6.111 Project Proposal

Mario Bros Classic in Real Life

Jose Guajardo, Nancy Hidalgo, Isabelle Chong

System Overview and Block Diagram:

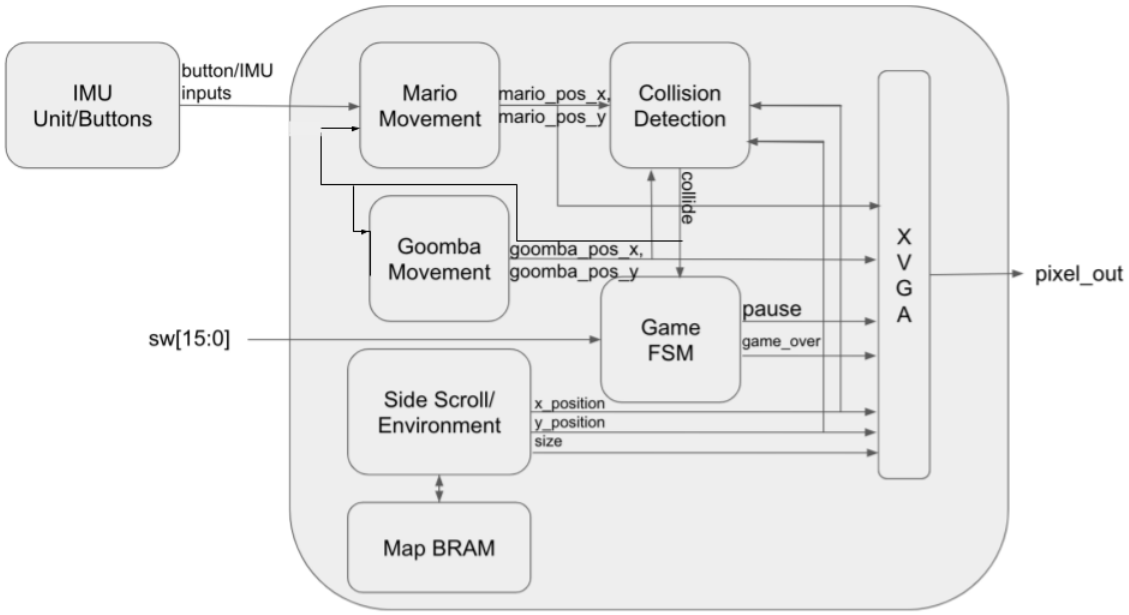
For our 6.111 final project, we will recreate the side-scrolling Mario Bros Classic game for Nintendo with several added functionalities. One of the major challenges we expect will be storing and loading the graphics of the level as Mario moves. Additionally, we will be implementing interactions between Mario and his environment using collision detection.

We will program this project on the Nexys4-DDR FPGA using the XVGA display to host the game graphics. We will also make use of some of the internal/external memory to load the map and sprites onto the display. The initial goal of the project is to create a level where Mario can jump over and stomp Goombas to complete the level.

We are also thinking about possible add-ons for our game. Instead of using buttons to move Mario around the screen, the player will make Mario perform actions by performing them in real life with an IMU controller with at least two IMU sensors. For instance, to make Mario jump, the controller should be accelerated upward. To make Mario move in a direction, the IMU controller can be tilted in either direction.

Simplified Overall Block Diagram:

A block diagram with modules needed and simplified inputs:



