**Design Description**

**Pacman.v:**

**Board\_state.v**

|  |  |
| --- | --- |
| Input | Output |
| * clk * reset * btn\_up * btn\_down * btn\_left * btn\_right * x * y | * score * game\_over * board\_data |

The aim of this module was to maintain a two dimensional board and update it at every positive edge of the clock that its fed in. The inputs to the module are a 2 Hz clock, a reset signal, directional controls for the user, and an x and y position to render the graphics. The directional controls for the user move the pacman player, up, left, down or right if it is possible. The x and y inputs to the module are to get the state of that grid location, so that the correct graphics can be rendered for that board position on the monitor.

The game logic behind the game board is as follows. Each grid location can be one of 5 possible states:

1. Empty Box
2. Wall Box
3. Pacman Box
4. Ghost Box
5. Food Box
6. Ghost and Food Box

When the module is instantiated, the board is initialized to a hardcoded game map. The walls are instantiated according to our design on the 6x8 game board. After that, every empty box is set to a food box. Next the pacman and the ghost are placed in opposite corners and the game begins.

If the reset signal is high, the above instantiation steps are repeated and the game begins again. If the game over signal is high or the player’s score has reached the maximum it can, the game ends and we display an all black screen to show this.

Otherwise if the ghost hasn’t eaten pacman and pacman hasn’t eaten all the food on the game board, we update the game board. To update the player, we check the button up, down, left or right signals. For instance, if the player is pressing the button up button, and the grid location above the player is within the bounds and not a wall, we move the player there. We then update the board correctly. If the grid there was a food box, then we increment the player’s score. If the player tries to move to a box where the ghost is, the game ends, and game over is set to true. It works similarly to this for the left, right and down input signals.

The ghost move logic works similar to this. Instead of getting an input for which direction to move in, the ghost tries to move in the UP direction initially. The ghost takes two steps in this direction if it can, and then tries to switch direction. If it can switch direction, it starts moving in this new direction and repeats the process of taking two steps and then switching direction. If the ghost can’t move in a direction, then it switches directions before reaching the two-step requirement.

The ideal ghost movement that we were going for involved changing the

ghost direction based off of the player position. However, due to getting bugs in the current ghost moving algorithm, we weren’t able to implement that.

**Clk\_Divider.v**

|  |  |
| --- | --- |
| Input | Output |
| * clk * reset | * 25 MHz clock * 400 Hz clock * 2 Hz clock |

The aim of this module is to generate different clocks from the 100 MHz clock provided by the hardware. The project required a 2 Hz clock for updating the game state, 400 Hz clock for the seven segment display and a 25 MHz clock for the VGA display. To generate these clocks, we created a counter that counts up to the respective values. For instance, for a 25 MHz clock, we want the clock signal to flip once for every four times the 100 MHz clock flips its output. Hence, we count up to 2 using the 100 MHz clock, incrementing the counter at every positive edge, and then flip the signal of the output clock. Similarly, we define different counters to get different output clocks.

**Simulation and Documentation:**

Clk\_Divider.v:

We wrote a test bench to check the clk\_divider.v module. The testbench simulated a 100 MHz clock, which was our input. To test the 25 MHz clock, we looked at the waveform and observed one oscillation in this signal for every 4 100 MHz oscillations. For the lower frequency clock (2 Hz), we measured the period of the signal from the wave window. This waveform is shown below. The exact frequency of the 400 Hz clock wasn’t too important since it only had to display the digits continuously on the seven-segment display. We noted that this signal was oscillating at a pretty frequency when compared to the 2 Hz clock. Furthermore, on an initial synthesis to hardware, the digits on the display were being displayed continuously, which gave a positive indication that we were using the correct frequency of this signal.

Board\_state.v: