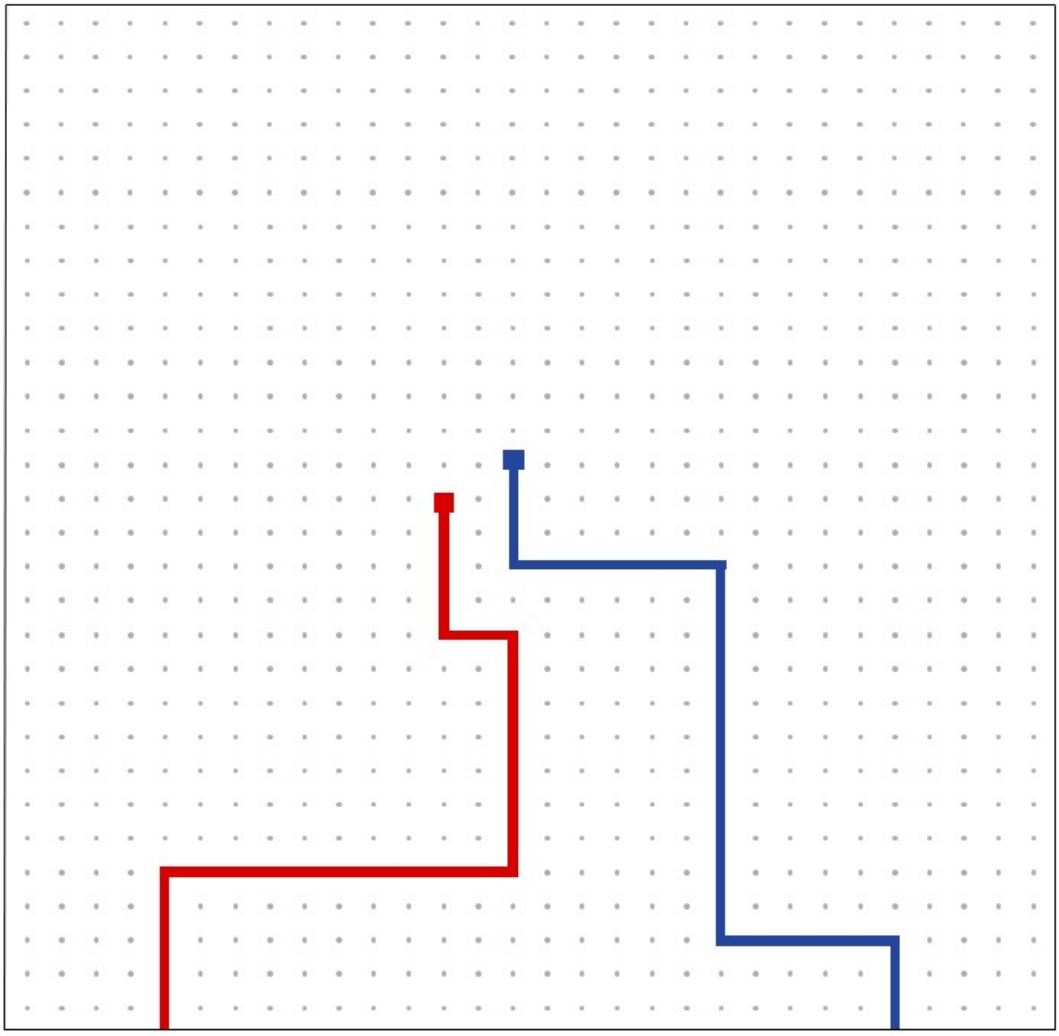
Game Engine

Eva Nootenboom, 5281288
Willem van Overbeeke, 5363950
Dagmar Westenbrink, 5379954

December 10, 2021



Abstract

This report describes a module of an integrated circuit, which is responsible for actualising the rules of a game. The game, called ElecTRON, is a two player game where players move at a constant speed while trying to enclose the other player and avoid being enclosed themselves. This module does the operations that make the game run. This so called game engine consist of a counter, a few registers and a controller. The game engine has been tested by synthesis and simulations. After simulating the synthesized version, it can be concluded that this module works correctly.

Contents

1 Introduction

The game engine is a module of the chip for ElecTRON. ElecTRON is a game, based on the movie tron [**1**], of two players in a battle to stay alive the longest. The game is played on a grid, consisting of two layers displayed on top of each other. While both player are alive they move at a constant speed throughout the grid. Every grid cell that is visited by a player is made into a wall and is no longer accessible to either player. This leads to a game of rising tension as the playing field gets smaller and smaller while the players try to enclose each other.

From the description above, a few functions for the chip can be distilled. Firstly, the players need to move and the position needs to be changed by the game engine. Secondly, the traveled path needs to be stored in memory, and lastly memory must be read to see if a player collides with a wall. All these operations which make the game run are done by the game engine. Two other modules control the memory and the output to a monitor.

Some other features were proposed but have not yet been implemented. These features include: a sound engine; adding a second layer to the playing field; and giving the players the ability to get a short boost. Although these features are not yet implemented, the code has been written with these features in mind. This is done to make the implementation of these extras easier.

2 Specification

In this section all the requirements, what the module should do and the in- and outputs of the module are explained in detail. The black box with all the inputs and outputs can be seen in Figure 1.

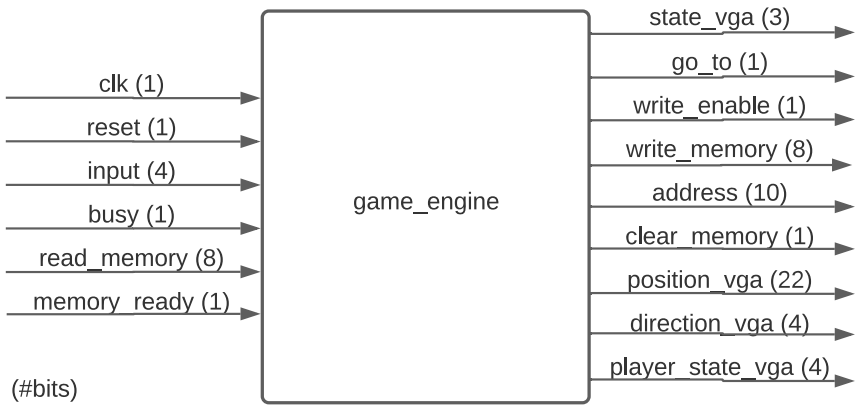


Figure 1: Black box Game Engine.

2.1 Requirements

The module is required to do the following (all these requirements go for both players):

- Determine which direction the player is moving.
- Determine the next position of the player.
- Send the direction and current position to the graphics engine.
- Determine the shape of the wall of the previous position of the player and send this to the memory module.
- Determine whether or not the player collides with the border.
- Determine whether or not a player has already been at the next position; meaning that the player collides with a wall.
- Determine whether or not the players are colliding with each other.
- Determine who won, when one or both players collided with a border or a wall.
- Determine the state of a player: collided at a border of a cell; collided in the middle of the cell; still playing the game; the player is not yet ready.
- Send the state of the player to the graphics engine.
- Send to the graphics engine if the game has started or not, is going on or else which player has won.
- Determine when the game starts and finishes.

In order to do all of this the game engine needs to store data. In Section 3.2 is it explained which data should be stored and how that is done.

2.2 Inputs

The game engine has the following inputs: `clk`, `reset`, `input`, `busy`, `read_memory` and `memory_ready` (see Figure 1). The following sections give details on the different inputs.

2.2.1 Clk

`clk` is the main clock signal from the chip. The frequency of the clock is $25\,\mathrm{MHz}$. The FSM only goes to another state when the clock is going from '0' to '1'; on a rising edge. The register also only updates the stored values at a rising edge of the clock.

2.2.2 Reset

This is a real button. When this button is pressed the chip is reset. This means that the game will go back to its starting screen, the game engine will go back to the `reset_state` and the register will be made empty so a new game can be played.

2.2.3 Input

The `input` is a four bit signal. This input comes from the buttons which are pressed by the players. Both players can press four buttons: up, down, left and right. By using SR-latches among other things this is transferred into a four bit signal. The two most significant bits give the direction of player 1 and the two least significant bits give the direction of player 0. In Table 1 the bits that correspond to each direction can be seen.

Table 1: The meaning of the bits related to the direction of the players.

bits	direction
00	up
01	left
10	down
11	right

2.2.4 Busy

`busy` is a signal that is created by the graphics engine. This is a signal slightly below 60 Hz. It is '1' when the graphics engine is using the memory and '0' otherwise. While the graphics engine is using the memory, the game engine cannot. So only when the busy signal is '0' the game engine can use the memory. The signal is '0' for 36000 clock cycles, which is reasoned to be enough for all calculations and reading from and writing to memory.

2.2.5 Read_memory

`read_memory` is an 8 bit signal that is created by the memory module in the chip. The memory stores a value for 32x32 addresses. `read_memory` is the value at one of those addresses, as requested from the game engine. The four most significant bits are used for layer 1 and the four least significant bits are used for layer 0. Due to the fact that at this moment the game is only made for one layer the four most significant bits are always '0'. Of the other four bits, the fourth least significant bit represents

the player and the third least significant bit to the first least significant bit represent the wall shape. For example, `0111` means that player 0 left a trail in the shape ∟. In Table 2 the bits that correspond to each wall shape can be seen.

Table 2: The meaning of the bits related to the wall shape

bits	wall shape
000	empty
001	–
010	ı
110	⌜
101	⌝
100	⌟
111	∟

2.2.6 Memory_ready

`memory_ready` is an one bit signal that is created by the memory module of the chip. This signal is '0' when the memory module is busy reading a value from the memory component or writing a value to the memory component. Otherwise the value is '1', meaning the memory controller is not busy.

2.3 Outputs

The game engine has the following outputs: `state_vga`, `write_enable`, `write_memory`, `address`, `position_0_vga`, `position_1_vga`, `direction_0_vga`, `direction_1_vga`, `player_state_0_vga`, `player_state_1_vga`, `go_to` and `clear_memory` (see Figure 1).The following sections give details on the different outputs.

2.3.1 state_vga

`state_vga` is three bit signal that is send to the graphics engine. This signal contains information about what state the game is in. The following values correspond to the following state in the game:

Table 3: The meaning of the bits related to the game state.

bits	game state
000	startscreen
001	tie
010	player 0 won
011	player 1 won
111	playing the game

2.3.2 player_state_0_vga and player_state_1_vga

There are several states the player can be in: alive, collided at a border, collided at the center or not ready. The signals `player_state_0_vga` and `player_state_1_vga` consist of two bits and let

the graphics engine know what state the players are in. With this information the graphics engine can display the player correctly. The following values correspond to the following situations in the game.

Table 4: The meaning of the bits related to the situation in the game.

bits	player state
00	player collided in the middle of a cell
01	player collided at the border of a cell
10	player is not ready to start the game
11	player is playing the game

2.3.3 write_enable

`write_enable` is a signal that goes to the memory module. This signal has to be '1' when writing a value to the memory. Once the memory is done writing this value onto the right address (see Section 2.3.5) the `write_enable` goes to '0'.

2.3.4 write_memory

`write_memory` is the value that should be written to memory onto the correct address (see Section 2.3.5). It has the same format as `read_memory` (see Section 2.2.5).

2.3.5 address

`address` is a 10 bit signal. This signal refers to one address in the memory. Each address corresponds to one cell in the playing field. The field has a grid of 30x30 and the memory has 32x32 addresses, meaning that the two highest addresses for both x and y are not used. The most significant five bits are the position on the y axis, with '00000' being the first row of the field (the highest on the screen). Five least significant bits: position in x-axis, with '00000' the most left position in the field.

2.3.6 position_0_vga and position_1_vga

`position_0_vga` and `position_1_vga` are 11 bit signals. These signals are send to the graphics engine and give information about the current position of the players. The first bit corresponds to the layer the player is on. At this moment it is always '0', because only one layer used. The other ten bits are split in two. See Section 2.3.5 for the meaning of the bits.

2.3.7 direction_0_vga and direction_1_vga

`direction_0_vga` and `direction_1_vga` are signals of two bits each, which send information to the graphics engine about the current direction of the players. The bits have the same meaning as the input signal. The meaning of the bits can be seen in Table 1.

2.3.8 go_to

`go_to` is a one bit signal which is sent to the memory module. This signal needs to be '1' for one clock cycle when something is read from or written to a specific address in memory.

2.3.9 clear_memory

`clear_memory` is a one bit signal which is sent to the memory module. When this signal is '1' the memory module will clear the memory, meaning that all the values in the memory will be set to "00000000".

3 Design

The design of the game engine is divided in three parts: A counter, a register and a controller. The counter counts the amount of times the busy signal goes from high to low. The register functions as memory to store values. Finally, the controller is used to determine, for example, the next position of the player, send information to the graphics engine and send data which needs to be stored to the memory. In this section it is explained in detail how all these parts work.

3.1 Busy counter

The busy counter is a component that counts the number of times the `busy` signal goes from high to low. The busy signal tells when the graphics engine is creating the output for the monitor. Because the graphics engine reads from the memory when rendering, the game engine cannot read from or write to from the memory at that same time. Only when busy is low it is possible for the game engine to do computations. Keeping in mind the delays from reading from and writing to memory the game engine is reasoned to still have enough time to complete all computations before busy rises to '1' again.

If calculations are done every time busy is low, the players move every frame update. In that case the movement of the player is too fast. For this problem the busy counter was made. It counts the falling edges of the busy signal and outputs its count to the controller. In the controller the count is compared to a constant, which is set at 16 for now. Once all the different modules are implemented together it will be known whether or not this value is okay, too low or too high. When the count is bigger or equal to the constant the counter will be reset by the controller.

The busy counter is realised as an FSM. The finite state diagram can be seen in Figure 2 and the VHDL code in Appendix 3.1. In the state `busy_high_to_low` the count is increased, thus counting falling edges. This means the count increases one clock cycle after the falling edge of busy. However, given the low frequency of busy, one clock cycle does not matter.

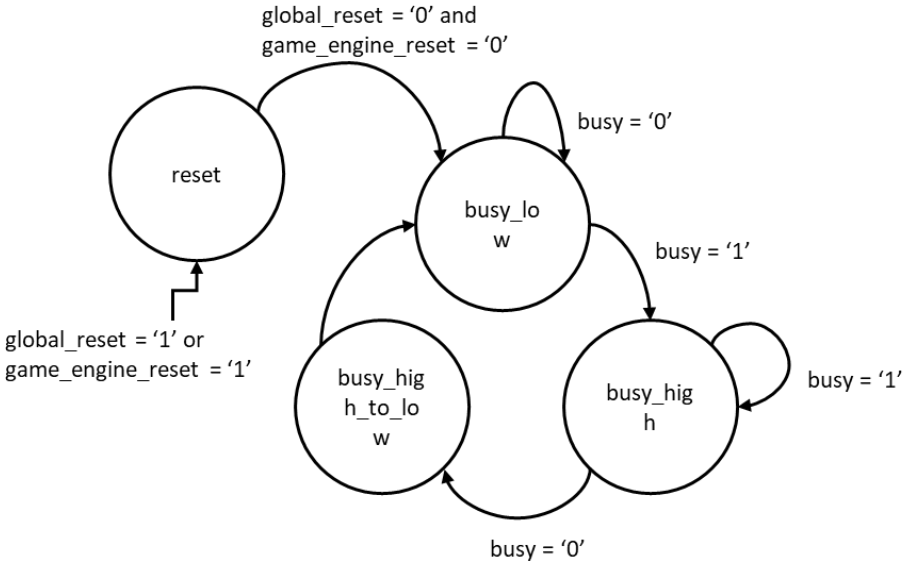


Figure 2: FSM busy counter.

3.2 Register

The game engine needs registers to store values, because some values need to be kept the same even though the FSM is going to another state. The register has an enable function, meaning only when the enable signal is high the value should be changed. For instance, when the inputs of player 1 are being sampled, the enable signal `e_next_direction_1` is set to '1' while the data signal `d_next_direction_1` is made equal to the input of player 1. The VHDL code for the register can be seen in Appendix C.2 and the encoding for the different signals can be found in Section 2.2 or Section 2.3.

Because the area on the chip and the number of flip-flops are limited, storing a certain value on the chip must be necessary otherwise it takes up much needed space. The reasons for storing a value in the register is given for each value that needs to be stored in the following paragraphs. The entity in the VHDL code of the register is named `ge_register` (with `ge` being short for game engine).

3.2.1 position and next_position

To keep track of the players during the game, the position needs to be stored because the next position of a player depends on its last position as well as the inputs. This can be achieved by only saving one pair of coordinates, however, both values are needed for a special situation. Namely, when the two players are directly facing each, as seen in Figure 5b. The next cycle both players crash into the wall left by the other player. However, the player should explode at the edge of the cell, meaning this situation has to be checked for separately. In terms of `position` and `next_position`, the position of one player equals the next position of the other player and vice versa. Both `position` and `next_position` are needed in this comparison so both have to be stored in the register.

3.2.2 direction

The shape of a wall does not only depend on the direction in which the player leaves a cell, but also on the direction the player entered the cell. The direction the player came from is the input of the previous cycle. This means that the direction needs to be stored for the next cycle to then again determine the wallshape.

3.2.3 wallshape

After assigning a value to the wall shape, it needs to be written to memory, this is done in different states of the FSM. Therefore the wallshape needs to be stored in the register.

3.2.4 read_memory

The part of the register for storing the data read from the memory is not yet used, but once the second layer is implemented, storing that data is necessary. This follows from the method of saving the data of the playing field in the memory. The 8 bits per address are divided in two, the most significant four are for layer 1 and the least significant four bits are for layer 0. Writing a wall to layer 1 should not destroy the data for layer 0 and vice versa. Thus the data read from the memory needs to be stored in order only change half of it.