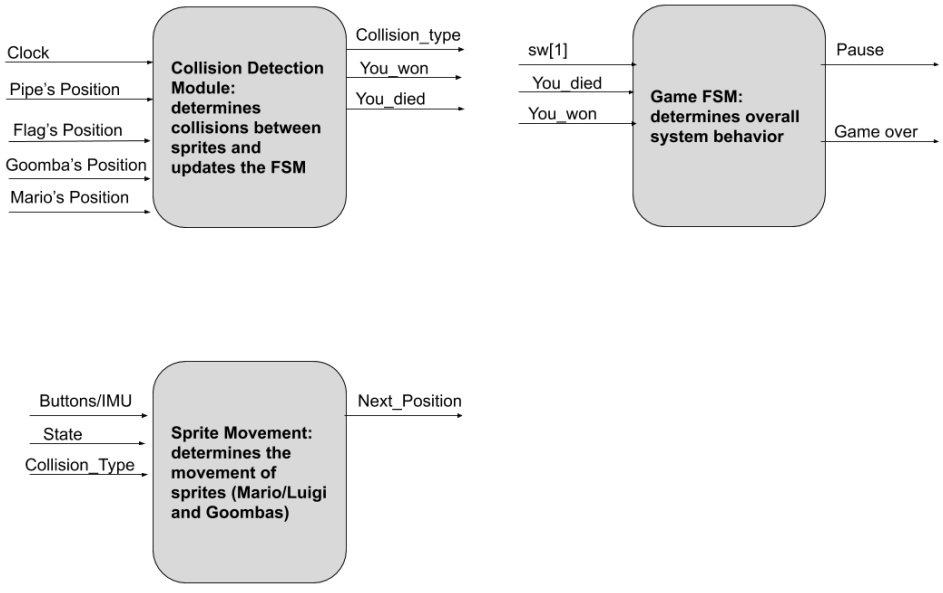
Collision Detection, Sprite Movement and Game State FSM: Nancy Hidalgo

The game mechanics will be primarily controlled with two modules: a Sprite Movement module and a collision detection module. The sprite movement module will dictate the movement of two different types of sprites: the Mario/Luigi sprites and the Goomba sprites. For Mario/Luigi's movement, the module will take the input of the buttons/IMU make Mario/Luigi jump and 'move' to the right/left (in actuality, the background will move left/right). For the Goomba's movement, they will constantly move at some speed, then flip directions when they run into a pipe. This module will also take the output from the collision detection module.

The collision detection module will determine whether there is a collision between sprites and then update the overall game FSM. This module will take the positions of the different sprites as inputs and output whether there was a collision, whether the player lost a life (by colliding into a Goomba or falling into a hole), and whether a player has won (by 'colliding' into the flag at the end of the level). This module will be the most complicated of the three that I'm making.

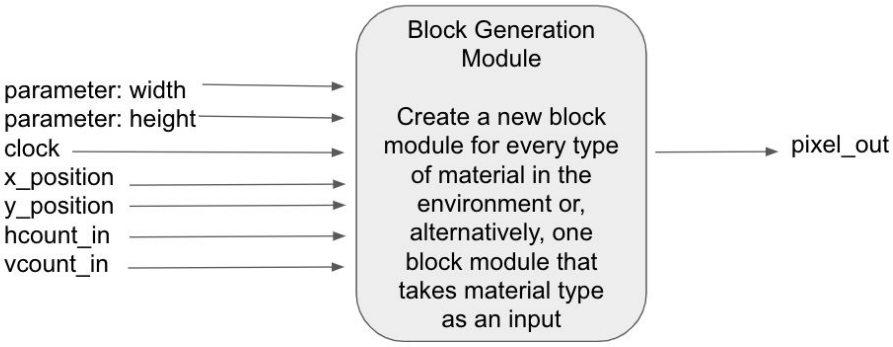
The Game state FSM will be a simple FSM consisting of 4 states: a PAUSE state where the player has pushed a button to pause the game and no sprites move, a GAME_OVER state where the player has lost 3 lives and the screen will display white "GAME OVER" text over a black screen, a START state where the game has just started or been restarted after a player has lost, and a GAME_PLAY which would be the default state, where a player is just playing the game.

