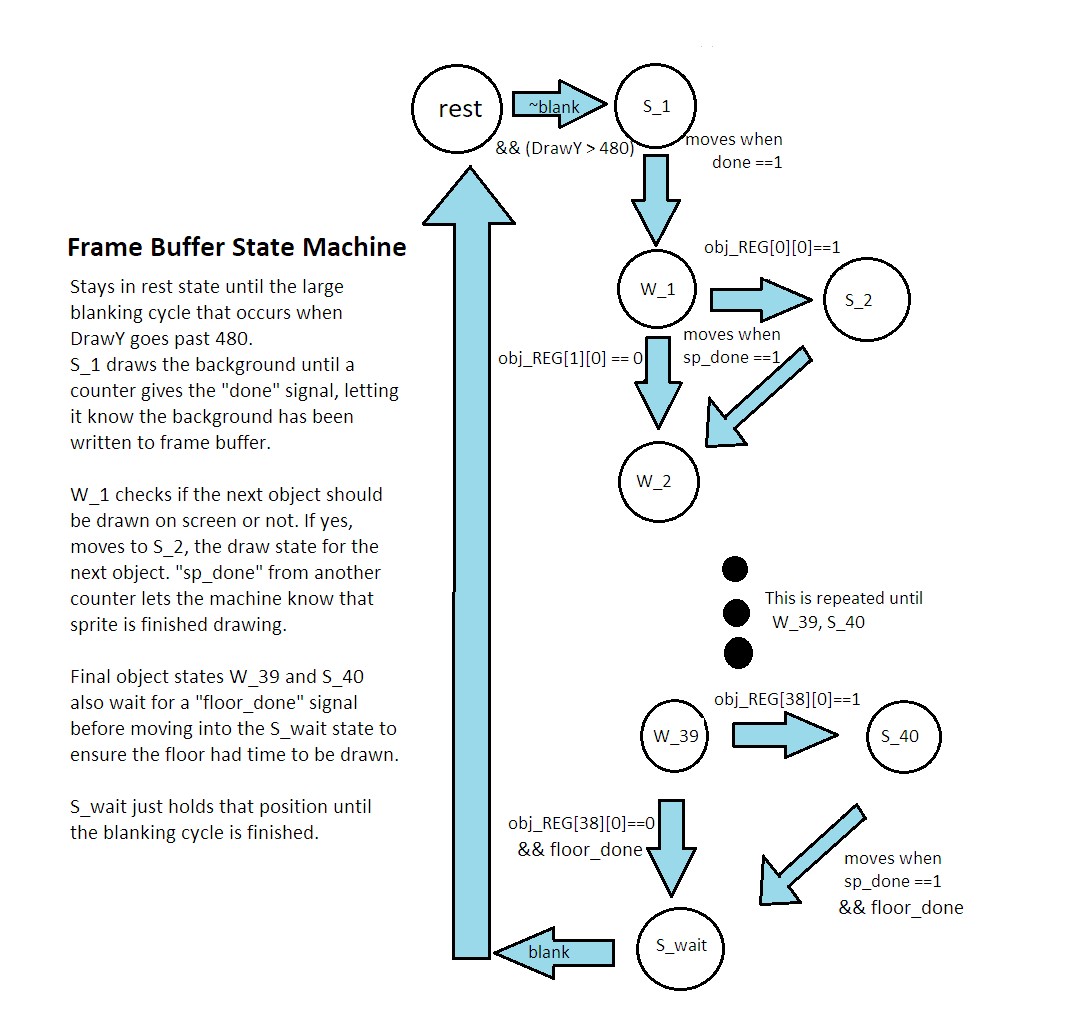
Enemy movement: This was accomplished similarly to the player movement by adding to the object register for each moving object (goombas and other enemies). We tried to finish this hastily before our demo, but a bug was causing the enemy to move underneath the ground once moving a certain distance. Additionally, if the player collided with the enemy from above, the enemy would be deleted from the screen by changing one of the status bits in its object register, however this same bug was preventing the collision from occurring, so that was not visible in the demo.

**Diagrams**



**Figure 1: System Block Diagram**

This diagram shows the layout of the system components. For a description of the inputs and outputs of each component, please refer to the module descriptions section. The “Final” module on the left is the top-level component of the system, though the majority of the work done on the final project was within the “Mario” module and the components connected to it.