3.2.5 next_direction

The part in the register labeled `next_direction` is used to store the inputs before starting the computations. This ensures that the direction a player leaves a cell does not change while calculating the next position or the shape of the wall.

3.2.6 player_state

The information about whether or not the player is alive or how a player died, is stored in the `player state`. Checking if and how a player died is done in multiple states and in some instances the next state of the controller FSM depends on the player state.

3.3 Controller

The calculations and the communication with the memory and register are done by the controller. First, some details about the communication with the memory module are given, thereafter about whether the FSM is a mealy or a moore machine and finally the states of the FSM are listed with a description of their functionality. Some state are doubled, one per player. If so, the number of the player is replaced with a #. The FSM of the controller can be seen in Figure 3.

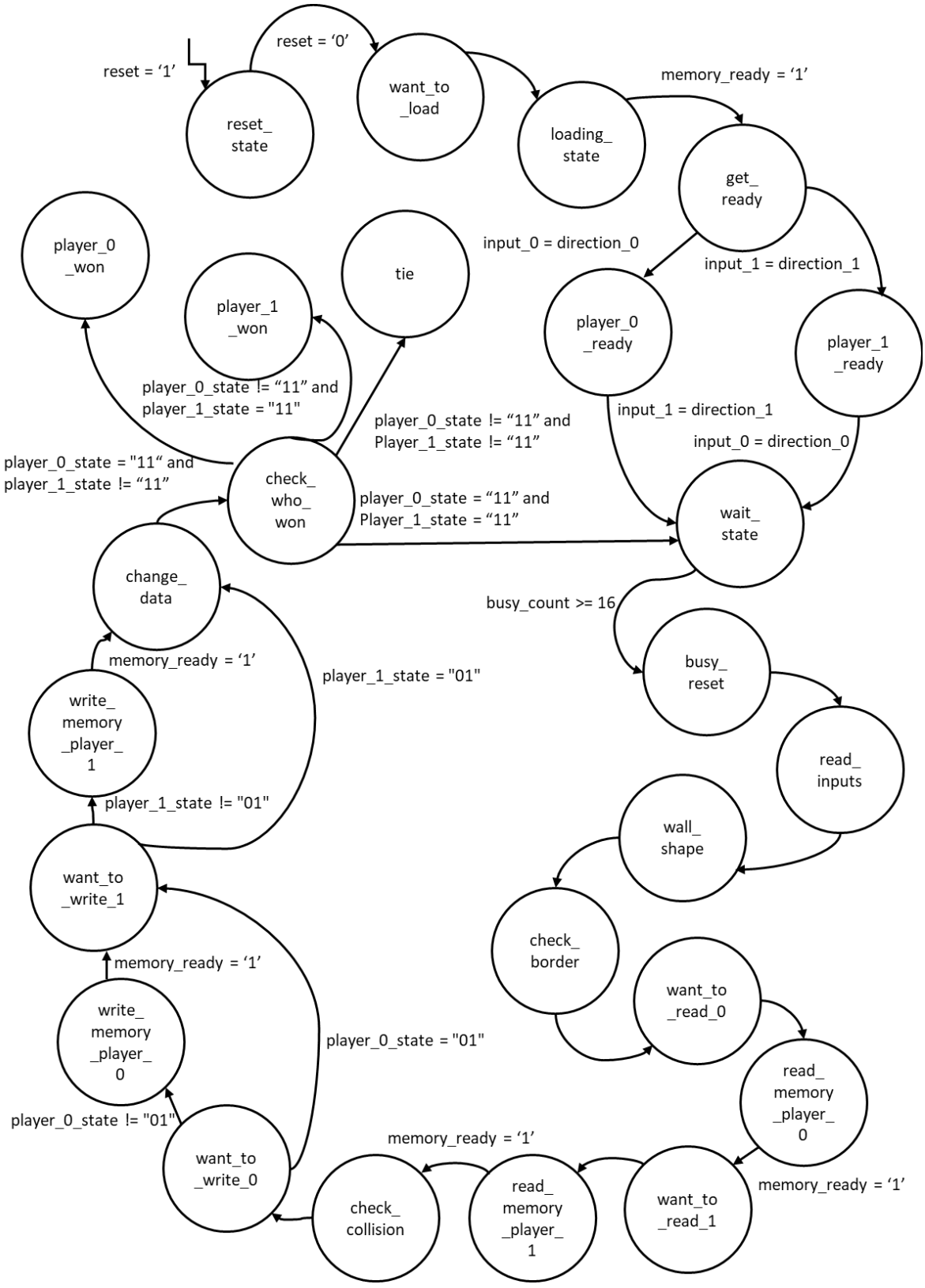


Figure 3: FSM game engine.

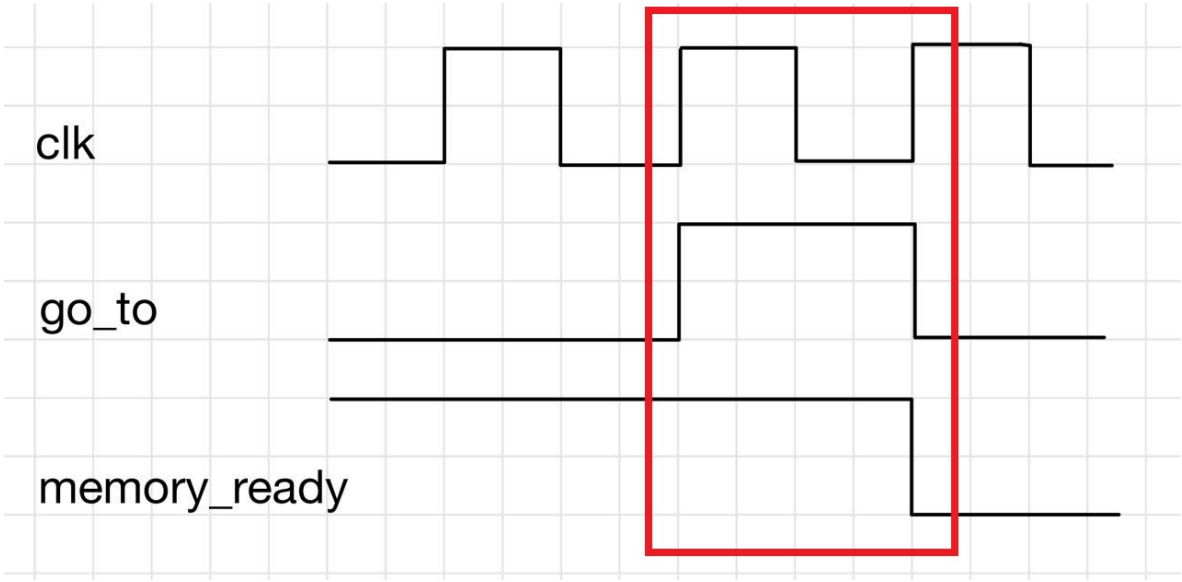


Figure 4: Waveform displaying the communication with the memory module.

3.3.1 Communication with the Memory Module

Whenever the game engine wants to communicate with the memory module two states are needed. The first reason is that the memory module needs the `go_to` signal to be high for only one clock cycle. The second reason is that `memory_ready` will be high for one more clock cycle after a task is sent to the memory module. This means that only after a rising clock edge `memory_ready` can be checked to see whether the FSM can go to the next state. Thus, the first state should send the task to the memory module and after one clock cycle the FSM can go to the state in which it checks whether or not the memory module is finished while also sending the task (this is needed by the memory controller).

Figure 4 shows a waveform containing a task for the memory module. Highlighted in red is the first of the two states. In this state `go_to` is set high, meaning the next rising edge the memory goes to a specified address. As can be seen, this state `memory_ready` is still high and thus the second state is needed.

3.3.2 Mealy or Moore Machine

The FSM of the controller is neither a moore machine nor a mealy machine. The VHDL code below is a sample of the FSM to illustrate. The first piece is taken from the state `read_memory_player_1` and shows how some of the output signals are dependent on input signals. For instance, the signal `e_player_1_state` changes when `read_memory` changes. The risk of having this mealy implementation is that the critical path becomes to long. Especially with multiple mealy machines together, there could be unexpectedly long delay times. It is reasoned however, that this is not the case in this instance because the outputs only go to the register. Because of this there is no path directly from input to output, there is always a register in between.

The second sample is from the state `want_to_write_1`. In this state the output signals depend directly on the input signals also. However, the input signals come from the register, which only change with the clock. Thus in this case a direct path from the inputs to the outputs of the game

engine is prevented as well.

Whenever a signal going out of the FSM not only depends on the state but also on inputs, it is similar to one of the two cases illustrated above. This ensures that no direct path from input to output is created without the register in between and no timing issues of the mealy implementation arise.

```
-- from the state 'read_memory_player_1'
if (memory_ready = '1') then
  if (read_memory = "00000000") then
    e_player_1_state <= '0';
    d_player_1_state <= (others => '0');
  else
    e_player_1_state <= '1';
    d_player_1_state <= "00";
  end if;
  new_state <= check_collision;
else
  e_player_1_state <= '0';
  d_player_1_state <= (others => '0');
  new_state <= read_memory_player_1;
end if;

-- from the state 'want_to_write_1'
if (player_1_state = "01") then
  write_enable         <= '0';
  write_memory         <= "00000000";
  address              <= "0000000000";
  go_to                <= '0';
  new_state            <= change_data;
else
  write_enable               <= '1';
  write_memory(7 downto 3)   <= "00001";
  write_memory(2 downto 0)   <= wallshape_1;
  address                    <= position_1(9 downto 0);
  go_to                      <= '1';
  new_state                  <= write_memory_player_1;
end if;
```

VHDL/mealy_vs_moore.vhd

3.3.3 Reset

When the reset button is pressed the FSM goes to the `reset_state`. This happens regardless of the state the FSM is in. This is not shown in Figure 3 for readability. In the `reset_state` all signals are set to zero. In order to start the game, the reset button always needs to be pushed beforehand. When `reset` goes low, the FSM goes to the `want_to_load` state.

3.3.4 Loading

In the want_to_load state and the loading state the initial condition are set before starting the game. Some initial values for the register are hard coded, such as position and direction, and some are set to zero. Furthermore the entire memory needs to be set to zero. This is done by the memory module when clear_memory becomes '1'. Two states are needed for this as is explained in Section 3.3.1. When the memory is finished and memory_ready becomes '1' the FSM goes to the state get_ready.

3.3.5 Getting Ready

The get_ready, player_0_ready and player_1_ready states are made to give the players a heads-up before the game starts. When a sound engine is added, it could create a sound effect at this point. The FSM stays in this state until one of players enters the direction that the player is facing. For instance, when the player is facing up the person playing has to press the button pointing up. For now this direction is always up, maybe later this can be randomized with the position. The starting positions at this moment can be seen in Figure 23 in the appendix. When the correct button is pressed the color of the player changes to let the person know which of the two players is theirs. The FSM then goes to the state player_0_ready or player_1_ready depending on which player presses the right button first. Once both players are paying attention and have entered the correct direction the game engine can starts its first game cycle and the FSM goes to wait_state.

3.3.6 Waiting

In the state wait_state the FSM waits until the busy counter (see Section 3.1) reaches 16. By making this value lower or higher the players move faster or slower during the game respectively. After waiting for sixteen falling edges of busy, the FSM goes to busy_reset where the busy counter is reset. One clock cycle later the FSM is in the read_inputs state.

3.3.7 Reading the Inputs

In the state read_inputs it is determined which direction the player wants to go. The value of the input input is stored in the register for next_direction_0 and next_direction_1. This value is stored so that the direction of the player cannot change halfway one cycle in the FSM and thus avoids errors. The FSM then goes to the state wall_shape

3.3.8 Wall Shape and Next Position

In the wall_shape state the wall shape and the next position are determined. The wall shape a player leaves behind is determined using direction_# and next_direction_#. When a player wants to go in the opposite direction of what is was going it collides against itself. In this case no wall shape is determined and the player_#_state is set to "00", meaning the player dies in the middle of a cell.

The next position is calculated based on the direction a player is going and the position it is now. This is done by increasing or decreasing the y or x value of position_# with 1. The FSM then goes to the state check_border.

3.3.9 Collision with Border

The playing field is a grid of 30 by 30 cells (five by five bits). Around this grid is a border. When a player runs into a border, that player loses. To check if a player collides with a border the state `check_border` is made. It checks whether or not a player is going to the border by looking at the position of a player and its direction. If the direction is towards the border and the player is next to the border, the player crashes into the border, meaning the `player_state` changes. Then the FSM goes to the state `want_to_read_0`.

3.3.10 Reading from Memory

The states `want_to_read_#` and `read_memory_player_#` are used to read from the memory component. To do this the value of `go_to` should be '1' for one clock cycle. Because of this and what is explained in Section 3.3.1 two states are needed. Furthermore the value of `address` is set to the cell of the next position of the player. With this information from the memory it is checked if a player already has been here and thus if there is a wall there. Next, the FSM goes to the `check_collision` state.

3.3.11 Checking for player against player crashes

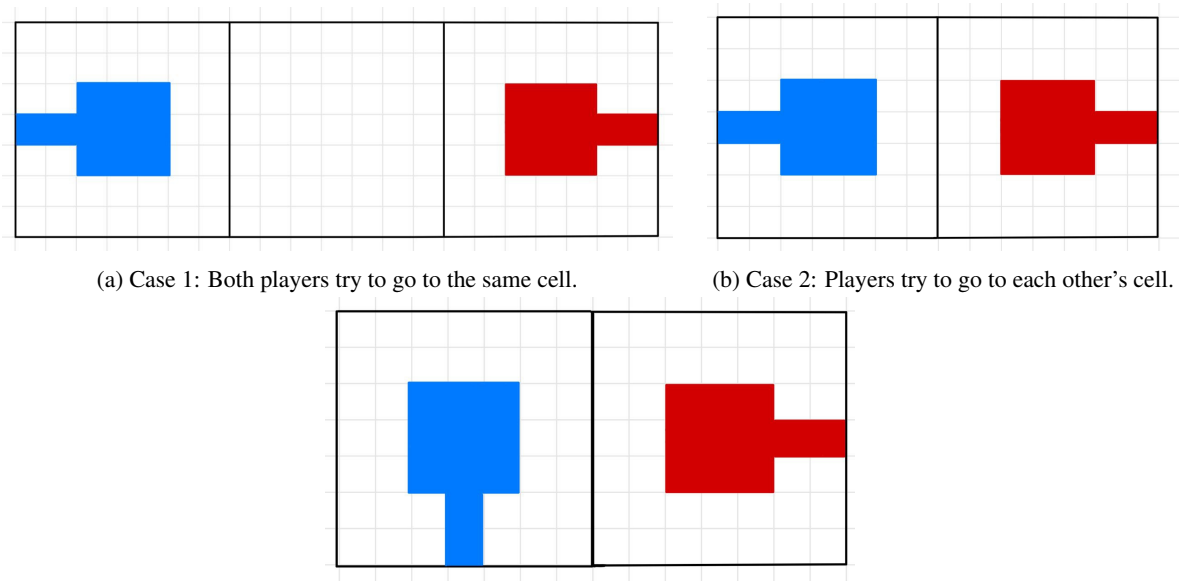
A few instances could occur in which the players crash against each other, which is checked in the state as `check_collision`. Four cases need to be checked. The first is when both players go to the same cell. In this case the players crash against each other in the middle of the cell. The second is when both players try to go to the cell the other player is on. The players then crash at the border of a cell. The last two cases are if one player tries to enter the cell which the other player is about to leave. Although the other player will leave a wall, this is not yet written to memory, thus making it necessary to check separately. Then the FSM goes to `want_to_write_0`. These four cases are also displayed in Figure 5.

3.3.12 Writing to Memory

Writing to memory is done by two states per player: `want_to_write_#` and `write_memory_player_#`. The `want_to_write_#` states are needed, next to the reasons mentioned in Section 3.3.1, because the state of the player must be checked. If a player crashes against the edge of a cell, the player never leaves the cell, so no wall should be written to the memory. After this the FSM goes to `change_data`.

3.3.13 Update Data

In the state `change_data` the signals `direction_#` and `position_#` are updated. Meaning that it gets the value of `next_direction_#` and `next_position_#`. However, if a player has crashed into the side of a cell, the position is not changed. By changing these signals the outputs `position_0_vga`, `position_1_vga`, `direction_0_vga` and `direction_1_vga` are also updated. After that the FSM goes to `check_who_won`.



(a) Case 1: Both players try to go to the same cell.

(b) Case 2: Players try to go to each other's cell.

(c) Case 3 and 4: One player tries to go to the cell of the other player.

Figure 5: Different player against player collisions.

3.3.14 Check Who Won

In the state `check_who_won` it is checked if one or both players have died. This is done by checking what the value for `player_#_state` is. If both players are still alive the game continues and the FSM goes back to the state `wait_state`. If one or both players have died the FSM goes to `player_0_won`, `player_1_won` or `tie`, depending on who won.

3.3.15 Game Over

After it is decided who has won the game, the FSM goes to one of the following states: `player_0_won`, `player_1_won` or `tie`. Depending on who won, the `state_vga` is set to a certain value (see Table 3). The FSM then stays in that state until the reset button is pressed.

4 Results

To test if the written VHDL code works correctly, it has been tested. This is done by first synthesizing the code and then simulating that using testbenches. First the register and the busy counter are synthesized and simulated separately after which the design is tested.

