EPO-3 Group B2 Module report objects and movement

December 7, 2018

Sam Aanhane	*4644581*
Joost van der Weerd	*4704320*

TU Delft

Abstract

This report will cover the Objects and movement module. A clear definition of all inputs and outputs will be provided. Using this information, FSMs can be created and explained in order to work out what VHDL code is required to achieve the appropriate behaviour for the game to function. A testbench will be analyzed in order to provide a detailed explanation whether everything functions accordingly.

Contents

1 Introduction

Designing a video game is a lengthy process. When creating a product with the end-goal of providing entertainment, plenty of things need to be taken into account. This gives rise to careful planning, as such a complex product can easily dysfunction. Graphics are not enough to make a game enjoyable. The gameplay is important too. In order to make something truly immersive, the game needs to be responsive, challenging, and balanced. These three concepts are detrimental to the success of the game and therefore require their own module. Our game, based on the NES classic gradius, features a spaceship, controlled by the player, which is able to move and shoot at enemies coming towards it. The aim of the game is to score as many points as possible by surviving for as long as possible. The control of the player's movement and the control of all game-objects like enemies and bullets, are all treated and optimized in the same sub-block, later refered to as the objects and movement module.

The objects and movement module is at the heart of the design. It is responsible for most of the logic in the game. After all inputs have been buffered by the controls module 1.1, they are received by the objects and movement module. These inputs include whether the player should move and if it should shoot. Next, several processes are initiated:

1. When one of four movement buttons is pressed, the player's next position needs to be calculated. It achieves this using a counter for the horizontal and vertical position. This is handled by the movement block.
2. When the player wants to shoot, a bullet should leave from the position of the ship, and travel straight to the right.

A different mechanism, unrelated to the input of the player, calculating the next position of different types of enemies is also calculated in this block. Implemented using horizontal and vertical counters, these positions are determined.

The reason why the objects and movement module is at the heart of the system, is that its output is the source of all other modules. The algorithm deciding where to insert the enemies in the screen uses pseudo-randomness relying on the player's position. As its name implies, the collision logic requires the position of the bullets and the player's ship to determine whether either one comes into contact with an enemy. Lastly, though elementary, in order for the display module to know where objects need to be drawn, it requires all objects their position.

In order for the final product to function correctly, it is important that all modules have the same end goal in mind. A well-defined design is therefore crucial to the success of the game. Of utmost importance is that all modules have clearly defined in- and outputs. That is why the module's specifications and design will be thoroughly described and illustrated in the following pages. Based on the results of the initial module design, a conclusion will be drawn whether it is fortunate to continue the design, or to start over due to problems related to functionality or memory issues.

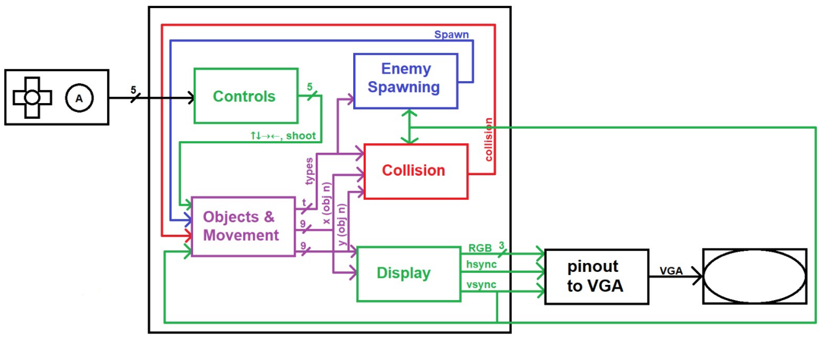

Figure 1.1: Overview showing all modules and the most important signals. The clock signal has been excluded because every module receives it.

2 Specifications

As discussed in the introduction this modules main function is to keep track of the positions and the movement of these objects. These two functions are coupled together for simplicity since movement is just a update in the position. In the game we have three different types of objects namely: the player, the bullets, and the enemies. These will all have a different specifications and behaviors. So we will discuss them all separately. But for every component all the actions are synchronous on the clock and respond to the reset to fall back to the reset state, so all the components have the clock and reset signals as inputs and we will not further discuss this.

2.1. The player

The player is controlled by a person at the outside of the game thus inputs of the player are at least the controls. In this game the player should be able to move in all 2-D directions so up, down, left, and right so it will have 4 one bit inputs move up, move down, move left, and move right. Also it should be able to shoot at an enemy. But because there can be multiple bullets in the screen at the same time and they cannot be shot again until they are out of the screen there needs to be some logic controlling the bullets and this will not happen in the player itself but in a separate component. This will be elaborated on in section (2.3). The output of the player should be the position of the player so the display module can draw the player in the right place on the screen. Since our screen has a resolution of 320x480 the x- and y-coordinates as output are both 9 bits to makes sure it can be in the whole screen. Now lets look at the behavior of the player. The player should move according to the inputs so if the move right button is pressed the player should move right and if the move left button is pressed it should move left, and if both or none are pressed the player should not move in the x direction and of course the same goes for up and down but then in the y direction. Now we have to relate this to the output of the player. The output of the player when the button move right is pressed the x-coordinate of the output of the player should increase with one and when the move left is pressed the x-coordinate should decrease with one. The same applies for the y-coordinates with the move down and move up buttons. By handling the x- and y-coordinates separately the player can also move diagonal.

2.2. The bullets

Each bullet is controlled by bullet control and the collision module it has as input a one bit shoot signal, two 9 bit signals of its x- and y-coordinates, and a collision signal. The output of the bullet is two 9 bit signals of its x- and y-coordinates and a 1 bit signal of its state (bullet in the screen or out of the screen). The bullet has a simple function when the shoot signal is high the bullet spawns on the position of the player and starts moving right. This is done by increasing the the x-coordinate. When the bullet is in the screen it will have a high output of its state indicating to the logic that it is in screen and thus can't be fired again.