**Figure 2: State Machine**

Note: This state machine shows the number of states that were implemented by the time of the demo. The number of object state pairs can be scaled up to several hundred without issue as long as no more than 100 are on screen at any one time.

**Module Descriptions**

Module: Final

Inputs: MAX10\_CLK1\_50, [1:0] KEY, [9:0] SW, [15:0] DRAM\_DQ, [15: 0] ARDUINO\_IO, ARDUINO\_RESET\_N

Outputs: [9:0] LEDR, [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, DRAM\_CLK, DRAM\_CKE, DRAM\_ADDR, [1:0] DRAM\_BA, DRAM\_LDQM, DRAM\_UDQM, DRAM\_CS\_N, DRAM\_WE\_N, DRAM\_CAS\_N, DRAM\_RAS\_N, VGA\_HS, VGA\_VS, [ 3: 0] VGA\_R, [ 3: 0] VGA\_G, [ 3: 0] VGA\_B

Description: This is the top level module for our system. It serves to receive the system I/O and instantiate our soc and Mario game module. It also contains the interconnections between them.

Purpose: This module connects all of our other high level modules together. It also instantiates some HexDriver modules that we used for debugging.

Module: HexDriver

Inputs: [3:0] keycode

Outputs: [6:0] HEX

Description: These are the same drivers used in earlier labs.

Purpose: These were used to check our keyboard inputs for debugging purposes with the controller.

Module: Mario

Inputs: CLK, RESET, [7:0] keycode, keycode1

Outputs: [hs, vs, [3:0] red, green, blue

Description: This module contains all of our game logic, object registers, and frame buffer writing logic. It also instantiates all other system modules within it. For more information, please reference the written description section above.

Purpose: This module contains the vast majority of the logic used to run the game as well write and read from the frame buffer.

Module: palette

Inputs: clr\_idx

Outputs: r, g, b

Description: This is a simple 16x12 register which stores the red, green, and blue values for 16 different colors used in game. Only a single color value can be accessed at a time, which is all that is necessary since reading from the VRAM is done a single pixel at a time via DrawX and DrawY.

Purpose: Utilizing a color palette frees up memory from the frame buffer since we now only have to store a 4 bit index value for each pixel rather than the 12 bit rgb value for each one. It also simplifies the sprite rom for the same reason.

Module: sprite\_rom

Inputs: [5:0] char\_idx\_a, char\_idx\_b, [7:0] rom\_address\_a, rom\_address\_b

Outputs: [3:0] rom\_q\_a, rom\_q\_b

Description: This module contains the instantiations of several smaller roms, one for each 16x16 sprites and initialized from its own memory initialization file. The character index inputs are used in muxes to select which sprite is being read from. The rom\_address inputs select which pixel of the 256 composing each sprite is being selected. The b addresses were initially used to essentially make this a two port rom, but they were eventually changed to permanently be used for the floor sprite, since it has to be drawn approximately 32 times per screen on its own.

Purpose: This module stores all sprite information and allows us to access it when writing to the frame buffer. It is scalable to any number of sprites as they are added.

Module: framebuffer

Inputs: CLK, RESET, blank, sp\_done, done, floor\_done, [9:0] DrawX, [9:0] DrawY, [39:0] obj\_REG [101], [15:0] cam\_REG

Outputs: LD\_bgd\_all, LD\_fgd\_all, LD\_fgd\_sprite, wren\_a, count\_start, sp\_count, [7:0] object

Description: This module contains the state machine used to control writing to the frame buffer. It begins in a rest state, then goes to the background drawing state once it reaches the blanking interval. A counter outside the module makes sure enough time has elapsed for the previous screen’s pixels to be cleared, then it sends the done signal to let the state machine know it can move to the next state. Every other state is actually composed of two state pairs. In the first state, a wait state, the machine checks if the next object is on the screen based on its coordinates compared to the cam\_REG. If it’s not, it skips to the next wait state, and if it is, it goes to the draw (S) state for that object. All 101 objects we implemented have two states for this reason. At the end of the state machine, the floor\_done signal is checked before moving back to the rest state to confirm the floor is finished drawing.