4.1 Busy Counter

In Figure 6 and 7 the simulation results of the synthesized circuit can be seen. It can be seen that after the `reset` has become '0' the busy counter counts. Furthermore it can be seen that it will start counting again from zero if `reset` goes high and low. From this it can be concluded that the busy counter works correctly.

4.2 Register

In Figures 8 and 9 the simulation results of the synthesized register can be seen. When `reset` is high, the outputs are all zeros. The data input signals, the signals starting with `d_`, are set to an arbitrary value by the testbench. These arbitrary values only appear at the output signals, starting with `q_`, when the corresponding enable signal is high. The enable signals are labeled using `e_` at the beginning of the name. In the simulation results this can be seen, meaning that the register works correctly.

4.3 Controller

To test the controller a couple of different testbenches are used to test most ways the game can run. For all simulations the synthesized circuit has been used.
For the first simulation (see Figure 10 and 11) one cycle in the FSM is tested where both players do not die. In this simulation the busy signal has a much higher frequency and a different duty cycle than in reality to reduce simulation times. Moreover, for the same reason the signal `memory_ready` is made '1' after less clock cycles then the memory module would do. From this simulation it can be concluded that the game starts up the correct way. Furthermore it can also be concluded that the busy signal counts correctly and that reading and writing from and to the memory works correct.

In all simulations explained below the `wait_state` and `busy_reset` states are skipped and `memory_ready` is always set to '1'. This is because it is already know that it works and it makes the simulation a lot easier and shorter. With this the following simulations are done:

- Both players collided against a border at the same time. One of the players collides against the border on left side of the playing field and the other player collides against the border on the right side of the playing field. How the players move thorough the field can be seen in Figure 24 and the simulation can be seen in Figure 12.
- One player collides against a border and the other player stays alive. In Figure 13 player 0 collides against the border at the top of the playing field and in Figure 14 player 1 collides at the border at the bottom of the playing field.
- One of the player presses the button in the direction opposite of what the player in going and the other player stays alive (see Figure 15 for the simulation results).

- The players collide head-on at the border of a cell. How the players collide can be seen in Figure 5b. (see Figure 16 and 17 for the simulation results)
- The players collide head to head in the middle of a cell. How the players collide can be seen in Figure 5a (see Figure 18 and 19 for the simulation results).
- The players move in a wall made by either himself or the other player. In the simulations in Figure 20, Figure 21 and Figure 22 it can be seen that first player 1 moves into a wall and player 0 stays alive. Then the other way around and lastly both players move into a wall at the same moment. Because the game engine is not connected to the memory module yet, the testbench just creates a wall in a cell even though the players have not been there yet, just to test the code.

All these simulation give the correct output and thus it can be concluded this all works correct.

4.4 Synthesized Circuit

After synthesises the following information is obtained about the circuit

Table 5: Synthesized circuit.

Number of sequential cells	75
Number of combinatorial cells	473
Area of the circuit	$10\,271.341\,\mu\mathrm{m}^2$

After synthesis the total area of the complete circuit should not pass $65\,000\,\mu\mathrm{m}^2$. The game engine module uses around 16 % of this. Depending of how much area the other modules use, the design will fit on the the chip.

5 Conclusion

The simulations of both the synthesized version and the non-synthesized version show that the game engine works correctly. The game engine can determine positions, directions, wall shapes, if a player has died and how and it can send the necessary information to the graphics engine and the memory module. Thus the game engine meets all the requirements.

References

[1] Steven Lisberger, ”Tron”, 1983, Walt Disney Productions,

A Simulations

In this section all the simulation results of the synthesized circuit can be seen.

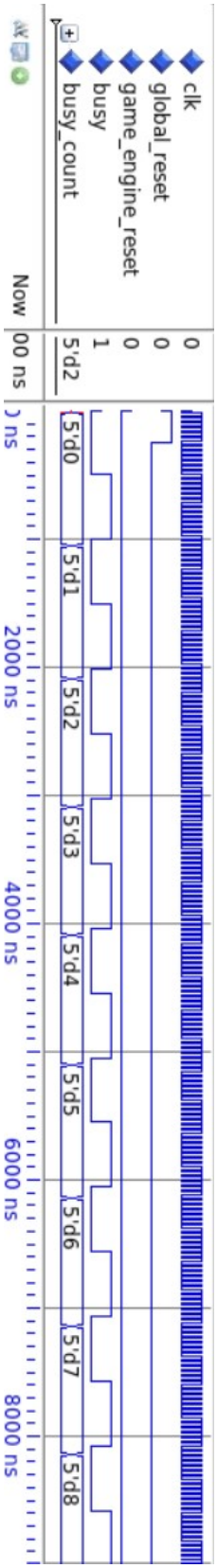


Figure 6: Simulation results from the busy counter (1).
This is the simulations result of the `busy_counter`. Because of readability it is split into two figures.

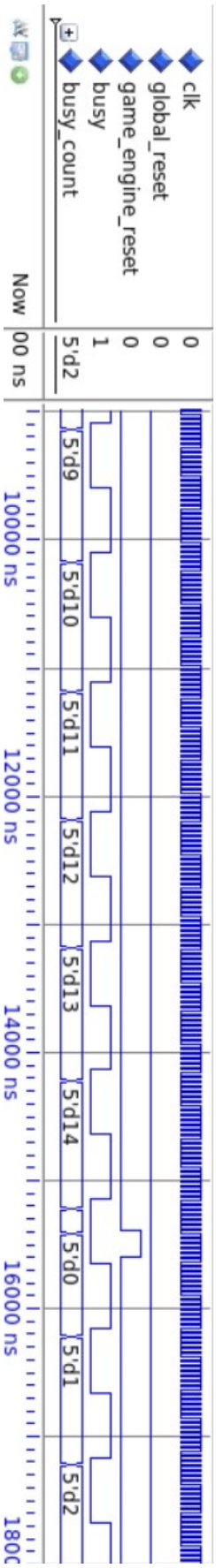


Figure 7: Simulation results from the busy counter (2).

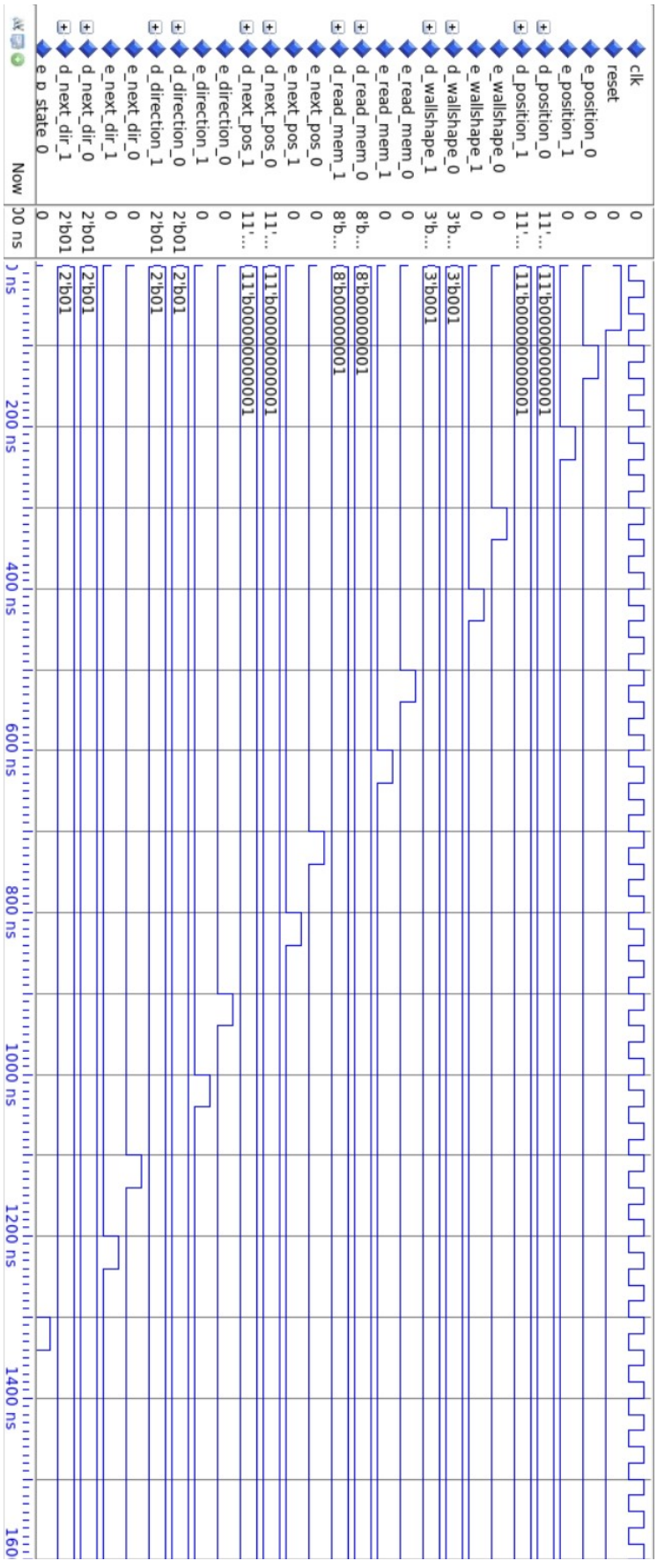


Figure 8: Simulation results from the register (1).

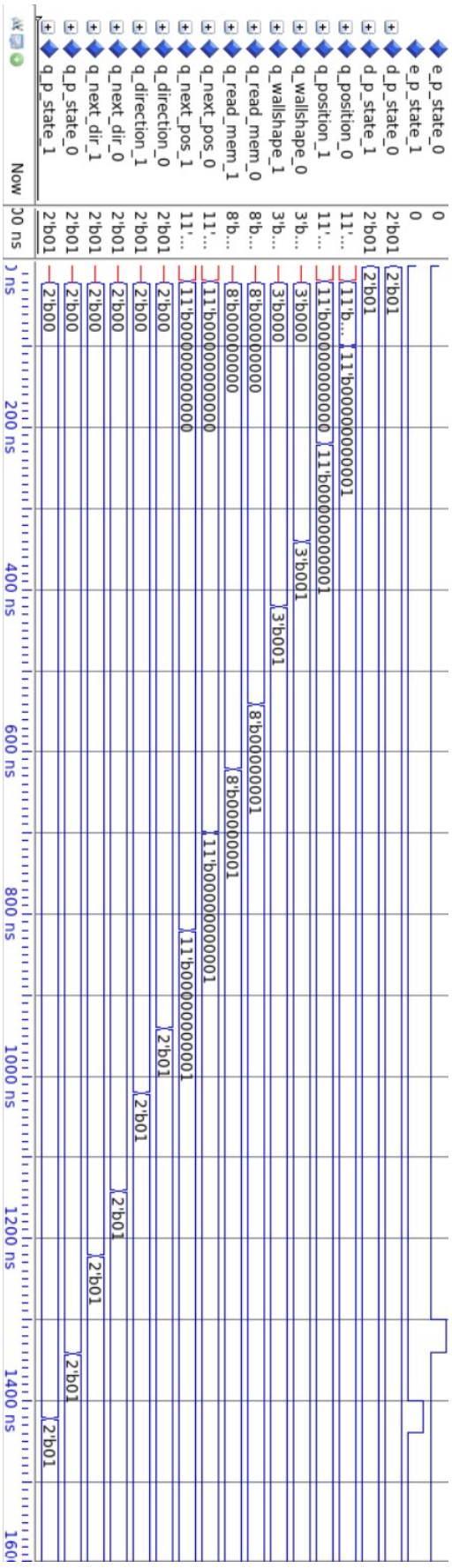


Figure 9: Simulation results from the register (2).

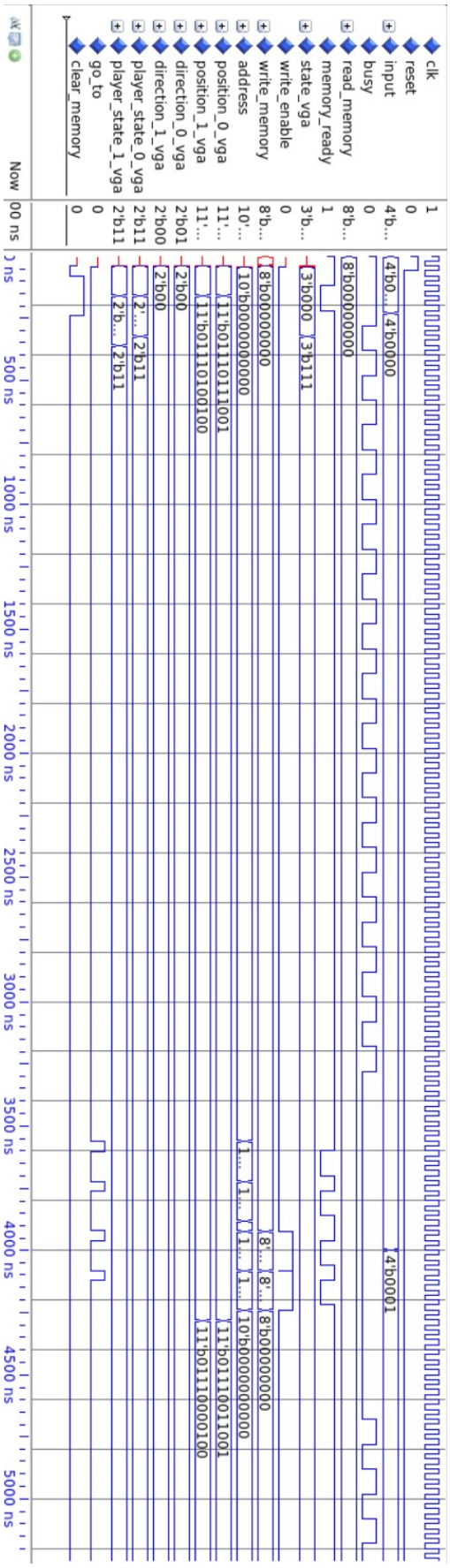


Figure 10: Simulation results from the game engine to test the wait_state and the states with memory (1).

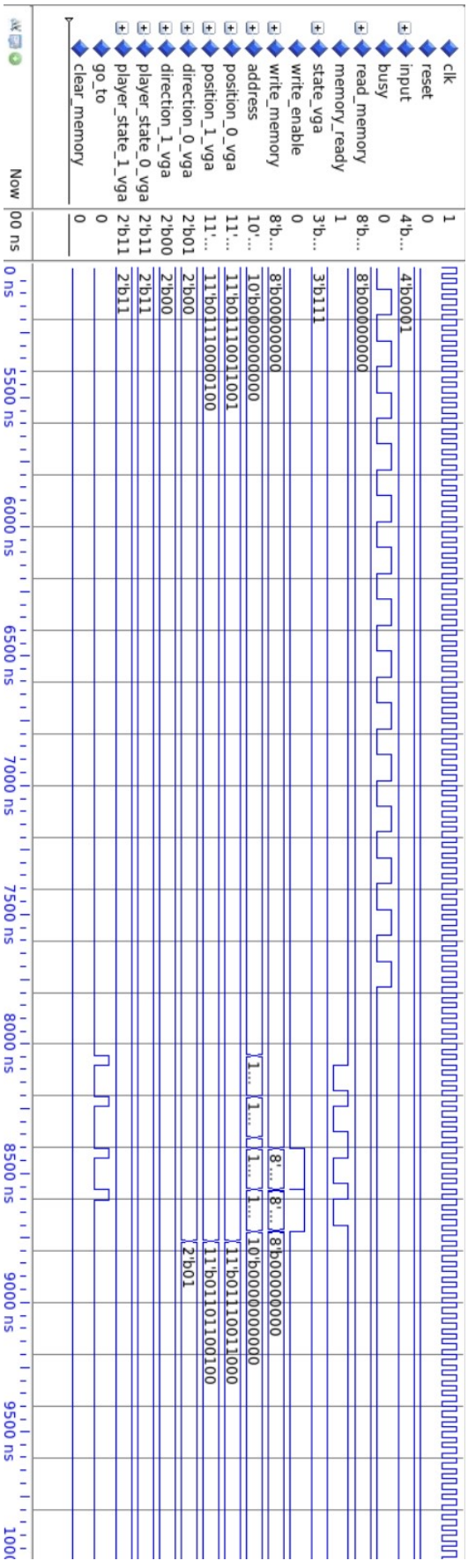


Figure 11: Simulation results from the game engine to test the wait_state and the states with memory (2).