2.3. Bullet control

As mentioned in section (2.1) the handling of the bullets is done in a separate component. This is done because there is only limited space on the chip so there is a maximum of objects we can have. For the bullets this means that there can be only three bullets at the same time in the screen. Since we are reusing the bullets there needs to be some logic that when a bullet is in the screen it can't be shot again until it outside of the screen, because otherwise the bullet would keep skipping back to the player every time you shoot. For simplicity of the bullets this logic is done in super ordinate component. This component has a one bit input of the state of each bullet, so 3 one bit inputs. Also it will have the 2 nine bit x- and y-coordinates of the player as input since the bullets have to begin at the player. And last but not least the control input of the shoot button, also a one bit signal. The output Now lets look a the behavior of this control component, its basic objective is when the person playing the game presses shoot that there should spawn a bullet at the player. But we don't want a bullet which is already in the screen to spawn at the player. So this control component should keep track of which bullets are in screen, done with the three inputs of the bullet states. And it should fire a bullet which is not in the screen. It should also make sure that there is a fire delay to make sure that when u press the button once it wont spawn all three the bullets at nearly the same time.

2.4. The enemies

There are 3 types of enemies and because their behavior is different they will be discussed separately. The first enemy type is the most basic when it is spawned in it will just move left from its position until it collides with something. The second enemy type is almost the same except that when it moves left it also moves up and down creating a zigzag path. The last enemy is the most complex since when it moves left if follows the player in the y direction. When the player is above the enemy it will go up and when the player is lower than the enemy it will move down.

The first and second enemy type behave in almost the same manner and have the same in- and outputs so we will discuss them together. The enemies have a, same as the bullets, super ordinate component controlling when and where they spawn, this is the spawn module, so we will not discuss the workings of that module in this report. But it gives us one of the inputs which each enemy has, the one bit spawn signal. Giving information on when to spawn. The enemy also needs to know where to spawn for this it has 2 nine bit inputs for the x- and y-coordinates. In addition to this each enemy has a one bit collision input giving information on when to die. The enemies have to be reused as in the same manner as the bullets so for output they have a one bit signal indication if it is alive or not. Also the position of the enemy will be an output, so 2 nine bit signals for its x- and y-coordinates. Since the third enemy type behaves according to its relative position to the player the enemy needs the 2 nine bit inputs of the x- and y-coordinates of the player as input as addition to the inputs the first and second enemy types have. But the output stays the same.

2.5. Overview

No we have established what every sub-block has as in- and output signals we want to make it more accessible so we made a diagram showing all the sub-blocks and their in- and output signals. This overview can be seen in Figure (2.1). The coloured signals are multiple of the same width and type signals to keep the figure accessible.

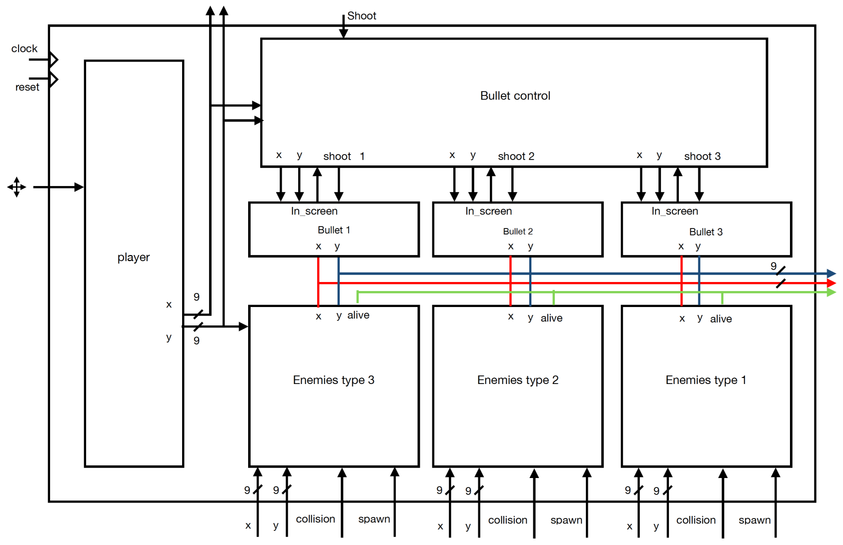

Figure 2.1: Overview specifications objects and movement

3 Design

Now that it is well-established what signals the module can expect and what its outputs should be, the foundation has been created for the design, though its far from finished. Based on the design specifications, the ability arises to divide the module into sub-blocks in order to more easily guarantee it handles everything as it is supposed to.

3.1. The sub-blocks of the objects and movement module

A good place to start the division is to define what encompasses the objects and movement module. As its name implies, it is responsible for the objects and the player's movement. This brings us to the main sub-blocks of the system:

1. The player's movement. Arguably the most important object in the game, an FSM will have to be created in order to accommodate the player's inputs in order to move the ship or shoot its bullets.
2. The bullets. The three bullets will be controlled by the player, so can therefore be assigned a sub-block. It will be responsible for relating the player's position, shoot signal, and number of bullets shot to three vectors defining the bullets' positions.
3. The movement of enemies. After the module has relayed the player's position to the module responsible for determining the enemies' starting position which is in turn send back to the module, it is responsible for determining their next position. This is done in a separate set of FSM's, so it requires its own sub-block.

