III. Map generation (Macade Walker)

1. function

For the game to work, we needed some function that creates the various obstacles for the player. One thing I could have done was just make a large binary number and have that represent our map of obstacles, but that would not only be extremely tedious to actually type into the computer, it would also keep outputting the exact same map if they played multiple times, which could become boring for the player. So I went for a randomly generated approach.

B. Implementation

Instead of using the $random or $urandom commands, I instead used a counter that counts at the speed of the 100Mhz clock. This number is a 16 bit register, so we can use it as 4 random 4 bit numbers. Note that, although the register is named seed, it is not actually acting as a seed, it’s just a random number.

We use this 16 bit number to basically choose which output will be on the map. We decided that we wanted 3 different types of outputs for each section of the map: empty, bottom block, and top block. This means we need 2 bits to represent each spot on the map. There are 8 segmented displays on the fpga, so we need 16 bits to represent the map.

To break down the Always block that starts on line 94 (the largest part of the program), every time the positive edge of the divided clock occurs, we shift the map to the left and add the next section to the right 2 bits of the map. The majority of the block is spent figuring out what that next map section is supposed to be. Basically, the next part of the map is whichever part of the preload we have not put into the map yet, and when there is not any preload left, the seed determines what the next preload is.

C. Problems

One problem I initially had with this design was that it had to be fair. For example, if there were a bottom block followed by a top block on the next display, then the player would lose whether or not they jumped. This drove me to make it so the seed decides between a number of different modules that all include spaces and a block, which ensures there are enough spaces between blocks so the game does not feel too unfair. I got the idea from a game called Spelunky, which is a game where the caves explored are randomly generated in a similar way, where each portion of the map is chosen between several different possible blocks which guarantees the game is fair and no impossible levels are created.

Another problem I had when testing was the occurrence of loops where there are no bottom jumping blocks created by the seed. When this happens, there is no reason for the player to jump, and the jump input is the only way the seed would change. To fix this, I had the program check to see if there is a bottom block on the map before it determines a preload. If there is not a bottom block, the preload becomes a bottom block, which guarantees the player will jump and the seed will change.

For bibliography: https://www.eecs.umich.edu/courses/eecs270/270lab/270\_docs/debounce.html