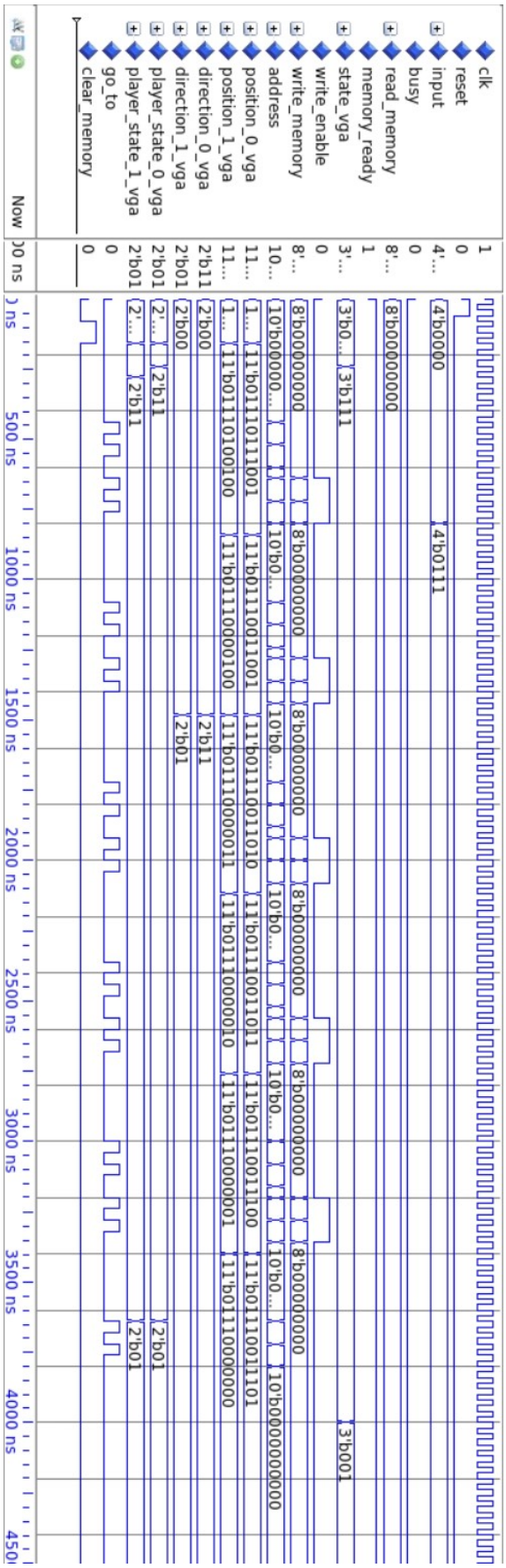


Figure 12: Simulation results from the game engine to test the left and right border.

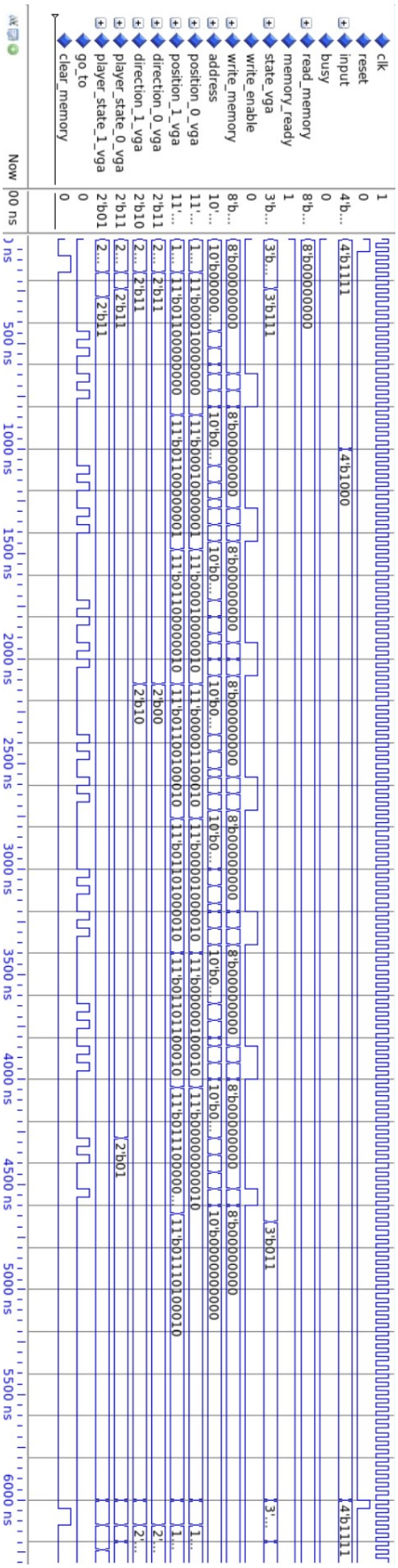


Figure 13: Simulation results from the game engine to test the upper and lower border (1).

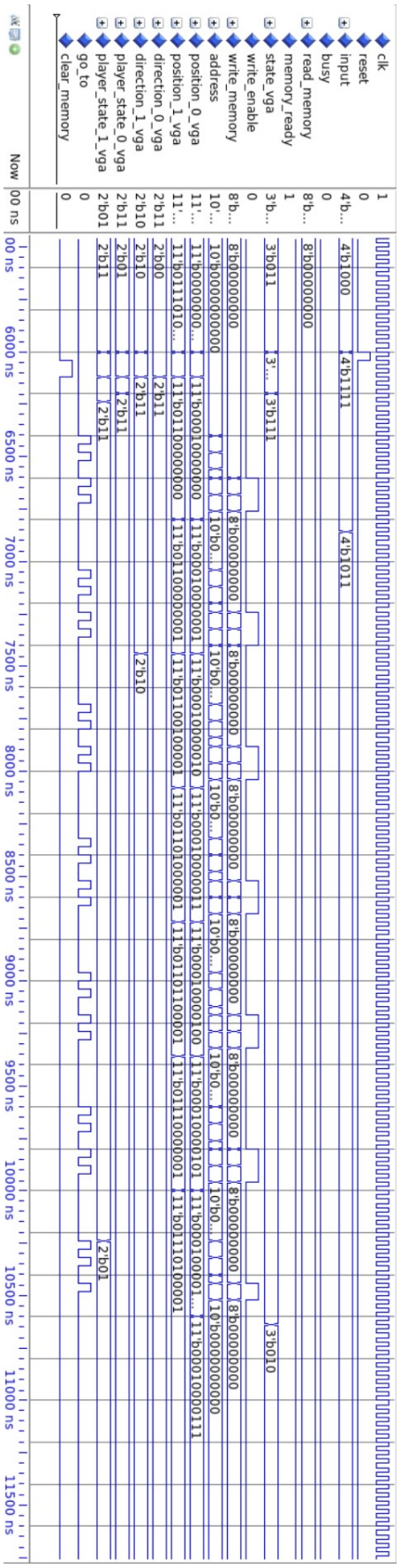


Figure 14: Simulation results from the game engine to test the upper and lower border (2).

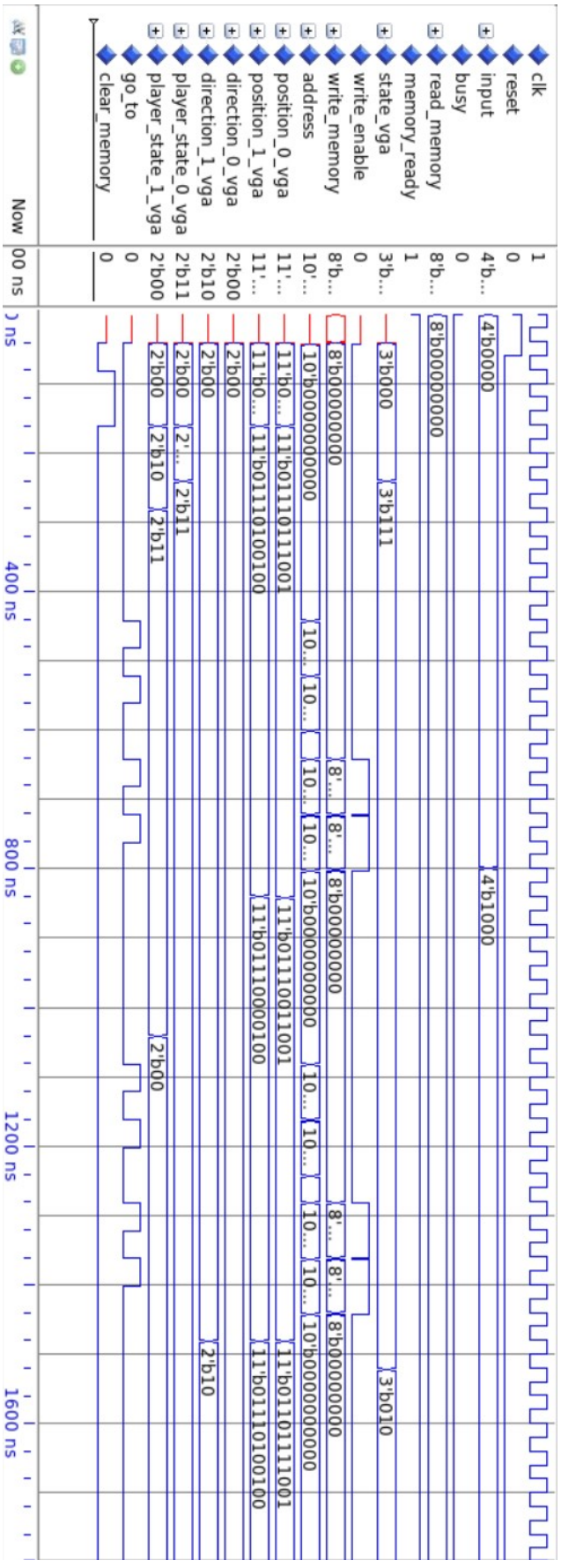


Figure 15: Simulation results from the game engine to test if a player can go back.

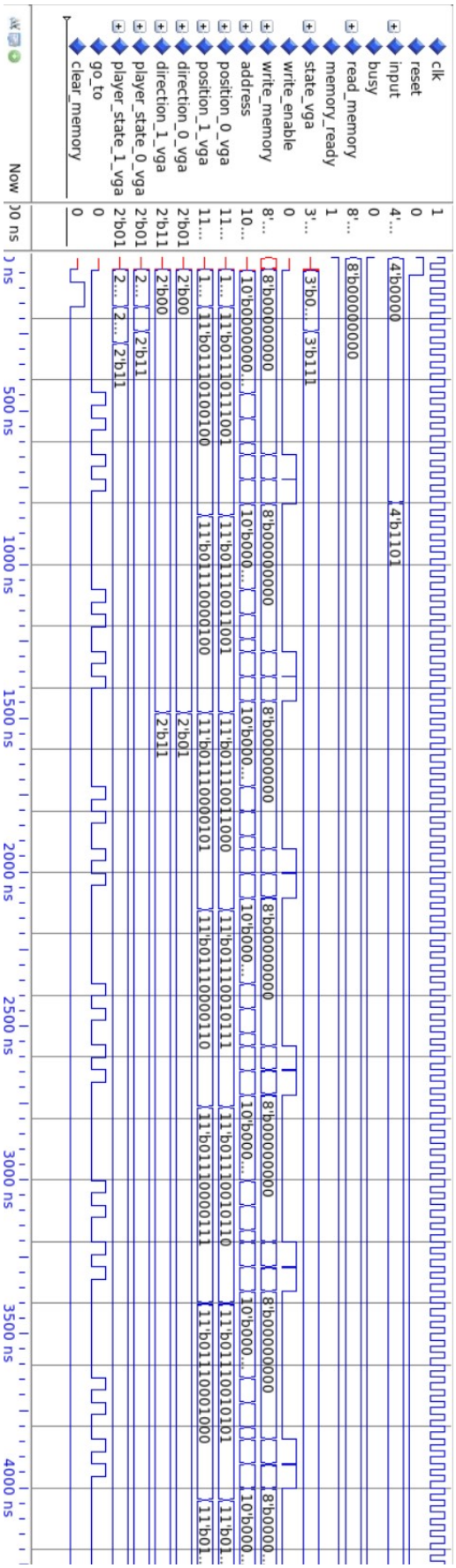


Figure 16: Simulation results from the game engine to test the head-on collision at the border of a cell (1).

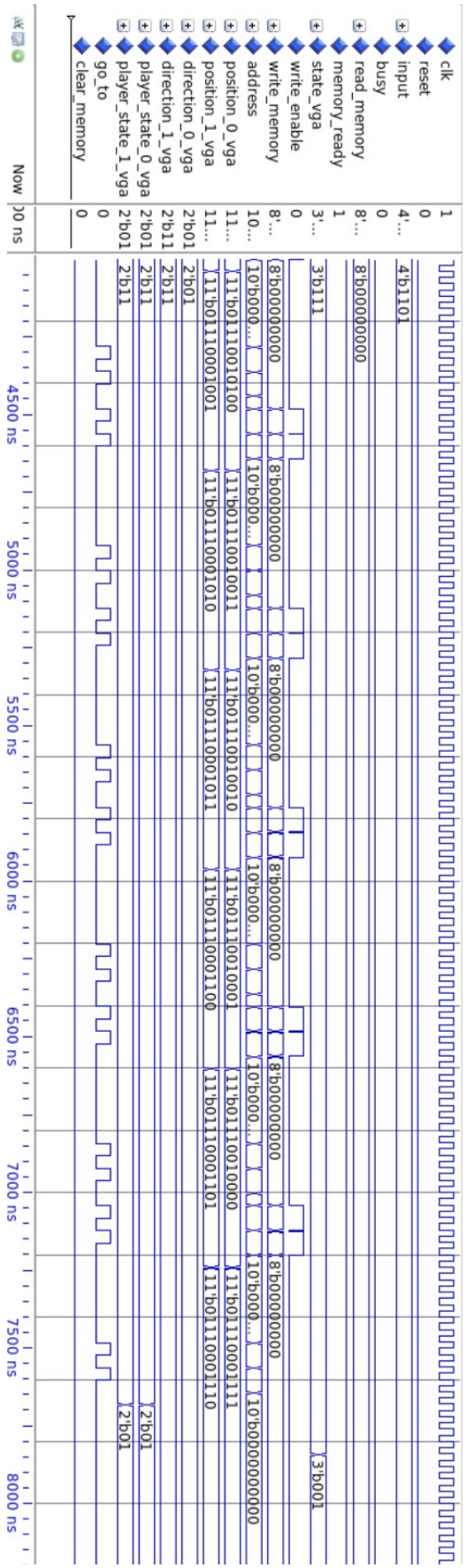


Figure 17: Simulation results from the game engine to test the head-on collision at the border of a cell (2).

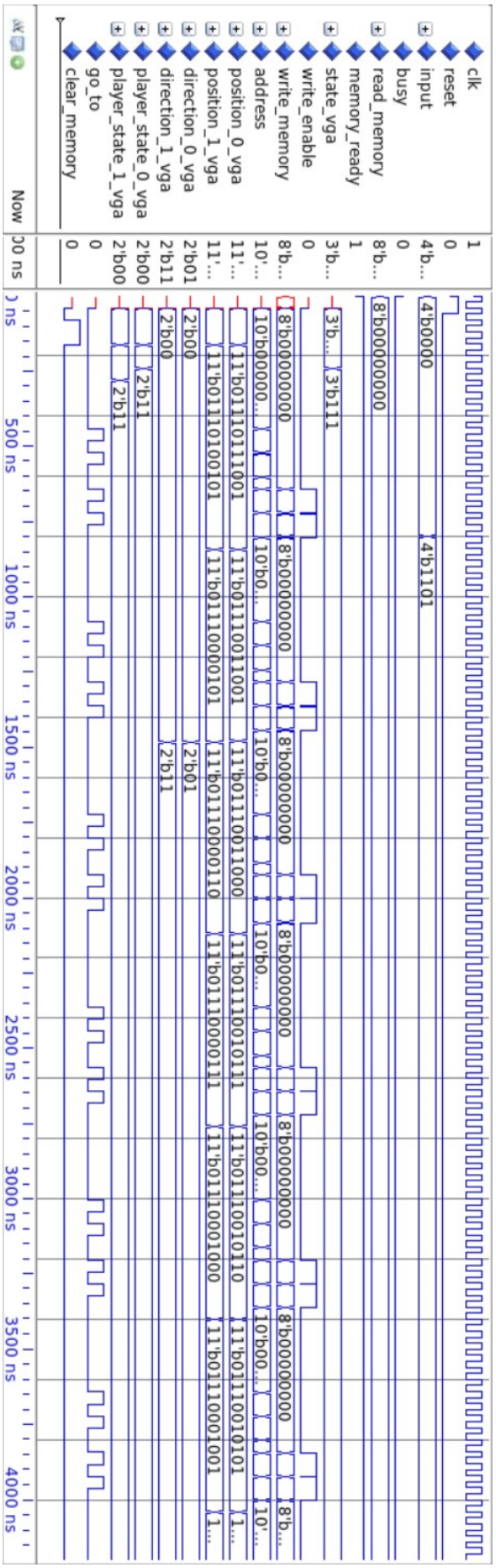


Figure 18: Simulation results from the game engine to test the head-on collision in the middle of a cell (1).

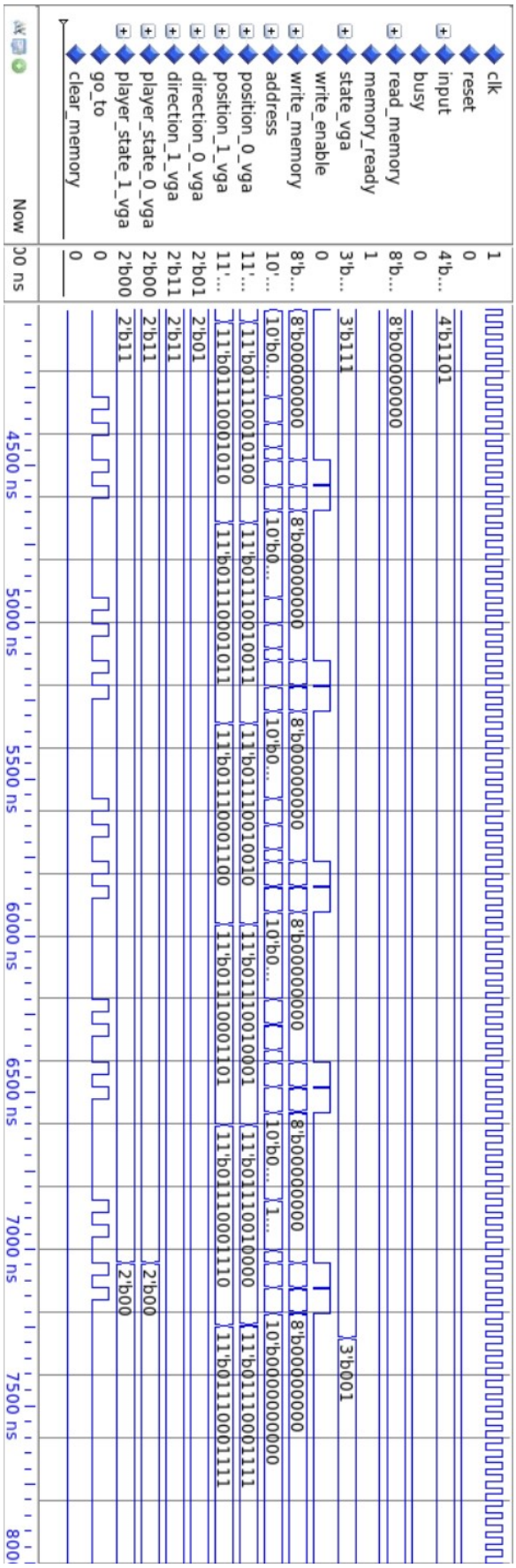


Figure 19: Simulation results from the game engine to test the head-on collision in the middle of a cell (2).