3.1.1. The player's movement

As mentioned before, the sub-block dedicated to the player's movement requires, apart from the reset and clock signals, a x- and y-control signal consisting of a 2-bit vector in order to perform the logic which is described by the VHDL code in appendix A.1. It features two FSMs which can be found in appendix A.1 and A.2. One is responsible for the horizontal movement, and the other is responsible for the vertical movement. Both function essentially the same way. The only difference is the value on which they perform the arithmetic. In the starting state, the player is assigned a position in the FSMs according to the following table 3.1:

Orientation	Vector value	Pixel value
Horizontal	00111111	63
Vertical	01111111	127

Table 3.1: Starting positions of the player

Both FSMs then await a control signal. This determines whether the position vector is increased or decreased. A "01" signal initiates the right- or up-state, resulting in an increment of the value. A "10" signal results in the left- or down-state, meaning a deduction of the value. When either a "11" or "00" is received, the still-state is entered. In this state, the position value remains constant, because when either input is received the ship is not supposed to move.

3.1.2. The bullets

The bullets are a difficult part of the design. Not only do they require a fair share of the memory, they are also the most complicated objects. In fact, one sub-block is not enough to easily describe their behaviour. That is why another subdivision is possible. One VHDL entity will be responsible for the functionality of a single bullet. Another is required to manage the bullets, enforcing the bullet limit of three, and keeping their outputs separated. In appendix A.2 and A.3 the VHDL description of the bullet and the bullet controller can be found.

The bullet

The FSM describing the bullet behaviour can be found in appendix A.3. It contains four important inputs: the horizontal and vertical player position, a shoot signal, and a collision value. Based on these inputs, three outputs will be determined: the horizontal and vertical position of the bullet, and the state of the bullet. The latter is important for the bullet controller, and will therefore be explained later.

In the starting state, the initial x- and y-positions are defined. Those values are essentially trivial. The only requirement is that the values belong to a pixel coordinate outside of the screen. That is due to the requirement that, in order for objects to exist in the memory, they need to be initialed. They should not however be able to interact with other objects, and are therefore placed out of their vicinity. The state is changed to "placeinscreen" when the shoot signal becomes a '1', indicating that the player wants to shoot.

When a player shoots, the bullet has to leave from the current position of the player. That is why the "placeinscreen" state outputs the player's horizontal and vertical position as the current position. This automatically brings us to the following state.

In the final state, responsible for the movement of the bullet, the new position is determined. Because the bullets only move in the positive x-direction, the vertical position remains constant, indicated by the y_pos_new <= y_pos command. The horizontal position however is subjected to an increment with a constant value, resulting in movement to the right. Next, it awaits a collision signal. When this signal is high, it indicates that the bullet has either hit a target, or has reached the end of the visible screen. In both cases, the bullet as an object can be considered dead, and should therefore be removed from the screen. This is done by reentering the starting state, transporting the bullet outside of the screen.

The bullet controller

The responsibility of the bullet controller is to keep track of the amount of bullets in the screen and their state. According to this information, it decides whether one out of three individual bullet FSMs is allowed to fire. It uses the shoot input signal from the player, and the three bullet states produced by the individual bullet FSMs. These signals indicate whether a bullet is alive or not, allowing us to determine not only if a bullet is allowed to fire, but also which.

The VHDL code describing the controller can be found in appendix A.3. It shows the logic responsible for deciding when to fire which bullet. Table 3.2 shows all inputs and corresponding output.

Bullet number	Condition (the first three signals representing bullet_inscreen_1,2 and 3 respectively)	Output (the first three signals representing shootb1,2 and 3 respectively
No bullets #1	’0’ and ’0’ and ’0’ and shoot = ’0’	’0’ ’0’ ’0’
No bullets #2	’1’ and ’1’ and ’1’ and (shoot = ’1’ or shoot = ’0’)	’0’ ’0’ ’0’
Bullet 1 #1	’0’ and ’0’ and ’0’ and shoot = ’1’	’1’ ’0’ ’0’
Bullet 1 #2	’0’ and ’1’ and ’0’ and shoot = ’1’	’1’ ’0’ ’0’
Bullet 1 #3	’0’ and ’0’ and ’1’ and shoot = ’1’	’1’ ’0’ ’0’
Bullet 1 #4	’0’ and ’1’ and ’1’ and shoot = ’1’	’1’ ’0’ ’0’
Bullet 2	’1’ and ’0’ and (’0’ or ’1’) and shoot = ’1’	’0’ ’1’ ’0’
Bullet 3	’1’ and ’1’ and ’0’ and shoot = ’1’	’0’ ’0’ ’1’

Table 3.2: An overview showing all corresponding inputs and outputs

When the first shoot signal is received, the first bullet fires no matter what. This is done by sending "shootb1" signal to the first bullet FSM, from now on referred to as component. This activates the calculations, in the process sending feedback consisting of the "bullet_state" value. This signal consists of a ’1’ when the bullet has been placed in the screen, letting the controller know that the bullet exists. It needs to know these values in order to determine if another bullet can be placed. To summarize:

1. Bullet 1 will always be shot when it is not yet present in the screen when the shoot signal is received.

2. Bullet 2 will be shot when it is not yet present in the screen, and the first bullet has been fired.

3. Bullet 3 will only be shot when bullet 1 and 2 are already in the screen.

4. No bullet will be shot when all bullet are already in the screen.

The "shootb2" and "shootb3" signal correspond to the signals triggering the other two bullet components.

3.1.3. Enemy movement

The enemies in the game will make up for the other objects. They are relatively easy to incorporate, because their programming is relatively similar to the spaceship of the player. The main difference is that they do not require a continuous input to determine their next location. The VHDL code describing their behaviour can be found in appendix A.4. This sub-block, similarly to the bullet, could be divided further to incorporate the management of the four enemies. However, this has not been done for the moment being. The first reason is that this requires little logic also deemed elementary. The second reason is that this stage of development has not yet been reached.