Purpose: The signals generated by the state machine go back to the Mario module and are used to tell what object to draw and when to draw it into the frame buffer RAM.

Module: VGA\_controller

Inputs: Clk, Reset

Outputs: hs, vs, pixel\_clk, blank, sync, [9:0] DrawX, DrawY

Description: This module is unchanged from the version used in labs 6 and 7. It uses the system clock to generate the sync signals, a half-speed clock, and the draw values necessary to output to the VGA display.

Purpose: This module is necessary to create signals needed by our system to know where on the screen a pixel is being drawn, as well as creating the horizontal and vertical sync signals needed by the VGA output.

Module: Final\_soc

Inputs: accumulate\_wire\_export, clk\_clk, [1:0] key\_external\_connection\_export, reset\_reset\_n, [15:0] sdram\_wire\_dq, spi0\_MISO, usb\_gpx\_export, usb\_irq\_export

Outputs: [15:0] hex\_digits\_export, [7:0] keycode\_export, [7:0] keycode1\_export, [13:0] leds\_export, sdram\_clk\_clk, [12:0] sdram\_wire\_addr, [1:0] sdram\_wire\_ba, sdram\_wire\_cas\_n, sdram\_wire\_cke, sdram\_wire\_cs\_n, [1:0] sdram\_wire\_dqm, sdram\_wire\_ras\_n, sdram\_wire\_we\_n, spi0\_MOSI, spi0\_SCLK, spi0\_SS\_n, usb\_rst\_export

Description: This is the soc generated by the platform designer. It primarily contains the IPs necessary for interactions with the NIOS II processor.

Purpose: This is necessary because our system uses the NIOS II processor to read inputs from the keyboard we use as a game controller.

**Design Resources and Statistics**

| **LUT** | 4107 |
| --- | --- |
| **DSP** | 0 |
| **Memory (BRAM)** | 273408 bits |
| **Flip-Flop** | 2741 |
| **Frequency** | 70.05 MHz |
| **Static Power** | 96.56 mW |
| **Dynamic Power** | 21.40 mW |
| **Total Power** | 188.66 mW |

**Table 1: Design Statistics**

**Conclusion**

In the end this project was fun and interesting as it really allowed us to be tested in terms of our coding abilities and ability to think through the creation of a video game. It was awesome to get a real idea of what goes into creating these games and the ones that you play on your computer or xbox and such. As we all created relatively basic and simple games with basic game logic and movement. We could only imagine what kind of work goes into the games that are created nowadays for current consoles. Watching other people play them now and playing them ourselves really makes you look at the game in a whole different perspective as you try to think of how they were able to add that feature and the different ways that they could have gone about creating that. It is truly amazing the kind of work that goes into them and the possibilities of it.

This project and this class is yet another way that this university has given us insight into the real world of how technology started and how it has evolved throughout the years. It really helps you think about the development aspect of it instead of following instructions and doing as you are told. The portion to which we found most enriching is to think for yourself, but get help when you need it as an aspect of the class. As there are numerous things you can do on the FPGA and even more ways to implement each idea. It was a real test of ability as we had to utilize the resources that have been provided to us throughout the semester and really decide for ourselves how we wanted to organize and implement each and every feature.

I would say the real learning started when we started to run into problems where certain things did not work the way we thought they were going to and did not realize till it was too late. Thus, leaving us sitting there and thinking about how we should have done it this way instead. Either that or it came to our realization that there was a much simpler way to do a certain aspect, but we over-thought it and made it much more complicated than it needed to be. Also we often started a portion of the project forgetting to include what we had added to it later meaning we had to go back later and readjust previous work to fit the new additions.

Overall, this course and this project heavily assisted us in creating the engineer mindset that is required to survive and succeed in the industry. As we all know there are people who are good at memorizing material and following directions and then there are those who have the mind of an engineer and think why and how something is going to work. Thus, this course is something that has helped develop and reinforce that engineering mind in ourselves. As we often started to understand the difference between possible and feasible in terms of time and ability. At the end of the day there was a lot gained from this course and this project and we want to say thank you to the course staff as none of us would have succeeded without you and the class was a lot more manageable because of you.