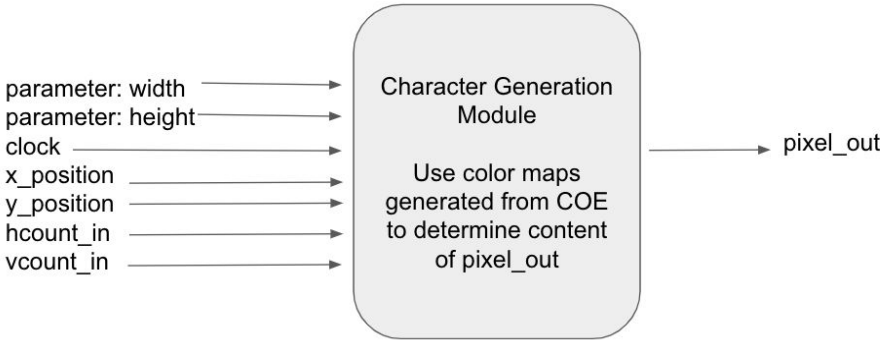
The Game FSM and the sprite movement modules each have outputs that are inputs to the XVGA module which then outputs to the screen. Therefore, combined with the XVGA module, they have to have a throughput faster than the screen refresh rate (60 Hz).



Sprite Generation: Izzy Chong

The generation of sprites within the game will be done using COE files, similar to how the Death Star blob was generated in lab 3, and pure pixel blobs, similar to the simple rectangle blob also done in the pong game. Characters like Mario and the Goombas will be generated using the COE files, while environmental elements will be generated as “blocks,” or squares of pixel “material,” such as “ground,” “pipe,” or “cloud.” This allows the game setting to take up less storage in memory because rather than having to store multiple pixels, we can store a reduced number of block locations. Each sprite (Mario, Goomba, blocks of different materials) will get their own module. Typical inputs would be parameters for width and height, clock, x_position, y_position, hcount_in, vcount_in, and pixel_out.

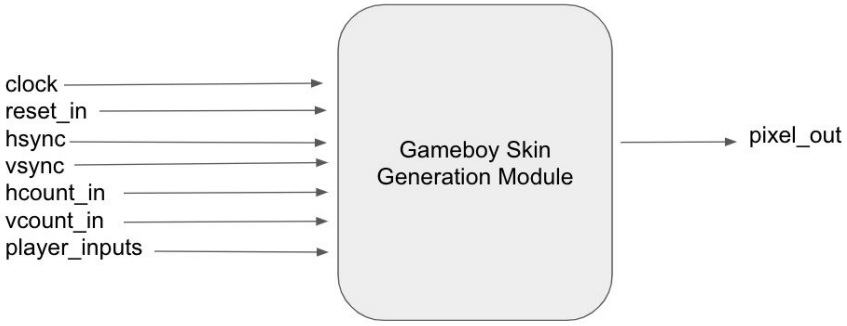




We will also include a Gameboy device skin to be displayed on the FPGA around the gameplay screen. This will help enhance user experience while also lowering the amount of memory needed to generate the side-scrolling effect of the game by making the number of pixels stored smaller. The Gameboy skin will be constantly displayed and will highlight player actions in real time on the screen (e.g., if the player makes Mario jump, the corresponding button on the gameboy skin will indicate being pressed), and will not need to be stored in memory. To create this device skin, we will create a draw_gameboy module which will take as input a clock, hcount_in, vcount_in, hsync, vsync, and player actions and output a correspondingly altered gameboy design through pixel_out. This module will update the

gameboy screen once every full raster of the XVGA.

Because these modules are visually-based, the majority of testing for them will involve synthesizing the modules to hardware and checking if the necessary blocks appear on screen.



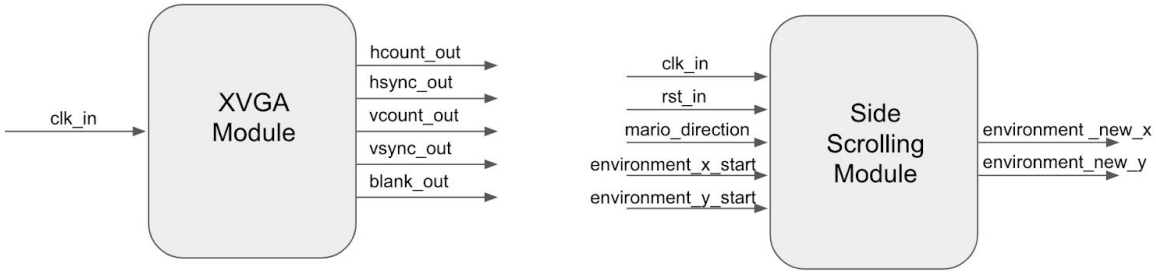
XVGA, Map Memory, Side Scrolling: Jose Guajardo

We plan to use an XVGA module similar to the one used in the pong game lab. In the pong game lab, we became familiar with generating, displaying, and moving objects through XVGA. However, in order to implement out Mario Bros. game, we will have to add additional functionality.