The screen will contain a maximum of four enemies every cycle, which come in three types:

1. The first type is the easiest. After is has been placed in the screen, it will move at a constant pace and height towards the left according to the FSM in appendix A.4.

2. The second type will move in a bouncing way up and down towards the player at a constant pace according to the FSM in appendix A.5.

3. The third enemy will take the player’s position into account , and move towards it at a constant pace according to the FSM in appendix A.6.

The FSMs describing the enemies can be found in appendix A. All three are relatively similar. The most important inputs are the spawn coordinates form the enemy spawning module, the spawn signal itself, and the collision. The output consists of the new horizontal and vertical positions of the enemies, and a value describing whether they are alive or not, which is send back to the spawn module.

All three FSMs are nearly identical to the one describing the bullet behaviour. The only differences are the designation of the outputs, and output of the final state. Most comparable is the basic enemy. It is of type one. Therefore, instead of incrementing the horizontal value, as we did with the bullets, we have to reduce the position value, resulting in constant movement to the right. This is done until the collision value becomes ’1’, indicating one of three things: the enemy has been hit by a bullet, the enemy has hit the player, or the enemy has reached the beginning of the screen. In all three cases the enemy is dead, so it should be removed outside of the screen.

Similarly, with the enemies of type one and two, the position value is decreased. However, another important difference is that with these two enemies the vertical position is not kept constant. Though not yet fully developed, the main jist is clear. For type two, the vertical position will also be incremented. When it has reached the top of the screen, the value will be decreased in order for it to move downwards, switching again when it has reached the bottom. For the third type, an additional state is introduced. This results in two "move" states. After is has reached the spawn state, it will be brought to the first move state. In this state, all position values are subtracted. Then follow a comparison to the position of the player. If the player is below an enemy, it will remain on its course, otherwise, it will move up. It does this until the collision value becomes ’1’.

4 Results

Now the design has been made and implemented it is time to test if it reaches all of the specifications. This will be done in with two different tests the first way is to simulate the VHDL code we have written to see if it actually works. The second test is to see how much chip area it will take to implement, since it will eventually all have to fit on the chips.

4.1. Simulations

First lets test if the VHDL code works. For this we have written test benches and simulated the VHDL. We have not had the time to write and test all the individual compenents but we will discuss the components that are finished and simulated, these are: the player, the bullet, the enemy of type one, and the bullet control.

4.1.1. Player

Lets observe the behavior of the player according to its inputs. The start position given in the test bench for the player was x = "00011111" and y = "001111111". The first interesting point to look at is around 130 ns. Both the input of x and y become "01" which means the move right and down button are pressed and thus the x- and y-coordinates should be increased as seen in the FSM of the player in Figure (A.1) and Figure (A.2). As we can see x is now "001000000" and has indeed increased by one. The same goes for the y-coordinate as y is now "010000000". The next interesting point is around 750 ns, there the control input is "10" for both x and y and thus the value should decrease which it does. But unfortunately there are still a few errors in the player which will have to be corrected. For instance the player only moves when the input controls are changed when in we want that if you keep pressing right it should keep going right. Also at 500 ns the input signal goes back to "00" and nothing should happen but x and y still increase. These are issues we are going to tackle.

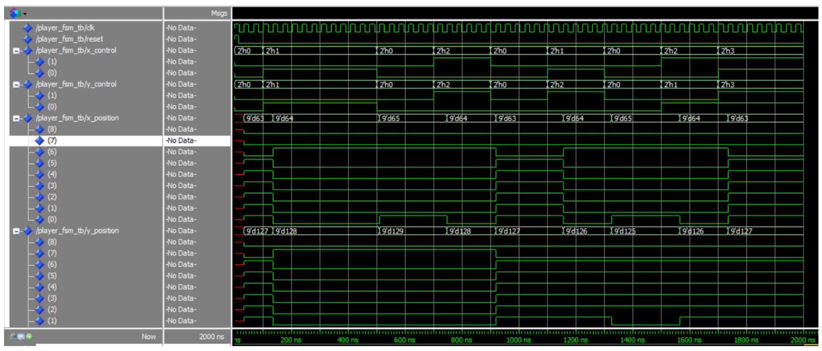

Figure 4.1: Simulation results player

4.1.2. Enemy type one

The first enemy type has a really easy behavior, it has to spawn at the coordinates provided at the input and then move left while staying on the same vertical position. As one can clearly see in Figure (4.2) the y position stays constant, but the x position is gradually going down with every clock cycle.

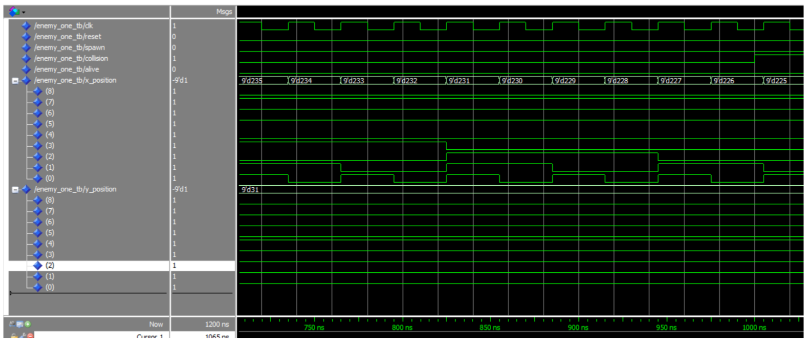

Figure 4.2: Simulation results enemy type one

4.1.3. Bullet

Lest observe the behavior of the bullet when the input signal shoot is high the bullet should be shot and thus be placed in screen at the position of the player. The wave form of the test bench is shown in Figure (4.3). This can be seen to happen at around 70 ns with a value for x now chosen as "000111111" and a value for y now chosen as "000000000". After it is placed in screen it should only move in the positive x direction. This can be seen to be working from 70 ns to 240 ns after that collision is high and the bullet is reset to the position x = "111111111" and y = "111111111".