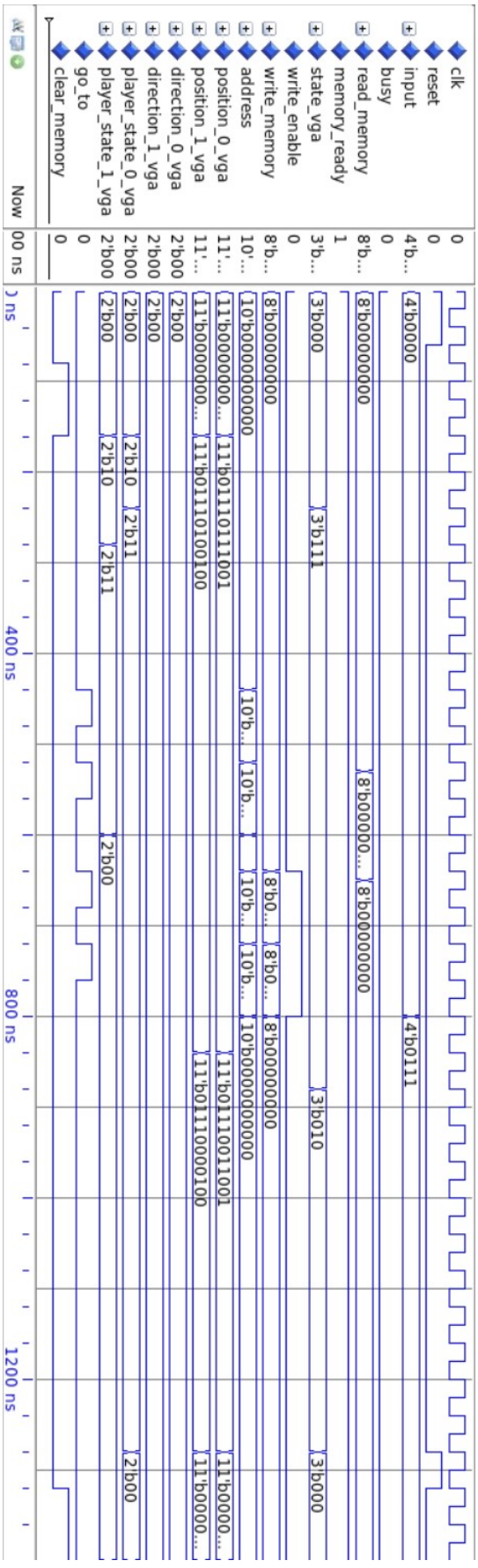


Figure 20: Simulation results from the game engine to test when a player moves into a wall(1).

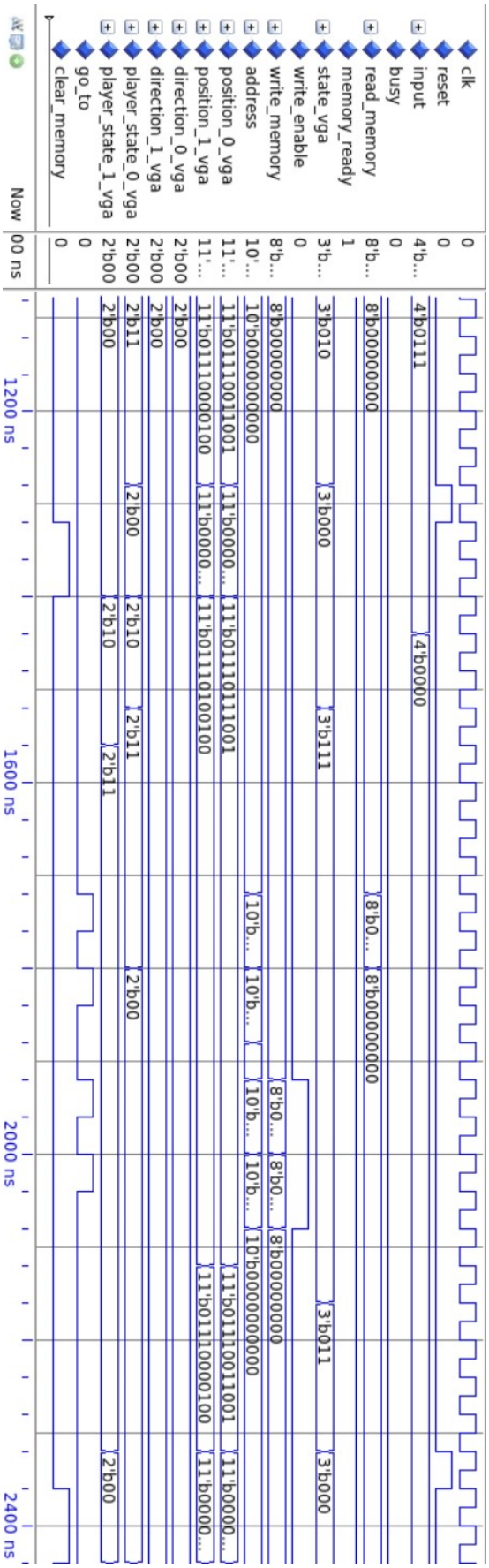


Figure 21: Simulation results from the game engine to test when a player moves into a wall(2).

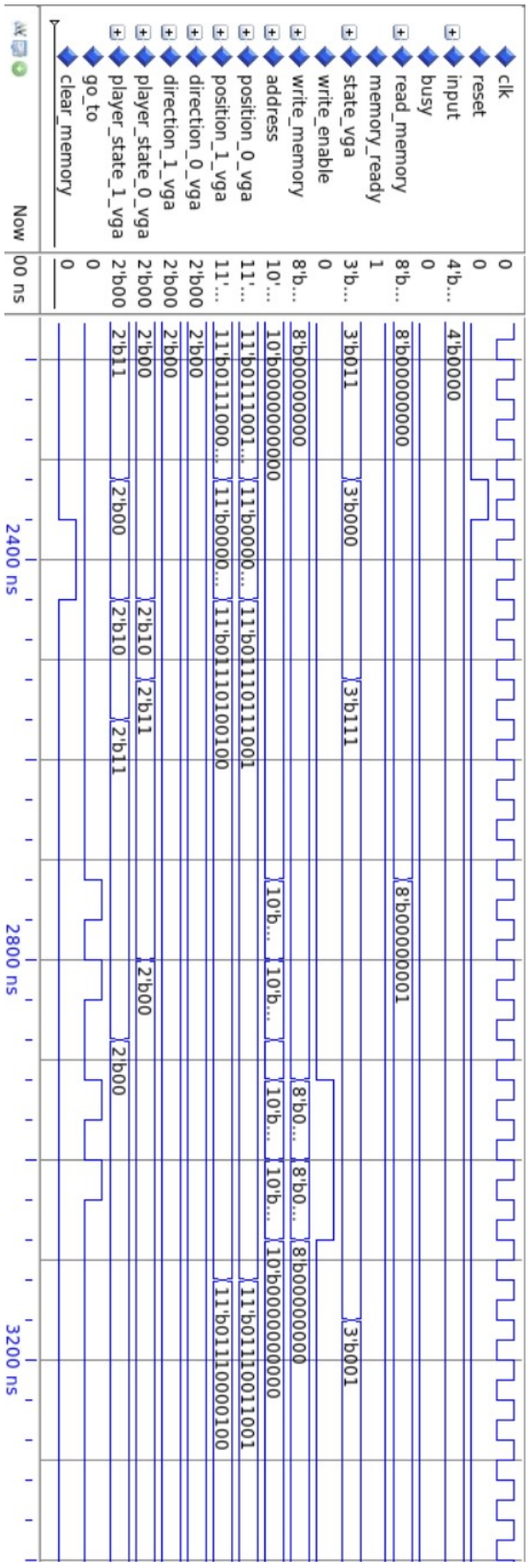


Figure 22: Simulation results from the game engine to test when a player moves into a wall(3).

B Figures

B.1 Playing field

This is the playing field with the players at their starting positions.

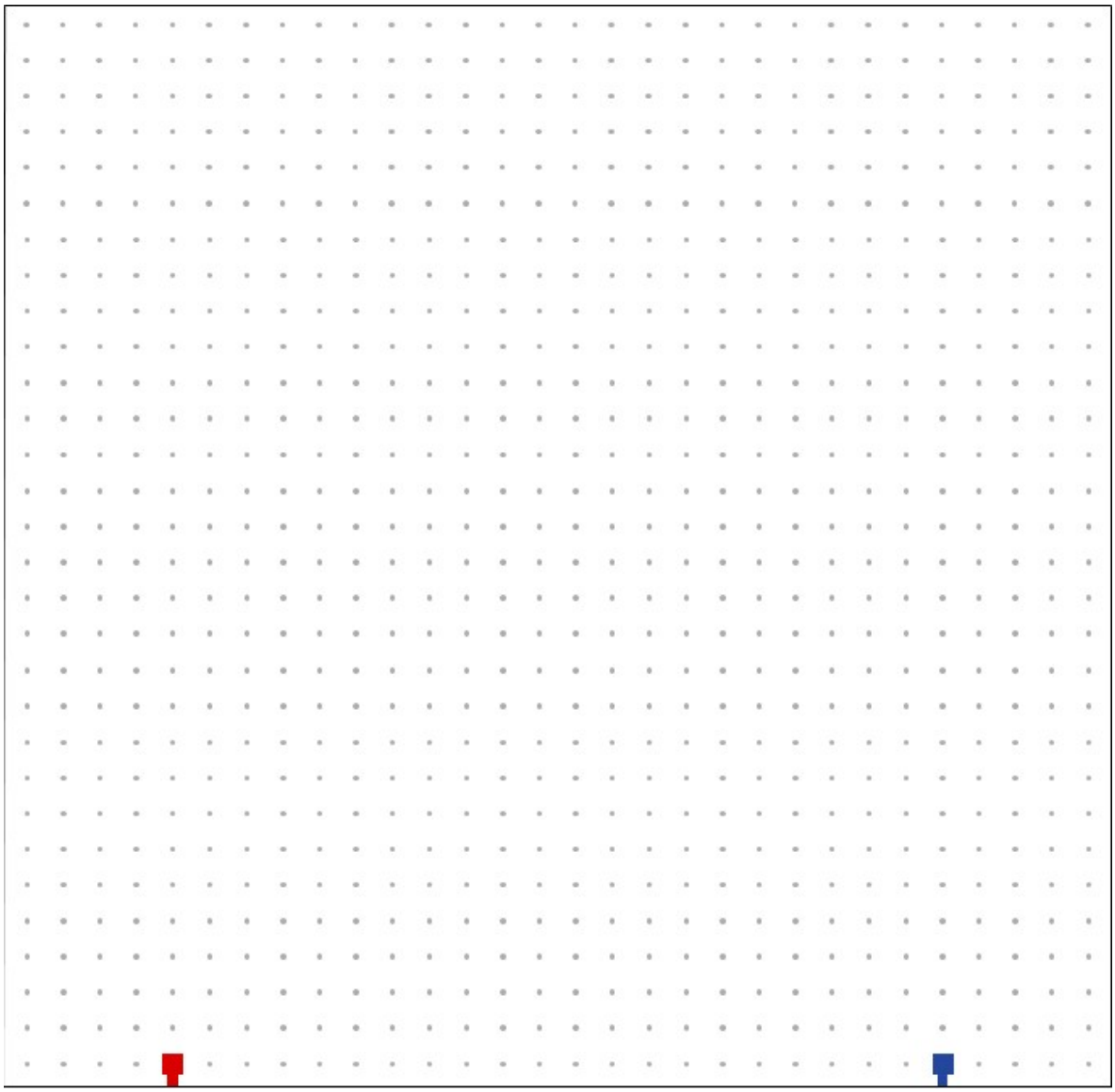
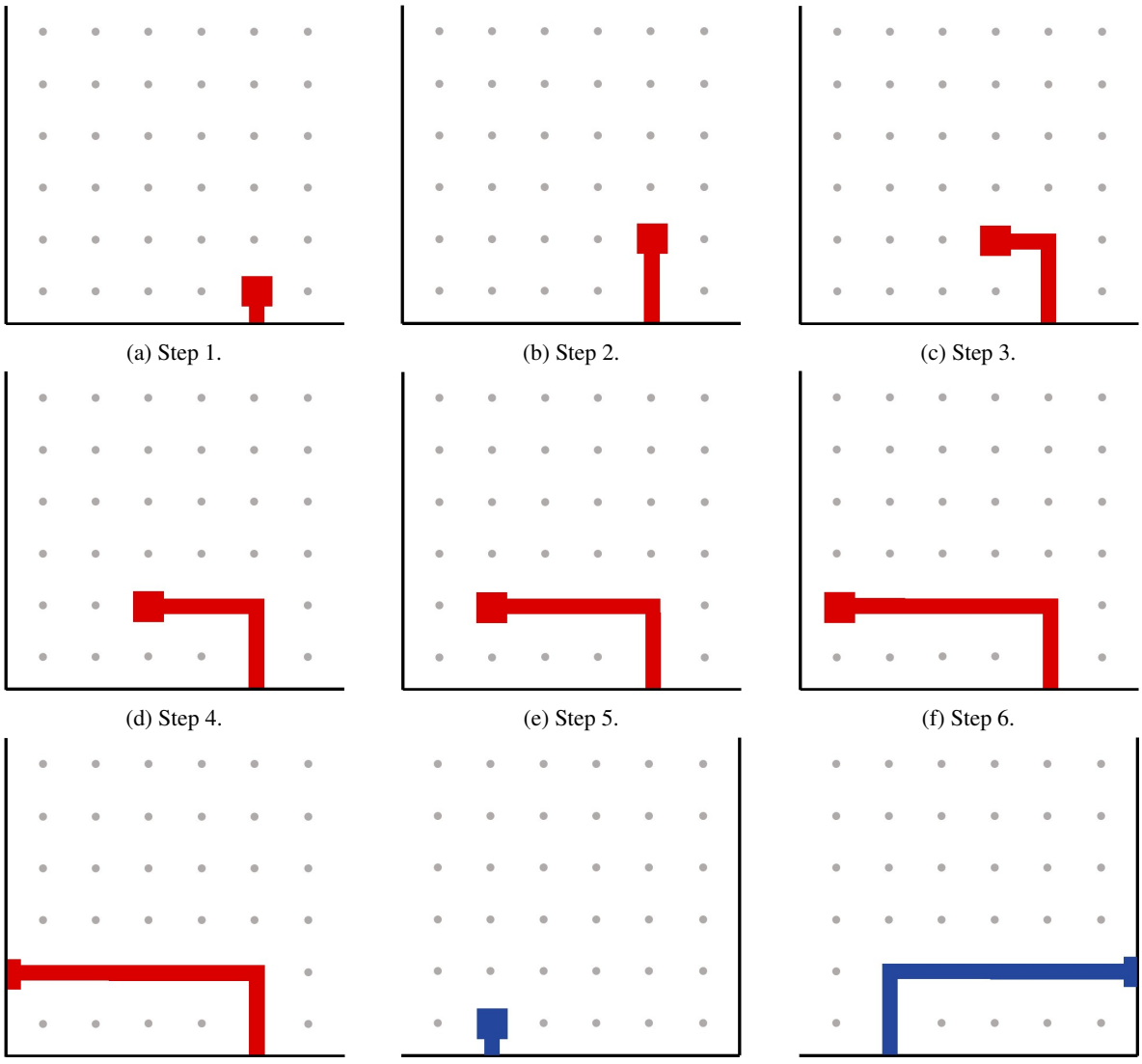


Figure 23: The starting positions of the players

B.2 Steps of the simulation

These are the steps of player 1 according to the simulation that checks the left and right border. The starting position and the final position of player 0 according to the same simulation.



(a) Step 1. (b) Step 2. (c) Step 3.

(d) Step 4. (e) Step 5. (f) Step 6.

(g) Step 7. (h) Start position of player 0. (i) Final position of player 0.

Figure 24: The steps taken in the simulation for player 1 and the begin and end position of player 0.