We expect the side scrolling game functionality to be a significant challenge for our project. In the pong game lab, we became familiar with generating, displaying, and moving objects through XVGA. We will be adding several functionalities to allow for side scrolling. Firstly, in order to load the map as Mario progresses throughout the level, we will have to store the starting location and size of each object in the environment (clouds, ground, Goomba) in BRAM. Additionally, we will need to keep track of the location of the player in relation to the entire level in order to determine which sprites to load onto the screen as Mario approaches them.

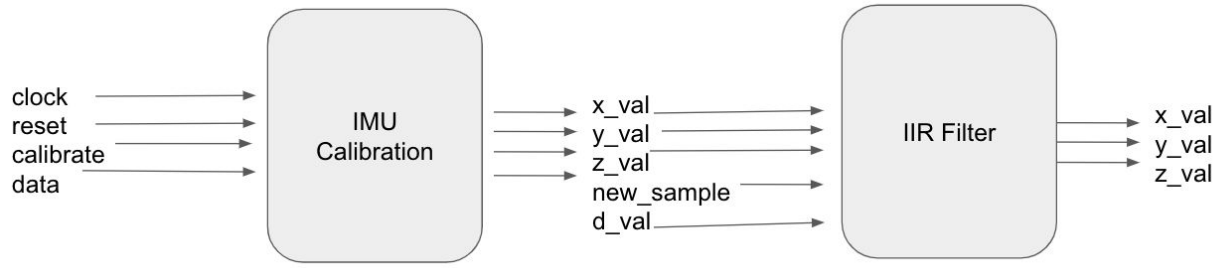
In general, our approach will be to move all objects in the environment in the direction opposite to the player's inputted direction. That is, if the player wants Mario to move right, all objects in the environment will move left, while Mario stays at the center of the screen. As Mario moves, he will be interacting with his environment and its collision boxes. The first iteration of the side scrolling module will have functionality in the forward direction. Then, we will implement side scrolling while moving backward. Eventually, the side scrolling module will connect to the Mario movement module, which provides information about the direction Mario will move in depending on his collisions. Then, we can decide if the map should scroll forward or backward.

In order to implement side scrolling functionality, we will read from BRAM with an address that represents the location of Mario in the level. At that address, the location and size of any objects in view will be saved and accessed. Once we know which objects should be present on the screen, we can generate their sprites using the sprite generation module, which will output them to XVGA.



IMU Implementation: Izzy Chong

We plan to implement an IMU controller that allows players to tilt the IMU and make Mario jump and move left and right. This is similar to lab 5b, so much of the code for this portion can be taken from that implementation. We will read IMU data in through the Teensy and use those acceleration measurements to control Mario's motion. Tilting the IMU around the X axis will direct Mario to move left and right, while tilting it around the Y axis (or, alternatively, lifting it up in the Z direction) will direct Mario to jump. This method will use an IIR filter, similar to that used in lab 5b, to prevent Mario from "jittering" too much, but with the d value hard-coded in.



To test this module, we could create a test bench with dummy signals to make sure the IMU is reading in data in the proper cycles (i.e., waiting for data at appropriate times, reading at appropriate times).

Miscellaneous Add-Ons:

If we are able to complete the material already listed in the proposal, we could begin working on the following add-ons to improve the quality of our game:

- Adding nostalgic music and sound effects
- Ground pounding/going into pipes
- Including a green screen avatar of the player in the game so they can play as themselves instead of as Mario.
- Two player mode
- Making Mario look like he is running when moving