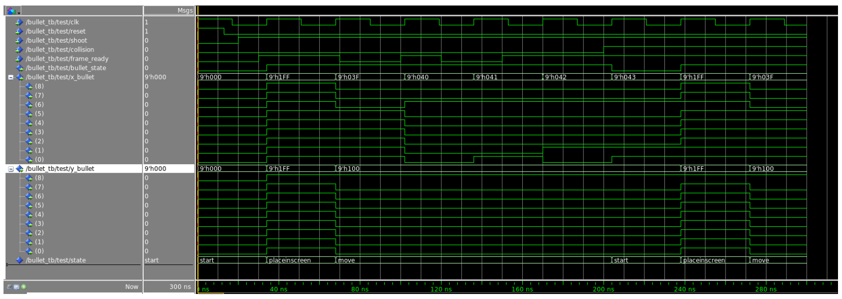

Figure 4.3: Simulation results bullet

4.1.4. Bullet control

Now we know the bullet work lets see if we can control which one should be shot at what times. You can see in table (4.4) what all the states are. The wave form of the test bench is shown in Figure (4.4). First time shoot is pressed bullet one is already in the screen so bullet 2 should be shot which happens at 50 ns. The next interesting point is at 250 ns where both bullet 1 and bullet 2 are already in the screen so bullet 3 is shot. The last two interesting points are at 375 ns and 550 ns, where respectivly there are no bullets shot because all three bullets are in the screen and bullet one is shot because no bullets are in the screen. So also the bullet control works.

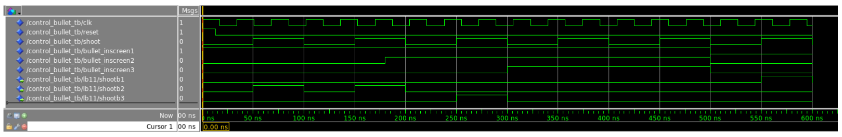
/control_bullet_tb/clk
/control_bullet_tb/reset
/control_bullet_tb/shoot
/control_bullet_tb/bullet_inscreen1
/control_bullet_tb/bullet_inscreen2
/control_bullet_tb/bullet_inscreen3
/control_bullet_tb/lb11/shootb1
/control_bullet_tb/lb11/shootb2
/control_bullet_tb/lb11/shootb3

Figure 4.4: Simulation results bullet controller

4.2. Chip size

Since not all the components are completely correct these numbers will change but they gave a rough estimate of how much area all the components will take on the chip. The total chip area we can use is $65000\mu m^2$.

4.2.1. player

The results of the place and route are shown below in Table (4.1). As we can see the player uses quite a big area but it is also of the most important components in the game and there is only one player in the game so there should not be a problem.

Type	Instances	Area	Area %
sequential	22	728.806	32.0
inverter	39	414.893	18.2
logic	93	1132.723	49.8
Total	154	2276.422	100.0

Table 4.1: Place and route results player

4.2.2. Enemy type one

The results of the place and route are shown below in Table (4.2). This is a rather alarming result since it uses quite a lot of room of the chip for one enemy and there will be twelve enemies. So for this we should consider changing something because otherwise it might not fit.

Type	Instances	Area	Area %
sequential	20	965.888	51.4
inverter	21	296.352	15.8
logic	39	616.851	32.8
Total	80	1879.091	100.0

Table 4.2: Place and route results enemy type one

4.2.3. Bullet

The results of the place and route for the bullets are shown below in Table (4.3). This is again quit a large area but less of a problem that the enemies since there will only be three bullets.

Type	Instances	Area	Area %
sequential	20	884.666	44.8
inverter	15	201.154	10.3
buffer	6	118.541	6.0
logic	62	766.125	38.8
Total	103	1973.485	100.0

Table 4.3: Place and route results bullet

4.2.4. Bullet control

The results of the place and route for the bullet control are shown below in Table (4.4). This is really small and will always fit on the chip so this is really positive.

Type	Instances	Area	Area %
inverter	1	6.586	5.8
logic	5	107.565	94.2
Total	6	11.150	100.0

Table 4.4: Place and route results bullet control

5 Conclusion

Unfortunately the module and the sub components weren't completely done, so we cant comment on the module as a whole but we will look at all the sub components separately. The first sub component we will look at is the player, as mentioned in Chapter 4 the player does not completely work as intended and there are improvements that need to be made. These improvements are that when the controls are held in the same position for example move right, the player should keep moving right and that when the button is released the player stops moving immediately. The size of the player on the chip is manageable but some improvements in efficiency could be made. The bullet and bullet control do work as intended and are also small enough to not cause issues on the chip. The enemies are a different story they work as intended but are really big, each enemy will take up about 3% of the chip and while we can only use 70% and since we have 12 enemies the enemies alone will take up almost half of the chip area we can use. So to conclude the specifications have not yet been met and there are a lot of improvements to be made.

Give a summary on whether the design meets the specifications.