C VHDL code

C.1 busy_counter

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

entity busy_counter is
   port(clk                : in  std_logic;
        global_reset       : in  std_logic;
    game_engine_reset : in  std_logic;
```

```
8         busy               : in  std_logic;
9         busy_count         : out std_logic_vector(4 downto 0));
10 end busy_counter;
11
12 library IEEE;
13 use IEEE.std_logic_1164.ALL;
14 use IEEE.numeric_std.all;
15
16 architecture behaviour of busy_counter is
17
18   -- implemeted as a FSM with the count increasing when busy
       changes from high to low
19   type busy_counter_state_type is (reset, busy_low, busy_high,
       busy_high_to_low);
20   signal busy_counter_state, next_busy_counter_state:
       busy_counter_state_type;
21   signal unsigned_busy_count, next_unsigned_busy_count: unsigned
       (4 downto 0);
22
23 begin
24
25   process (clk)
26   begin
27     -- only count when the reset from the game engine and the
         global reset are '0'
28     if (clk'event and clk = '1') then
29       if (global_reset = '1' or game_engine_reset = '1') then
30         busy_counter_state <= reset;
31         unsigned_busy_count <= (others => '0');
32       else
33         busy_counter_state <= next_busy_counter_state;
34         unsigned_busy_count <= next_unsigned_busy_count;
35       end if;
36     end if;
37   end process;
38
39   process (busy, busy_counter_state)
40   begin
41     case busy_counter_state is
42
43       when reset =>
44         -- the counter is '0'
45         next_busy_counter_state <= busy_low;
46         next_unsigned_busy_count <= (others => '0');
47
48       when busy_low =>
49         -- the busy signal is '0'
```

```
50          -- the counter remains the same
51          next_unsigned_busy_count <= unsigned_busy_count;
52
53          -- when the busy signal goes to '1' the next state is
                'busy_high'
54          if (busy = '1') then
55            next_busy_counter_state <= busy_high;
56          else
57            next_busy_counter_state <= busy_low;
58          end if;
59
60        when busy_high =>
61          -- the busy signal is '1'
62          -- the counter remains the same
63          next_unsigned_busy_count <= unsigned_busy_count;
64
65          -- when the busy signal goes to '0' the next state is
                'busy_high_to_low'
66          if (busy = '0') then
67            next_busy_counter_state <= busy_high_to_low;
68          else
69            next_busy_counter_state <= busy_high;
70          end if;
71
72        when busy_high_to_low =>
73          -- the busy signal went from '1' to '0'
74          -- the counter is added by 1
75          next_unsigned_busy_count <= unsigned_busy_count + 1;
76
77          -- the next state is 'busy_low'
78          next_busy_counter_state <= busy_low;
79      end case;
80    end process;
81    busy_count <= std_logic_vector(unsigned_busy_count);
82
83  end behaviour;
```

VHDL/busy_counter.vhd

C.2 register

C.2.1 ge_register

```
1  library IEEE;
2  use IEEE.std_logic_1164.ALL;
3
4  entity ge_register is
```

```
5     port(clk, reset      : in   std_logic;
6    e_position_0  : in  std_logic;
7          e_position_1  : in  std_logic;
8          d_position_0  : in  std_logic_vector(10 downto 0);
9          d_position_1  : in  std_logic_vector(10 downto 0);
10         e_wallshape_0 : in  std_logic;
11         e_wallshape_1 : in  std_logic;
12         d_wallshape_0 : in  std_logic_vector(2 downto 0);
13         d_wallshape_1 : in  std_logic_vector(2 downto 0);
14         e_read_mem_0  : in  std_logic;
15         e_read_mem_1  : in  std_logic;
16         d_read_mem_0  : in  std_logic_vector(7 downto 0);
17         d_read_mem_1  : in  std_logic_vector(7 downto 0);
18         e_next_pos_0  : in  std_logic;
19         e_next_pos_1  : in  std_logic;
20         d_next_pos_0  : in  std_logic_vector(10 downto 0);
21         d_next_pos_1  : in  std_logic_vector(10 downto 0);
22         e_direction_0 : in  std_logic;
23         e_direction_1 : in  std_logic;
24         d_direction_0 : in  std_logic_vector(1 downto 0);
25         d_direction_1 : in  std_logic_vector(1 downto 0);
26         e_next_dir_0  : in  std_logic;
27         e_next_dir_1  : in  std_logic;
28         d_next_dir_0  : in  std_logic_vector(1 downto 0);
29         d_next_dir_1  : in  std_logic_vector(1 downto 0);
30         e_p_state_0   : in  std_logic;
31         e_p_state_1   : in  std_logic;
32         d_p_state_0   : in  std_logic_vector(1 downto 0);
33         d_p_state_1   : in  std_logic_vector(1 downto 0);
34         q_position_0  : out std_logic_vector(10 downto 0);
35         q_position_1  : out std_logic_vector(10 downto 0);
36         q_wallshape_0 : out std_logic_vector(2 downto 0);
37         q_wallshape_1 : out std_logic_vector(2 downto 0);
38         q_read_mem_0  : out std_logic_vector(7 downto 0);
39         q_read_mem_1  : out std_logic_vector(7 downto 0);
40         q_next_pos_0  : out std_logic_vector(10 downto 0);
41         q_next_pos_1  : out std_logic_vector(10 downto 0);
42         q_direction_0 : out std_logic_vector(1 downto 0);
43         q_direction_1 : out std_logic_vector(1 downto 0);
44         q_next_dir_0  : out std_logic_vector(1 downto 0);
45         q_next_dir_1  : out std_logic_vector(1 downto 0);
46         q_p_state_0   : out std_logic_vector(1 downto 0);
47         q_p_state_1   : out std_logic_vector(1 downto 0));
48 end ge_register;
```

VHDL/ge_register.vhd

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

architecture structural of ge_register is
   component reg_8
      port(clk   : in  std_logic;
           reset : in  std_logic;
           e     : in  std_logic;
           d     : in  std_logic_vector(7 downto 0);
           q     : out std_logic_vector(7 downto 0));
   end component;

   component reg_3
      port(clk   : in  std_logic;
           reset : in  std_logic;
           e     : in  std_logic;
           d     : in  std_logic_vector(2 downto 0);
           q     : out std_logic_vector(2 downto 0));
   end component;

   component reg_2
      port(clk   : in  std_logic;
           reset : in  std_logic;
           e     : in  std_logic;
           d     : in  std_logic_vector(1 downto 0);
           q     : out std_logic_vector(1 downto 0));
   end component;

   component reg_11
      port(clk   : in  std_logic;
           reset : in  std_logic;
           e     : in  std_logic;
           d     : in  std_logic_vector(10 downto 0);
           q     : out std_logic_vector(10 downto 0));
   end component;

begin

  pos0:reg_11 port map          ( clk  => clk,
          reset=> reset,
          e    => e_position_0,
          d    => d_position_0,
          q    => q_position_0);
  pos1:reg_11 port map            ( clk  => clk,
          reset=> reset,
          e    => e_position_1,
          d    => d_position_1,
```

```
48          q    => q_position_1);
49  next_pos0:reg_11 port map       ( clk  => clk,
50          reset=> reset,
51          e    => e_next_pos_0,
52          d    => d_next_pos_0,
53          q    => q_next_pos_0);
54  next_pos1:reg_11 port map       ( clk  => clk,
55          reset=> reset,
56          e    => e_next_pos_1,
57          d    => d_next_pos_1,
58          q    => q_next_pos_1);
59  wall_0:reg_3 port map           ( clk  => clk,
60          reset=> reset,
61          e    => e_wallshape_0,
62          d    => d_wallshape_0,
63          q    => q_wallshape_0);
64  wall_1:reg_3 port map           ( clk  => clk,
65          reset=> reset,
66          e    => e_wallshape_1,
67          d    => d_wallshape_1,
68          q    => q_wallshape_1);
69  r_mem_0:reg_8 port map           ( clk  => clk,
70          reset=> reset,
71          e    => e_read_mem_0,
72          d    => d_read_mem_0,
73          q    => q_read_mem_0);
74  r_mem_1:reg_8 port map           ( clk  => clk,
75          reset=> reset,
76          e    => e_read_mem_1,
77          d    => d_read_mem_1,
78          q    => q_read_mem_1);
79  dir_0:reg_2 port map            ( clk  => clk,
80          reset=> reset,
81          e    => e_direction_0,
82          d    => d_direction_0,
83          q    => q_direction_0);
84  dir_1:reg_2 port map            ( clk  => clk,
85          reset=> reset,
86          e    => e_direction_1,
87          d    => d_direction_1,
88          q    => q_direction_1);
89  n_dir_0:reg_2 port map           ( clk  => clk,
90          reset=> reset,
91          e    => e_next_dir_0,
92          d    => d_next_dir_0,
93          q    => q_next_dir_0);
94  n_dir_1:reg_2 port map           ( clk  => clk,
```

```
95            reset=> reset,
96            e     => e_next_dir_1,
97            d     => d_next_dir_1,
98            q     => q_next_dir_1);
99    p_state_0:reg_2 port map          ( clk   => clk,
100           reset=> reset,
101           e     => e_p_state_0,
102           d     => d_p_state_0,
103           q     => q_p_state_0);
104   p_state_1:reg_2 port map          ( clk   => clk,
105           reset=> reset,
106           e     => e_p_state_1,
107           d     => d_p_state_1,
108           q     => q_p_state_1);
109
110 end structural;
```

VHDL/ge_register–structural.vhd

C.2.2 reg_2

```
1  library IEEE;
2  use IEEE.std_logic_1164.ALL;
3
4  entity reg_2 is
5     port(clk   : in  std_logic;
6          reset : in  std_logic;
7          e     : in  std_logic;
8          d     : in  std_logic_vector(1 downto 0);
9          q     : out std_logic_vector(1 downto 0));
10 end reg_2;
```

VHDL/reg_2.vhd

```
1  library IEEE;
2  use IEEE.std_logic_1164.ALL;
3
4  architecture behaviour of reg_2 is
5  begin
6
7    process (clk)
8    begin
9      if (clk'event and clk = '1') then
10       if (reset = '1') then
11         q <= "00";
12       elsif (e = '1') then
13         q <= d;
```

```
      end if;
    end if;
  end process;

end behaviour;
```

VHDL/reg_2–behaviour.vhd

C.2.3 reg_3

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

entity reg_3 is
   port(clk   : in  std_logic;
        reset : in  std_logic;
        e     : in  std_logic;
        d     : in  std_logic_vector(2 downto 0);
        q     : out std_logic_vector(2 downto 0));
end reg_3;
```

VHDL/reg_3.vhd

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

architecture behaviour of reg_3 is
begin

  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        q <= "000";
      elsif (e = '1') then
        q <= d;
      end if;
    end if;
  end process;

end behaviour;
```

VHDL/reg_3–behaviour.vhd

C.2.4 reg_8

```
library IEEE;
```

```
use IEEE.std_logic_1164.ALL;

entity reg_8 is
   port(clk   : in  std_logic;
        reset : in  std_logic;
        e     : in  std_logic;
        d     : in  std_logic_vector(7 downto 0);
        q     : out std_logic_vector(7 downto 0));
end reg_8;
```

VHDL/reg_8.vhd

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

architecture behaviour of reg_8 is
begin

  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        q <= "00000000";
      elsif (e = '1') then
        q <= d;
      end if;
    end if;
  end process;

end behaviour;
```

VHDL/reg_8–behaviour.vhd

C.2.5 reg_11

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

entity reg_11 is
   port(clk   : in  std_logic;
        reset : in  std_logic;
        e     : in  std_logic;
        d     : in  std_logic_vector(10 downto 0);
        q     : out std_logic_vector(10 downto 0));
end reg_11;
```

VHDL/reg_11.vhd

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

architecture behaviour of reg_11 is
begin

  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        q <= "00000000000";
      elsif (e = '1') then
        q <= d;
      end if;
    end if;
  end process;

end behaviour;
```

VHDL/reg_11–behaviour.vhd

C.3 game_engine

```vhdl
library IEEE;
use IEEE.std_logic_1164.ALL;
use ieee.numeric_std.all;

entity game_engine is
   port(clk                : in  std_logic;
        reset              : in  std_logic;
        input              : in  std_logic_vector(3 downto 0);
        busy               : in  std_logic;
        read_memory        : in  std_logic_vector(7 downto 0);
        memory_ready       : in  std_logic;
        state_vga          : out std_logic_vector(2 downto 0);
        write_enable       : out std_logic;
        write_memory       : out std_logic_vector(7 downto 0);
        address            : out std_logic_vector(9 downto 0);
        position_0_vga     : out std_logic_vector(10 downto 0);
        position_1_vga     : out std_logic_vector(10 downto 0);
        direction_0_vga    : out std_logic_vector(1 downto 0);
        direction_1_vga    : out std_logic_vector(1 downto 0);
        player_state_0_vga : out std_logic_vector(1 downto 0);
        player_state_1_vga : out std_logic_vector(1 downto 0);
    go_to            : out std_logic;
    clear_memory       : out std_logic);
```

```
24 end game_engine;
```

VHDL/game_engine.vhd

```
1  library IEEE;
2  use IEEE.std_logic_1164.ALL;
3  use ieee.numeric_std.all;
4
5  architecture behaviour of game_engine is
6    type game_state is (reset_state, want_to_load, loading_state,
       get_ready, read_inputs, wall_shape, check_border,
       want_to_read_0, want_to_read_1, read_memory_player_0,
       read_memory_player_1, check_collision, check_who_won,
       wait_state, want_to_write_0, want_to_write_1,
       write_memory_player_0, write_memory_player_1, change_data,
       player_0_won, player_1_won, tie, player_0_ready,
       player_1_ready, busy_reset);
7
8    signal state, new_state: game_state;
9    signal direction_0, direction_1, next_direction_0,
       next_direction_1 : std_logic_vector(1 downto 0);
10   signal d_direction_0, d_direction_1, d_next_direction_0,
       d_next_direction_1 : std_logic_vector(1 downto 0);
11   signal position_0, position_1, next_position_0, next_position_1
       : std_logic_vector (10 downto 0);
12   signal d_position_0, d_position_1, d_next_position_0,
       d_next_position_1 : std_logic_vector (10 downto 0);
13   signal wallshape_0, wallshape_1 : std_logic_vector (2 downto 0);
14   signal d_wallshape_0, d_wallshape_1 : std_logic_vector (2 downto
       0);
15   signal read_memory_0, read_memory_1 : std_logic_vector (7 downto
       0);
16   signal d_read_memory_0, d_read_memory_1 : std_logic_vector (7
       downto 0);
17   signal player_0_state, player_1_state: std_logic_vector (1
       downto 0);
18   signal d_player_0_state, d_player_1_state: std_logic_vector (1
       downto 0);
19   signal e_position_0, e_position_1, e_wallshape_0, e_wallshape_1,
       e_read_memory_0, e_read_memory_1, e_next_position_0,
       e_next_position_1, e_direction_0, e_direction_1,
       e_next_direction_0, e_next_direction_1, e_player_0_state,
       e_player_1_state: std_logic;
20   signal busy_counter_reset: std_logic;
21   signal unsigned_busy_count: std_logic_vector(4 downto 0);
22
```

```
23  component busy_counter is
24  port(clk                : in  std_logic;
25     global_reset      : in  std_logic;
26     game_engine_reset : in  std_logic;
27         busy              : in  std_logic;
28         busy_count        : out std_logic_vector(4 downto 0));
29  end component;
30
31
32  component ge_register is
33    port(clk, reset   : in  std_logic;
34       e_position_0  : in  std_logic;
35       e_position_1  : in  std_logic;
36       d_position_0  : in  std_logic_vector(10 downto 0);
37       d_position_1  : in  std_logic_vector(10 downto 0);
38       e_wallshape_0 : in  std_logic;
39       e_wallshape_1 : in  std_logic;
40       d_wallshape_0 : in  std_logic_vector(2 downto 0);
41       d_wallshape_1 : in  std_logic_vector(2 downto 0);
42       e_read_mem_0  : in  std_logic;
43       e_read_mem_1  : in  std_logic;
44       d_read_mem_0  : in  std_logic_vector(7 downto 0);
45       d_read_mem_1  : in  std_logic_vector(7 downto 0);
46       e_next_pos_0  : in  std_logic;
47       e_next_pos_1  : in  std_logic;
48       d_next_pos_0  : in  std_logic_vector(10 downto 0);
49       d_next_pos_1  : in  std_logic_vector(10 downto 0);
50       e_direction_0 : in  std_logic;
51       e_direction_1 : in  std_logic;
52       d_direction_0 : in  std_logic_vector(1 downto 0);
53       d_direction_1 : in  std_logic_vector(1 downto 0);
54       e_next_dir_0  : in  std_logic;
55       e_next_dir_1  : in  std_logic;
56       d_next_dir_0  : in  std_logic_vector(1 downto 0);
57       d_next_dir_1  : in  std_logic_vector(1 downto 0);
58       e_p_state_0   : in  std_logic;
59       e_p_state_1   : in  std_logic;
60       d_p_state_0   : in  std_logic_vector(1 downto 0);
61       d_p_state_1   : in  std_logic_vector(1 downto 0);
62       q_position_0  : out std_logic_vector(10 downto 0);
63       q_position_1  : out std_logic_vector(10 downto 0);
64       q_wallshape_0 : out std_logic_vector(2 downto 0);
65       q_wallshape_1 : out std_logic_vector(2 downto 0);
66       q_read_mem_0  : out std_logic_vector(7 downto 0);
67       q_read_mem_1  : out std_logic_vector(7 downto 0);
68       q_next_pos_0  : out std_logic_vector(10 downto 0);
69       q_next_pos_1  : out std_logic_vector(10 downto 0);
```