A

Appendix

A.1. VHDL code describing the player's movement

```
library IEEE;
use IEEE.std_logic_1164.ALL;

entity bullet_logic is
   port(shoot                                         : in  std_logic;
        bullet_inscreen_1                        : in  std_logic;
        bullet_inscreen_2                        : in std_logic;
        bullet_inscreen_3                        : in std_logic;
        shootb1       : out std_logic;
        shootb2       : out std_logic;
        shootb3       : out std_logic);
end bullet_logic;

architecture behaviour of control_bullet is

begin
process (bullet_inscreen_1, shoot)
begin

if  (bullet_inscreen_3 = '0' && bullet_inscreen_2 = '0' && bullet_inscreen_1 = '0' &&
 ↪  shoot = '1') then
        shootb1 <= '1';
              shootb2 <= '0';
        shootb3 <= '0';
elsif (bullet_inscreen_3 = '0' && bullet_inscreen_2 = '0' && bullet_inscreen_1 = '1'
 ↪  && shoot = '1') then
         shootb1 <= '0';
         shootb2 <= '1';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '0' && bullet_inscreen_2 = '1' && bullet_inscreen_1 = '0'
 ↪  && shoot = '1') then
         shootb1 <= '1';
         shootb2 <= '0';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '0' && bullet_inscreen_2 = '1' && bullet_inscreen_1 = '1'
 ↪  && shoot = '1') then
         shootb1 <= '0';
         shootb2 <= '0';
```

```
                shootb3 <= '1';
elsif (bullet_inscreen_3 = '1' && bullet_inscreen_2 = '0' && bullet_inscreen_1 = '0'
 ↪  && shoot = '1') then
         shootb1 <= '1';
         shootb2 <= '0';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '1' && bullet_inscreen_2 = '0' && bullet_inscreen_1 = '1'
 ↪  && shoot = '1') then
         shootb1 <= '0';
         shootb2 <= '1';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '1' && bullet_inscreen_2 = '1' && bullet_inscreen_1 = '0'
 ↪  && shoot = '1') then
         shootb1 <= '1';
         shootb2 <= '0';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '1' && bullet_inscreen_2 = '1' && bullet_inscreen_1 = '1'
 ↪  && shoot = '1' then
         shootb1 <= '0';
         shootb2 <= '0';
                shootb3 <= '0';
else
         shootb1 <= '0';
         shootb2 <= '0';
                shootb3 <= '0';
end if;



end process;

end behaviour;
```

A.2. VHDL code describing the bullet logic

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_unsigned.ALL;
use IEEE.numeric_std.ALL;

ENTITY          bullet is

        port        (       clk                :        in std_logic;
                        reset                :        in std_logic;
                        x_position          :        in std_logic_vector(8 downto 0);
                        y_position          :        in std_logic_vector(8 downto 0);
                        shoot                :        in std_logic;
                        collision          :        in std_logic;
                        frame_ready          :        in std_logic;
                        bullet_state          :        out std_logic;
                        x_position_bullet :        out std_logic_vector(8 downto 0);
                        y_position_bullet :        out std_logic_vector(8 downto 0)
                );
end bullet;

ARCHITECTURE behaviour of bullet is

    type state_fsm_enemy is (start, placeinscreen, move);
    signal state, new_state : state_fsm_enemy;
    signal x_pos, x_pos_new : std_logic_vector(8 downto 0);
    signal y_pos, y_pos_new : std_logic_vector(8 downto 0);

    begin
    process(clk)
    begin
        if (clk'event and clk = '1') then
           if (reset = '1') then
              state <= start;
              x_pos <= (others => '0');
              y_pos <= (others => '0');
          else
              state <= new_state;
              x_pos <= x_pos_new;
              y_pos <= y_pos_new;
          end if;
        end if;
    end process;

    process (state, collision, shoot,x_pos,y_pos,x_position,y_position)
    begin
      case state is
      when start =>
          x_pos_new <= "111111111";
          y_pos_new <= "111111111";
          bullet_state <= '0';
          if (shoot = '1') then
              new_state <= placeinscreen;
          else
              new_state <= start;
          end if;
      when placeinscreen =>
          x_pos_new <= x_position;
          y_pos_new <= y_position;
          bullet_state <= '1';
          new_state <= move;

      when move =>
          x_pos_new <= x_pos + "000000001" ;
          y_pos_new <= y_pos;
          bullet_state <= '1';
          if (collision = '1') then
              new_state <= start;
          else
              new_state <= move;
          end if;

      when others=>
          x_pos_new <= "111111111";
          y_pos_new <= "000011111";
          bullet_state <= '0';
          if (shoot = '1') then
              new_state <= placeinscreen;
          else
              new_state <= start;
          end if;
      end case;

    end process;

    x_position_bullet <= x_pos;
    y_position_bullet <= y_pos;

END ARCHITECTURE behaviour;
```

A.3. VHDL code describing the bullet controller logic

```
library IEEE;
use IEEE.std_logic_1164.ALL;

entity bullet_logic is
   port(shoot                                            : in  std_logic;
        bullet_inscreen_1                         : in  std_logic;
        bullet_inscreen_2                         : in std_logic;
        bullet_inscreen_3                         : in std_logic;
        shootb1       : out std_logic;
        shootb2       : out std_logic;
        shootb3       : out std_logic);
end bullet_logic;