```
70        q_direction_0 : out std_logic_vector(1 downto 0);
71        q_direction_1 : out std_logic_vector(1 downto 0);
72        q_next_dir_0  : out std_logic_vector(1 downto 0);
73        q_next_dir_1  : out std_logic_vector(1 downto 0);
74        q_p_state_0   : out std_logic_vector(1 downto 0);
75        q_p_state_1   : out std_logic_vector(1 downto 0));
76    end component;
77
78  begin
79
80  reg: ge_register port map (clk => clk,
81        reset         => reset,
82        e_position_0  => e_position_0,
83        e_position_1  => e_position_1,
84        d_position_0  => d_position_0,
85        d_position_1  => d_position_1,
86        e_wallshape_0 => e_wallshape_0,
87        e_wallshape_1 => e_wallshape_1,
88        d_wallshape_0 => d_wallshape_0,
89        d_wallshape_1 => d_wallshape_1,
90        e_read_mem_0  => e_read_memory_0,
91        e_read_mem_1  => e_read_memory_1,
92        d_read_mem_0  => d_read_memory_0,
93        d_read_mem_1  => d_read_memory_1,
94        e_next_pos_0  => e_next_position_0,
95        e_next_pos_1  => e_next_position_1,
96        d_next_pos_0  => d_next_position_0,
97        d_next_pos_1  => d_next_position_1,
98        e_direction_0 => e_direction_0,
99        e_direction_1 => e_direction_1,
100       d_direction_0 => d_direction_0,
101       d_direction_1 => d_direction_1,
102       e_next_dir_0  => e_next_direction_0,
103       e_next_dir_1  => e_next_direction_1,
104       d_next_dir_0  => d_next_direction_0,
105       d_next_dir_1  => d_next_direction_1,
106       e_p_state_0   => e_player_0_state,
107       e_p_state_1   => e_player_1_state,
108       d_p_state_0   => d_player_0_state,
109       d_p_state_1   => d_player_1_state,
110       q_position_0  => position_0,
111       q_position_1  => position_1,
112       q_wallshape_0 => wallshape_0,
113       q_wallshape_1 => wallshape_1,
114       q_read_mem_0  => read_memory_0,
115       q_read_mem_1  => read_memory_1,
116       q_next_pos_0  => next_position_0,
```

```
      q_next_pos_1  => next_position_1,
      q_direction_0 => direction_0,
      q_direction_1 => direction_1,
      q_next_dir_0  => next_direction_0,
      q_next_dir_1  => next_direction_1,
      q_p_state_0   => player_0_state,
      q_p_state_1   => player_1_state);

counter: busy_counter port map (clk => clk,
      global_reset => reset,
      game_engine_reset => busy_counter_reset,
      busy => busy,
      busy_count => unsigned_busy_count);

-- outputs from the register to the graphics engine
position_0_vga  <= position_0;
position_1_vga  <= position_1;
direction_0_vga <= direction_0;
direction_1_vga <= direction_1;
player_state_0_vga <= player_0_state;
player_state_1_vga <= player_1_state;



updates:  process (clk)
  begin
    if (clk'event and clk = '1') then
      -- go to the reset state when the reset button is pressed
      if (reset = '1') then
        state <= reset_state;
      -- go to the next state according to the FSM
      else
        state <= new_state;
      end if;
    end if;
  end process;

create_next_state:  process (state, new_state, reset, input, busy,
   read_memory, memory_ready, clk, unsigned_busy_count,
   direction_0, direction_1, next_direction_0, next_direction_1,
   position_0, position_1, next_position_0, next_position_1,
   wallshape_0, wallshape_1, read_memory_0, read_memory_1,
   player_0_state, player_1_state, e_position_0, e_position_1,
   e_wallshape_0, e_wallshape_1, e_read_memory_0, e_read_memory_1,
   e_next_position_0, e_next_position_1, e_direction_0,
   e_direction_1, e_next_direction_0, e_next_direction_1,
   e_player_0_state, e_player_1_state )
```

```
155  begin
156    case state is
157      when reset_state =>
158        -- in this state all the values are set to zero to reset
               everything
159        state_vga           <= "000";
160        write_enable        <= '0';
161        write_memory        <= "00000000";
162        address           <= "0000000000";
163        go_to           <= '0';
164        busy_counter_reset      <= '0';
165        clear_memory        <= '0';
166
167        e_position_0        <= '0';
168        e_position_1        <= '0';
169        e_wallshape_0       <= '0';
170        e_wallshape_1       <= '0';
171        e_read_memory_0       <= '0';
172        e_read_memory_1       <= '0';
173        e_next_position_0     <= '0';
174        e_next_position_1     <= '0';
175        e_direction_0       <= '0';
176        e_direction_1       <= '0';
177        e_next_direction_0      <= '0';
178        e_next_direction_1      <= '0';
179        e_player_0_state      <= '0';
180        e_player_1_state      <= '0';
181
182        d_position_0        <= (others => '0');
183        d_position_1        <= (others => '0');
184        d_wallshape_0       <= (others => '0');
185        d_wallshape_1       <= (others => '0');
186        d_read_memory_0       <= (others => '0');
187        d_read_memory_1       <= (others => '0');
188        d_next_position_0     <= (others => '0');
189        d_next_position_1     <= (others => '0');
190        d_direction_0       <= (others => '0');
191        d_direction_1       <= (others => '0');
192        d_next_direction_0      <= (others => '0');
193        d_next_direction_1      <= (others => '0');
194        d_player_0_state      <= (others => '0');
195        d_player_1_state      <= (others => '0');
196
197        -- go to the state 'want_to_load' next
198        new_state <= want_to_load;
199
200      when want_to_load =>
```

```
201         -- let the memory module know that the memory has to be
                cleared
202         state_vga           <= "000";
203         write_enable        <= '0';
204         write_memory        <= "00000000";
205         address           <= "0000000000";
206         busy_counter_reset      <= '0';
207         go_to           <= '0';
208         clear_memory        <= '1';
209 
210         e_position_0        <= '0';
211         e_position_1        <= '0';
212         e_wallshape_0       <= '0';
213         e_wallshape_1       <= '0';
214         e_read_memory_0       <= '0';
215         e_read_memory_1       <= '0';
216         e_next_position_0     <= '0';
217         e_next_position_1     <= '0';
218         e_direction_0       <= '0';
219         e_direction_1       <= '0';
220         e_next_direction_0      <= '0';
221         e_next_direction_1      <= '0';
222         e_player_0_state      <= '0';
223         e_player_1_state      <= '0';
224 
225         d_position_0        <= (others => '0');
226         d_position_1        <= (others => '0');
227         d_wallshape_0       <= (others => '0');
228         d_wallshape_1       <= (others => '0');
229         d_read_memory_0       <= (others => '0');
230         d_read_memory_1       <= (others => '0');
231         d_next_position_0     <= (others => '0');
232         d_next_position_1     <= (others => '0');
233         d_direction_0       <= (others => '0');
234         d_direction_1       <= (others => '0');
235         d_next_direction_0      <= (others => '0');
236         d_next_direction_1      <= (others => '0');
237         d_player_0_state      <= (others => '0');
238         d_player_1_state      <= (others => '0');
239 
240         -- go to the state 'loading_state' next to continue to
                clear the memory
241         -- 'memory_ready' will still be '1', need to wait an extra
                clockcycle
242         new_state <= loading_state;
243 
244       when loading_state =>
```

```
245         -- continue to let the memory module know to clear the
               memory and check when the memory is done
246         state_vga           <= "000";
247         write_enable        <= '0';
248         write_memory        <= "00000000";
249         address           <= "0000000000";
250         busy_counter_reset      <= '0';
251         go_to           <= '0';
252         clear_memory        <= '1';
253 
254         -- initialize the position, direction and state of the
               players
255         e_position_0        <= '1';
256         e_position_1        <= '1';
257         e_direction_0       <= '1';
258         e_direction_1       <= '1';
259         e_player_0_state      <= '1';
260         e_player_1_state      <= '1';
261 
262         d_position_0        <= "01110111001";
263         d_position_1        <= "01110100100";
264         d_direction_0       <= "00";
265         d_direction_1       <= "00";
266         d_player_0_state      <= "10";
267         d_player_1_state      <= "10";
268 
269         e_wallshape_0       <= '0';
270         e_wallshape_1       <= '0';
271         e_read_memory_0       <= '0';
272         e_read_memory_1       <= '0';
273         e_next_position_0     <= '0';
274         e_next_position_1     <= '0';
275         e_next_direction_0      <= '0';
276         e_next_direction_1      <= '0';
277 
278         d_wallshape_0       <= (others => '0');
279         d_wallshape_1       <= (others => '0');
280         d_read_memory_0       <= (others => '0');
281         d_read_memory_1       <= (others => '0');
282         d_next_position_0     <= (others => '0');
283         d_next_position_1     <= (others => '0');
284         d_next_direction_0      <= (others => '0');
285         d_next_direction_1      <= (others => '0');
286 
287         -- when the memory is finished go to the next state:
               'get_ready' otherwise stay in this state
288         if (memory_ready = '1') then
```

```
289           new_state <= get_ready;
290         else
291           new_state <= loading_state;
292         end if;
293
294       when get_ready =>
295         -- wait for the player to press the button in the right
              direction: meaning they are ready to play
296         state_vga           <= "000";
297         write_enable        <= '0';
298         write_memory        <= "00000000";
299         address           <= "0000000000";
300         go_to           <= '0';
301         clear_memory        <= '0';
302
303         e_position_0        <= '0';
304         e_position_1        <= '0';
305         e_wallshape_0       <= '0';
306         e_wallshape_1       <= '0';
307         e_read_memory_0       <= '0';
308         e_read_memory_1       <= '0';
309         e_next_position_0     <= '0';
310         e_next_position_1     <= '0';
311         e_direction_0       <= '0';
312         e_direction_1       <= '0';
313         e_next_direction_0      <= '0';
314         e_next_direction_1      <= '0';
315         e_player_0_state      <= '0';
316         e_player_1_state      <= '0';
317
318         d_position_0        <= (others => '0');
319         d_position_1        <= (others => '0');
320         d_wallshape_0       <= (others => '0');
321         d_wallshape_1       <= (others => '0');
322         d_read_memory_0       <= (others => '0');
323         d_read_memory_1       <= (others => '0');
324         d_next_position_0     <= (others => '0');
325         d_next_position_1     <= (others => '0');
326         d_direction_0       <= (others => '0');
327         d_direction_1       <= (others => '0');
328         d_next_direction_0      <= (others => '0');
329         d_next_direction_1      <= (others => '0');
330         d_player_0_state      <= (others => '0');
331         d_player_1_state      <= (others => '0');
332
333         -- when player 0 is ready to play the next state is
              'player_0_ready'
```

```
334         if (input(1 downto 0) = direction_0) then
335           new_state         <= player_0_ready;
336         -- when player 1 is ready to play the next state is
              'player_1_ready'
337         elsif (input(3 downto 2) = direction_1) then
338           new_state       <= player_1_ready;
339         -- when neither player is ready the next state is this
              state
340         else
341           new_state       <= get_ready;
342         end if;
343
344       when player_0_ready =>
345         -- player 0 is ready to play
346         state_vga           <= "000";
347         write_enable        <= '0';
348         write_memory        <= "00000000";
349         address           <= "0000000000";
350         go_to           <= '0';
351         busy_counter_reset      <= '0';
352         clear_memory        <= '0';
353
354         -- change the player state of player 0 to let the VGA know
              that player 0 is ready to play
355         e_player_0_state      <= '1';
356         d_player_0_state      <= "11";
357
358         e_position_0        <= '0';
359         e_position_1        <= '0';
360         e_wallshape_0       <= '0';
361         e_wallshape_1       <= '0';
362         e_read_memory_0       <= '0';
363         e_read_memory_1       <= '0';
364         e_next_position_0     <= '0';
365         e_next_position_1     <= '0';
366         e_direction_0       <= '0';
367         e_direction_1       <= '0';
368         e_next_direction_0      <= '0';
369         e_next_direction_1      <= '0';
370         e_player_1_state      <= '0';
371
372         d_position_0        <= (others => '0');
373         d_position_1        <= (others => '0');
374         d_wallshape_0       <= (others => '0');
375         d_wallshape_1       <= (others => '0');
376         d_read_memory_0       <= (others => '0');
377         d_read_memory_1       <= (others => '0');
```

```
378         d_next_position_0     <= (others => '0');
379         d_next_position_1     <= (others => '0');
380         d_direction_0       <= (others => '0');
381         d_direction_1       <= (others => '0');
382         d_next_direction_0      <= (others => '0');
383         d_next_direction_1      <= (others => '0');
384         d_player_1_state      <= (others => '0');
385
386         -- when player 1 is ready go the the state 'wait_state' if
               player 1 is not ready the next state is this state
387         if (input(3 downto 2) = direction_1) then
388           new_state <= wait_state;
389         else
390           new_state <= player_0_ready;
391         end if;
392
393       when player_1_ready =>
394         -- player 1 is ready to play
395         state_vga           <= "000";
396         write_enable        <= '0';
397         write_memory        <= "00000000";
398         address           <= "0000000000";
399         go_to           <= '0';
400         busy_counter_reset      <= '0';
401         clear_memory        <= '0';
402
403         -- change the player state of player 1 to let the VGA know
               that player 1 is ready to play
404         e_player_1_state      <= '1';
405         d_player_1_state      <= "11";
406
407         e_position_0        <= '0';
408         e_position_1        <= '0';
409         e_wallshape_0       <= '0';
410         e_wallshape_1       <= '0';
411         e_read_memory_0       <= '0';
412         e_read_memory_1       <= '0';
413         e_next_position_0     <= '0';
414         e_next_position_1     <= '0';
415         e_direction_0       <= '0';
416         e_direction_1       <= '0';
417         e_next_direction_0      <= '0';
418         e_next_direction_1      <= '0';
419         e_player_0_state      <= '0';
420
421         d_position_0        <= (others => '0');
422         d_position_1        <= (others => '0');
```

```
423         d_wallshape_0       <= (others => '0');
424         d_wallshape_1       <= (others => '0');
425         d_read_memory_0       <= (others => '0');
426         d_read_memory_1       <= (others => '0');
427         d_next_position_0     <= (others => '0');
428         d_next_position_1     <= (others => '0');
429         d_direction_0       <= (others => '0');
430         d_direction_1       <= (others => '0');
431         d_next_direction_0      <= (others => '0');
432         d_next_direction_1      <= (others => '0');
433         d_player_0_state      <= (others => '0');
434
435         -- if player 0 is ready the next state is 'wait_state', if
               player 0 is not ready the next state is this state
436         if (input(1 downto 0) = direction_0) then
437           new_state <= wait_state;
438         else
439           new_state <= player_1_ready;
440         end if;
441
442
443
444       when wait_state =>
445         -- wait for a certain amount of busy signal cycles before
               going on
446         -- this determines how fast a player moves in the
               playingfield
447         state_vga           <= "111";
448         write_enable        <= '0';
449         write_memory        <= "00000000";
450         address           <= "0000000000";
451         go_to           <= '0';
452         busy_counter_reset      <= '0';
453         clear_memory        <= '0';
454
455         e_position_0        <= '0';
456         e_position_1        <= '0';
457         e_wallshape_0       <= '0';
458         e_wallshape_1       <= '0';
459         e_read_memory_0       <= '0';
460         e_read_memory_1       <= '0';
461         e_next_position_0     <= '0';
462         e_next_position_1     <= '0';
463         e_direction_0       <= '0';
464         e_direction_1       <= '0';
465         e_next_direction_0      <= '0';
466         e_next_direction_1      <= '0';
```

```
467
468
469         d_position_0        <= (others => '0');
470         d_position_1        <= (others => '0');
471         d_wallshape_0       <= (others => '0');
472         d_wallshape_1       <= (others => '0');
473         d_read_memory_0       <= (others => '0');
474         d_read_memory_1       <= (others => '0');
475         d_next_position_0     <= (others => '0');
476         d_next_position_1     <= (others => '0');
477         d_direction_0       <= (others => '0');
478         d_direction_1       <= (others => '0');
479         d_next_direction_0      <= (others => '0');
480         d_next_direction_1      <= (others => '0');
481
482         e_player_0_state      <= '1';
483         d_player_0_state      <= "11";
484         e_player_1_state      <= '1';
485         d_player_1_state      <= "11";
486
487         -- when waited long enough go the next state: read_inputs,
               otherwise keep waiting
488         if (unsigned( unsigned_busy_count) >= 16) then
489           new_state <= busy_reset;
490         else
491           -- new_state <= wait_state;
492           new_state <= busy_reset;
493         end if;
494
495       when busy_reset =>
496         -- reset the counter
497         busy_counter_reset      <= '1';
498
499         state_vga           <= "111";
500         write_enable        <= '0';
501         write_memory        <= "00000000";
502         address           <= "0000000000";
503         go_to           <= '0';
504         clear_memory        <= '0';
505
506
507         e_position_0        <= '0';
508         e_position_1        <= '0';
509         e_wallshape_0       <= '0';
510         e_wallshape_1       <= '0';
511         e_read_memory_0       <= '0';
512         e_read_memory_1       <= '0';
```

```
513         e_next_position_0     <= '0';
514         e_next_position_1     <= '0';
515         e_direction_0       <= '0';
516         e_direction_1       <= '0';
517         e_next_direction_0      <= '0';
518         e_next_direction_1      <= '0';
519         e_player_0_state      <= '0';
520         e_player_1_state      <= '0';
521
522         d_position_0        <= (others => '0');
523         d_position_1        <= (others => '0');
524         d_wallshape_0       <= (others => '0');
525         d_wallshape_1       <= (others => '0');
526         d_read_memory_0       <= (others => '0');
527         d_read_memory_1       <= (others => '0');
528         d_next_position_0     <= (others => '0');
529         d_next_position_1     <= (others => '0');
530         d_direction_0       <= (others => '0');
531         d_direction_1       <= (others => '0');
532         d_next_direction_0      <= (others => '0');
533         d_next_direction_1      <= (others => '0');
534         d_player_0_state      <= (others => '0');
535         d_player_1_state      <= (others => '0');
536
537         new_state <= read_inputs;
538
539       when read_inputs =>
540         -- read the inputs from the players and remember them
541         state_vga           <= "111";
542         write_enable        <= '0';
543         write_memory        <= "00000000";
544         address           <= "0000000000";
545         go_to           <= '0';
546         busy_counter_reset      <= '0';
547         clear_memory        <= '0';
548
549         -- remember the values of the input of the players in
               'next_direction_#player'
550         e_next_direction_0      <= '1';
551         e_next_direction_1      <= '1';
552
553         d_next_direction_0      <= input(1 downto 0);
554         d_next_direction_1      <= input(3 downto 2);