architecture behaviour of control_bullet is

begin
process (bullet_inscreen_1, shoot)
begin

if  (bullet_inscreen_3 = '0' && bullet_inscreen_2 = '0' && bullet_inscreen_1 = '0' && shoot = '1'
        shootb1 <= '1';
              shootb2 <= '0';
        shootb3 <= '0';
elsif (bullet_inscreen_3 = '0' && bullet_inscreen_2 = '0' && bullet_inscreen_1 = '1' && shoot = '
         shootb1 <= '0';
         shootb2 <= '1';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '0' && bullet_inscreen_2 = '1' && bullet_inscreen_1 = '0' && shoot = '
         shootb1 <= '1';
         shootb2 <= '0';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '0' && bullet_inscreen_2 = '1' && bullet_inscreen_1 = '1' && shoot = '
         shootb1 <= '0';
         shootb2 <= '0';
                shootb3 <= '1';
elsif (bullet_inscreen_3 = '1' && bullet_inscreen_2 = '0' && bullet_inscreen_1 = '0' && shoot = '
         shootb1 <= '1';
         shootb2 <= '0';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '1' && bullet_inscreen_2 = '0' && bullet_inscreen_1 = '1' && shoot = '
         shootb1 <= '0';
         shootb2 <= '1';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '1' && bullet_inscreen_2 = '1' && bullet_inscreen_1 = '0' && shoot = '
         shootb1 <= '1';
         shootb2 <= '0';
                shootb3 <= '0';
elsif (bullet_inscreen_3 = '1' && bullet_inscreen_2 = '1' && bullet_inscreen_1 = '1' && shoot = '
         shootb1 <= '0';
         shootb2 <= '0';
                shootb3 <= '0';
else
         shootb1 <= '0';
         shootb2 <= '0';
                shootb3 <= '0';
```

```
end if;



end process;

end behaviour;
```

A.4. VHDL code describing the first enemy

```
Library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_unsigned.ALL;
use IEEE.numeric_std.ALL;

ENTITY enemy_one is
    port  (   clk                   : in std_logic;
              reset                 : in std_logic;
              x_spawn_pos           : in std_logic_vector(8 downto 0);
              y_spawn_pos           : in std_logic_vector(8 downto 0);
              spawn                 : in std_logic;
              collision             : in std_logic;
              x_position            : out std_logic_vector(8 downto 0);
              y_position            : out std_logic_vector(8 downto 0);
              alive                 : out std_logic
    );
end enemy_one;

ARCHITECTURE fsm_enemy_one of enemy_one is

    type state_fsm_enemy is (start, placeinscreen, move);
    signal state, new_state : state_fsm_enemy;
    signal x_pos, x_pos_new : std_logic_vector(8 downto 0);
    signal y_pos, y_pos_new : std_logic_vector(8 downto 0);

    begin
    process(clk)
    begin
        if (clk'event and clk = '1') then
           if (reset = '1') then
              state <= start;
              x_pos <= (others => '0');
              y_pos <= (others => '0');
          else
              state <= new_state;
              x_pos <= x_pos_new;
              y_pos <= y_pos_new;
          end if;
        end if;
    end process;

    process (state, collision, spawn,x_pos,y_pos,x_spawn_pos,y_spawn_pos)
    begin
      case state is
      when start =>
          x_pos_new <= "111111111";
          y_pos_new <= "111111111";
          alive <= '0';
          if (spawn = '1') then
              new_state <= placeinscreen;
          else
              new_state <= start;
          end if;

      when placeinscreen =>
```

```
          x_pos_new <= x_spawn_pos;
          y_pos_new <= y_spawn_pos;
          alive <= '1';
          new_state <= move;

      when move =>
          x_pos_new <= x_pos - "000000001" ;
          y_pos_new <= y_pos;
          alive <= '0';
          if (collision = '1') then
              new_state <= start;
          else
              new_state <= move;
          end if;

      when others=>
          x_pos_new <= "111111111";
          y_pos_new <= "000011111";
          alive <= '0';
          if (spawn = '1') then
              new_state <= placeinscreen;
          else
              new_state <= start;
          end if;
      end case;

    end process;

    x_position <= x_pos;
    y_position <= y_pos;

END ARCHITECTURE fsm_enemy_one;
```

A.5. Finite State Machines

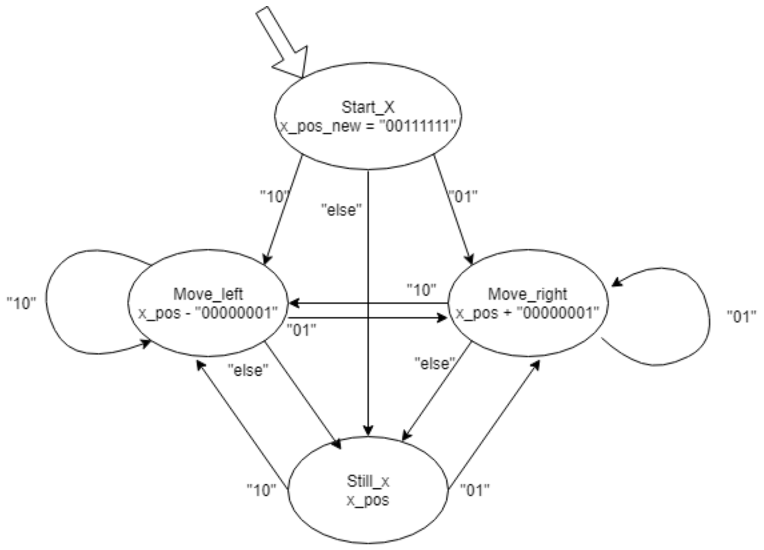

Figure A.1: FSM describing the horizontal player movement

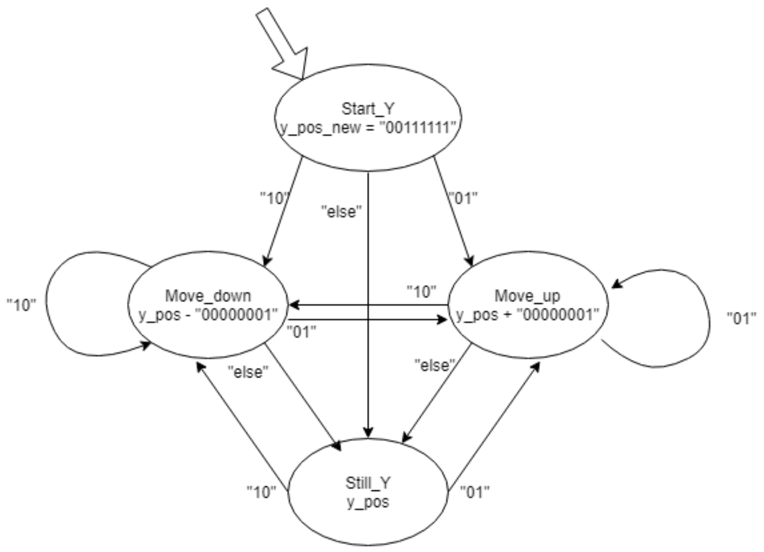

Figure A.2: FSM describing the vertical player movement

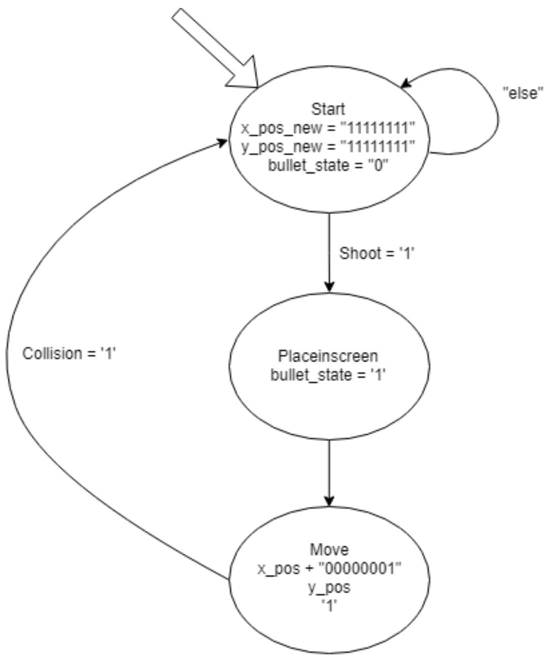

Figure A.3: FSM describing the bullet logic

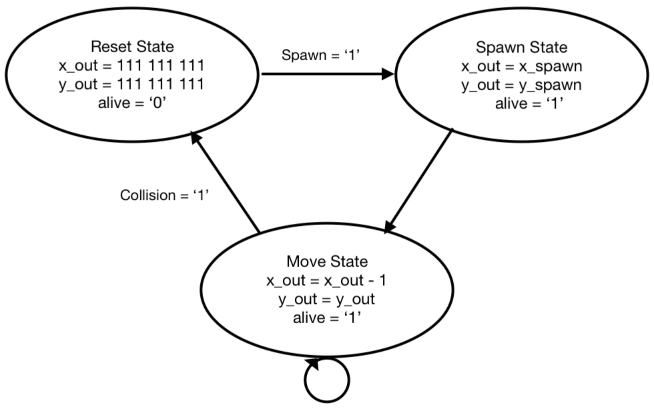

Figure A.4: FSM showing the basic enemy movement

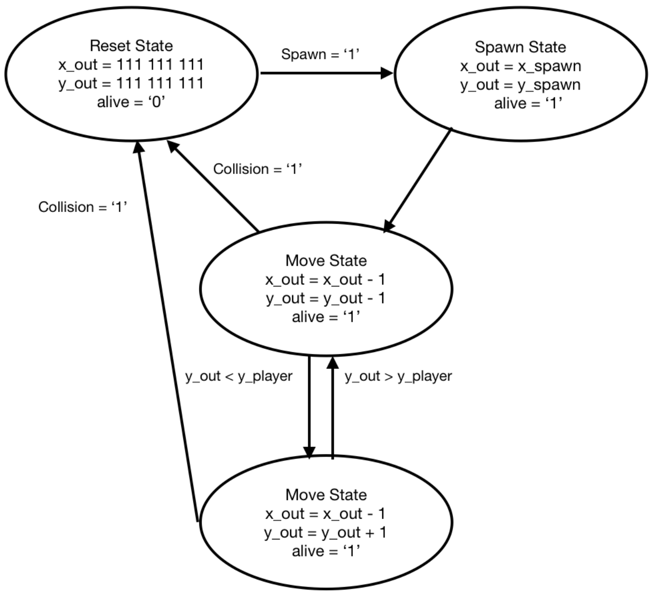

Figure A.6: FSM showing the type three enemy movement

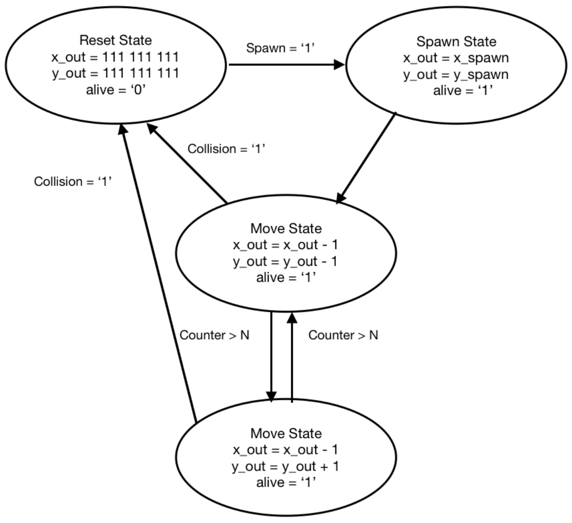

Figure A.5: FSM showing the type two enemy movement