555
556         e_position_0        <= '0';
557         e_position_1        <= '0';
```

```
558         e_wallshape_0       <= '0';
559         e_wallshape_1       <= '0';
560         e_read_memory_0       <= '0';
561         e_read_memory_1       <= '0';
562         e_next_position_0     <= '0';
563         e_next_position_1     <= '0';
564         e_direction_0       <= '0';
565         e_direction_1       <= '0';
566         e_player_0_state      <= '0';
567         e_player_1_state      <= '0';
568
569         d_position_0        <= (others => '0');
570         d_position_1        <= (others => '0');
571         d_wallshape_0       <= (others => '0');
572         d_wallshape_1       <= (others => '0');
573         d_read_memory_0       <= (others => '0');
574         d_read_memory_1       <= (others => '0');
575         d_next_position_0     <= (others => '0');
576         d_next_position_1     <= (others => '0');
577         d_direction_0       <= (others => '0');
578         d_direction_1       <= (others => '0');
579         d_player_0_state      <= (others => '0');
580         d_player_1_state      <= (others => '0');
581
582         -- go to the state 'wall_shape' next
583         new_state <= wall_shape;
584
585       when wall_shape =>
586         -- determine how the player went over a cell
587         state_vga           <= "111";
588         write_enable        <= '0';
589         write_memory          <= "00000000";
590         address             <= "0000000000";
591         go_to             <= '0';
592         busy_counter_reset      <= '0';
593         clear_memory        <= '0';
594
595         e_position_0        <= '0';
596         e_position_1        <= '0';
597         e_read_memory_0       <= '0';
598         e_read_memory_1       <= '0';
599         e_direction_0       <= '0';
600         e_direction_1       <= '0';
601         e_next_direction_0      <= '0';
602         e_next_direction_1      <= '0';
603
604         d_position_0        <= (others => '0');
```

```
605         d_position_1         <= (others => '0');
606         d_read_memory_0        <= (others => '0');
607         d_read_memory_1        <= (others => '0');
608         d_direction_0        <= (others => '0');
609         d_direction_1        <= (others => '0');
610         d_next_direction_0       <= (others => '0');
611         d_next_direction_1       <= (others => '0');
612
613         -- determine the wall shape on the cell of player 0
614         -- previous: left, next: left or previous: right, next:
               right --> horizontal
615         if ((direction_0= "01") and (next_direction_0  ="01")) or
               ((direction_0= "11") and (next_direction_0  ="11")) then
616           e_wallshape_0 <= '1';
617           d_wallshape_0 <= "001";
618           e_player_0_state <= '0';
619           d_player_0_state <= (others => '0');
620         -- previous: up, next: up or previous: down, next: down
               --> vertical
621         elsif ((direction_0= "00") and (next_direction_0  ="00"))
               or  ((direction_0= "10") and (next_direction_0  ="10"))
               then
622           e_wallshape_0 <= '1';
623           d_wallshape_0 <= "010";
624           e_player_0_state <= '0';
625           d_player_0_state <= (others => '0');
626         -- previous: up, next: right or previous: left, next: down
               --> corner in the lower right
627         elsif ((direction_0= "00") and (next_direction_0  ="11"))
               or  ((direction_0= "01") and (next_direction_0  ="10"))
               then
628           e_wallshape_0 <= '1';
629           d_wallshape_0 <= "110";
630           e_player_0_state <= '0';
631           d_player_0_state <= (others => '0');
632         -- previous: up, next: left or previous: right, next: down
               --> corner in the lower left
633         elsif ((direction_0= "00") and (next_direction_0  ="01"))
               or  ((direction_0= "11") and (next_direction_0  ="10"))
               then
634           e_wallshape_0 <= '1';
635           d_wallshape_0 <= "101";
636           e_player_0_state <= '0';
637           d_player_0_state <= (others => '0');
638         -- previous: right, next: up or previous: down, next: left
               --> corner in the upper left
639         elsif ((direction_0= "11") and (next_direction_0  ="00"))
```

```
           or  ((direction_0= "10") and (next_direction_0  ="01"))
           then
640          e_wallshape_0 <= '1';
641          d_wallshape_0 <= "100";
642          e_player_0_state <= '0';
643          d_player_0_state <= (others => '0');
644        -- previous: down, next: right or previous: left, next: up
              --> corner in the upper right
645        elsif ((direction_0= "10") and (next_direction_0 ="11"))
           or  ((direction_0= "01") and (next_direction_0  ="00"))
           then
646          e_wallshape_0 <= '1';
647          d_wallshape_0 <= "111";
648          e_player_0_state <= '0';
649          d_player_0_state <= (others => '0');
650        -- the player went back to where it came from and it
              collided with itself in the middle of a cell
651        else
652          e_wallshape_0 <= '0';
653          d_wallshape_0 <= (others => '0');
654          e_player_0_state <= '1';
655          d_player_0_state <= "00";
656        end if;
657
658        -- determine the wall shape on the cell of player 1
659        -- previous: left, next: left or previous: right, next:
              right --> horizontal
660        if ((direction_1= "01") and (next_direction_1 ="01")) or
           ((direction_1= "11") and (next_direction_1 ="11")) then

661          e_wallshape_1 <= '1';
662          d_wallshape_1 <= "001";
663          e_player_1_state <= '0';
664          d_player_1_state <= (others => '0');
665        -- previous: up, next: up or previous: down, next: down
              --> vertical
666        elsif ((direction_1= "00") and (next_direction_1 ="00"))
           or  ((direction_1= "10") and (next_direction_1 ="10"))
           then
667          e_wallshape_1 <= '1';
668          d_wallshape_1 <= "010";
669          e_player_1_state <= '0';
670          d_player_1_state <= (others => '0');
671        -- previous: up, next: right or previous: left, next: down
              --> corner in the lower right
672        elsif ((direction_1= "00") and (next_direction_1 ="11"))
           or  ((direction_1= "01") and (next_direction_1 ="10"))
```

```
673             then
674           e_wallshape_1 <= '1';
675           d_wallshape_1 <= "110";
676           e_player_1_state <= '0';
677           d_player_1_state <= (others => '0');
678         -- previous: up, next: left or previous: right, next: down
              --> corner in the lower left
679         elsif ((direction_1= "00") and (next_direction_1 ="01"))
              or  ((direction_1= "11") and (next_direction_1 ="10"))
              then
680           e_wallshape_1 <= '1';
681           d_wallshape_1 <= "101";
682           e_player_1_state <= '0';
683           d_player_1_state <= (others => '0');
684         -- previous: right, next: up or previous: down, next: left
              --> corner in the upper left
685         elsif ((direction_1= "11") and (next_direction_1 ="00"))
              or  ((direction_1= "10") and (next_direction_1 ="01"))
              then
686           e_wallshape_1 <= '1';
687           d_wallshape_1 <= "100";
688           e_player_1_state <= '0';
689           d_player_1_state <= (others => '0');
690         -- previous: down, next: right or previous: left, next: up
              --> corner in the upper right
691         elsif ((direction_1= "10") and (next_direction_1 ="11"))
              or  ((direction_1= "01") and (next_direction_1 ="00"))
              then
692           e_wallshape_1 <= '1';
693           d_wallshape_1 <= "111";
694           e_player_1_state <= '0';
695           d_player_1_state <= (others => '0');
696         -- the player went back to where it came from and it
              collided with itself in the middle of a cell
697         else
698           e_player_1_state <= '1';
699           d_player_1_state <= "00";
700           e_wallshape_1 <= '0';
701           d_wallshape_1 <= (others => '0');
702         end if;
703
704         -- determine the next position of player 0
705         e_next_position_0 <= '1';
706         if (next_direction_0 = "01") then     -- moves to the
              left, x is decreased with 1
            d_next_position_0(4 downto 0)  <=
               std_logic_vector(to_unsigned(to_integer(unsigned(position_0(4
```

```
            downto 0))) - 1, 5));
707         d_next_position_0(9 downto 5) <= position_0(9 downto 5);
708         d_next_position_0(10)     <= position_0(10);
709       elsif (next_direction_0 = "11") then  -- moves to the
             right, x is increased with 1
710         d_next_position_0(4 downto 0)  <=
            std_logic_vector(to_unsigned(to_integer(unsigned(position_0(4
            downto 0))) + 1, 5));
711         d_next_position_0(9 downto 5) <= position_0(9 downto 5);
712         d_next_position_0(10)     <= position_0(10);
713       elsif (next_direction_0 <= "00") then   -- moves up, y is
             decreased with 1
714         d_next_position_0(4 downto 0) <= position_0(4 downto 0);
715         d_next_position_0(9 downto 5) <=
            std_logic_vector(to_unsigned(to_integer(unsigned(position_0(9
            downto 5))) - 1, 5));
716         d_next_position_0(10)     <= position_0(10);
717       else                  --moves down, y is increased with 1
718         d_next_position_0(4 downto 0) <= position_0(4 downto 0);
719         d_next_position_0(9 downto 5) <=
            std_logic_vector(to_unsigned(to_integer(unsigned(position_0(9
            downto 5))) + 1, 5));
720         d_next_position_0(10)     <= position_0(10);
721       end if;
722
723       -- determine the next position of player 1
724       e_next_position_1 <= '1';
725       if (next_direction_1 = "01") then     -- moves to the
             left, x is decreased with 1
726         d_next_position_1(4 downto 0)  <=
            std_logic_vector(to_unsigned(to_integer(unsigned(position_1(4
            downto 0))) - 1, 5));
727         d_next_position_1(9 downto 5) <= position_1(9 downto 5);
728         d_next_position_1(10)     <= position_1(10);
729       elsif (next_direction_1 = "11") then  -- moves to the
             right, x is increased with 1
730         d_next_position_1(4 downto 0)  <=
            std_logic_vector(to_unsigned(to_integer(unsigned(position_1(4
            downto 0))) + 1, 5));
731         d_next_position_1(9 downto 5) <= position_1(9 downto 5);
732         d_next_position_1(10)     <= position_1(10);
733       elsif (next_direction_1 <= "00") then   -- moves up, y is
             decreased with 1
734         d_next_position_1(4 downto 0) <= position_1(4 downto 0);
735         d_next_position_1(9 downto 5) <=
            std_logic_vector(to_unsigned(to_integer(unsigned(position_1(9
            downto 5))) - 1, 5));
```

```
736           d_next_position_1(10)     <= position_1(10);
737         else                    --moves down, y is increased with 1
738           d_next_position_1(4 downto 0) <= position_1(4 downto 0);
739           d_next_position_1(9 downto 5) <=
                 std_logic_vector(to_unsigned(to_integer(unsigned(position_1(9
                 downto 5))) + 1, 5));
740           d_next_position_1(10)     <= position_1(10);
741         end if;
742
743         -- go the state 'check_border' next
744         new_state <= check_border;
745
746       when check_border =>
747         -- check whether or not a player collided with the border
               of the playing field
748         state_vga           <= "111";
749         write_enable        <= '0';
750         write_memory          <= "00000000";
751         address             <= "0000000000";
752         go_to             <= '0';
753         busy_counter_reset      <= '0';
754         clear_memory        <= '0';
755
756         e_position_0        <= '0';
757         e_position_1        <= '0';
758         e_wallshape_0       <= '0';
759         e_wallshape_1       <= '0';
760         e_read_memory_0       <= '0';
761         e_read_memory_1       <= '0';
762         e_next_position_0     <= '0';
763         e_next_position_1     <= '0';
764         e_direction_0       <= '0';
765         e_direction_1       <= '0';
766         e_next_direction_0      <= '0';
767         e_next_direction_1      <= '0';
768
769         d_position_0        <= (others => '0');
770         d_position_1        <= (others => '0');
771         d_wallshape_0       <= (others => '0');
772         d_wallshape_1       <= (others => '0');
773         d_read_memory_0       <= (others => '0');
774         d_read_memory_1       <= (others => '0');
775         d_next_position_0     <= (others => '0');
776         d_next_position_1     <= (others => '0');
777         d_direction_0       <= (others => '0');
778         d_direction_1       <= (others => '0');
779         d_next_direction_0      <= (others => '0');
```

```
780         d_next_direction_1      <= (others => '0');
781
782         -- check if player 0 collides with a border
783         if (((position_0(4 downto 0) = "00000") and
               (next_direction_0 = "01")) or (next_position_0(4 downto
               0) = "11110")) or (((position_0(9 downto 5) = "00000")
               and (next_direction_0 = "00")) or (next_position_0(9
               downto 5) = "11110"))then
784           e_player_0_state <= '1';
785           d_player_0_state <= "01";
786         else
787           e_player_0_state <= '0';
788           d_player_0_state <= (others => '0');
789         end if;
790
791         -- check if player 1 collides with a border
792         if (((position_1(4 downto 0) = "00000") and
               (next_direction_1 = "01")) or (next_position_1(4 downto
               0) = "11110")) or (((position_1(9 downto 5) = "00000")
               and (next_direction_1 = "00")) or (next_position_1(9
               downto 5) = "11110"))then
793           e_player_1_state <= '1';
794           d_player_1_state <= "01";
795         else
796           e_player_1_state <= '0';
797           d_player_1_state <= (others => '0');
798         end if;
799
800         -- the next state is 'want_to_read_0'
801         new_state <= want_to_read_0;
802
803       when want_to_read_0 =>
804         -- let the memory module know that we want to read
               information from the next position of player 0
805         state_vga           <= "111";
806         write_enable        <= '0';
807         write_memory          <= "00000000";
808         address           <= next_position_0 (9 downto 0);
809         go_to             <= '1';
810         busy_counter_reset      <= '0';
811         clear_memory        <= '0';
812
813         e_position_0        <= '0';
814         e_position_1        <= '0';
815         e_wallshape_0       <= '0';
816         e_wallshape_1       <= '0';
817         e_read_memory_0       <= '0';
```

```
818         e_read_memory_1       <= '0';
819         e_next_position_0     <= '0';
820         e_next_position_1     <= '0';
821         e_direction_0       <= '0';
822         e_direction_1       <= '0';
823         e_next_direction_0      <= '0';
824         e_next_direction_1      <= '0';
825         e_player_0_state      <= '0';
826         e_player_1_state      <= '0';
827 
828         d_position_0        <= (others => '0');
829         d_position_1        <= (others => '0');
830         d_wallshape_0       <= (others => '0');
831         d_wallshape_1       <= (others => '0');
832         d_read_memory_0       <= (others => '0');
833         d_read_memory_1       <= (others => '0');
834         d_next_position_0     <= (others => '0');
835         d_next_position_1     <= (others => '0');
836         d_direction_0       <= (others => '0');
837         d_direction_1       <= (others => '0');
838         d_next_direction_0      <= (others => '0');
839         d_next_direction_1      <= (others => '0');
840         d_player_0_state      <= (others => '0');
841         d_player_1_state      <= (others => '0');
842 
843         -- the next state is 'read_memory_player_0'
844         new_state <= read_memory_player_0;
845 
846       when read_memory_player_0 =>
847         -- read the data from the address of the next position of
              player 0
848         state_vga           <= "111";
849         write_enable        <= '0';
850         write_memory          <= "00000000";
851         address           <= next_position_0(9 downto 0);
852         go_to             <= '0';
853         busy_counter_reset      <= '0';
854         clear_memory        <= '0';
855 
856         e_position_0        <= '0';
857         e_position_1        <= '0';
858         e_wallshape_0       <= '0';
859         e_wallshape_1       <= '0';
860         e_read_memory_0       <= '0';
861         e_read_memory_1       <= '0';
862         e_next_position_0     <= '0';
863         e_next_position_1     <= '0';
```

```
        e_direction_0       <= '0';
        e_direction_1       <= '0';
        e_next_direction_0      <= '0';
        e_next_direction_1      <= '0';
        e_player_1_state      <= '0';

        d_position_0        <= (others => '0');
        d_position_1        <= (others => '0');
        d_wallshape_0       <= (others => '0');
        d_wallshape_1       <= (others => '0');
        d_read_memory_0       <= (others => '0');
        d_read_memory_1       <= (others => '0');
        d_next_position_0     <= (others => '0');
        d_next_position_1     <= (others => '0');
        d_direction_0       <= (others => '0');
        d_direction_1       <= (others => '0');
        d_next_direction_0      <= (others => '0');
        d_next_direction_1      <= (others => '0');
        d_player_1_state      <= (others => '0');

        -- wait till the memory module is done with processing the
           information to go to the next state: 'want_to_read_1'
        if (memory_ready = '1') then
          -- when there is already data on the next position of
             player 0, player 0 collides against wall
          if (read_memory = "00000000") then
            e_player_0_state <= '0';
            d_player_0_state <= (others => '0');
          else
            e_player_0_state <= '1';
            d_player_0_state <= "00";
          end if;
          new_state <= want_to_read_1;
        else
          e_player_0_state <= '0';
          d_player_0_state <= (others => '0');
          new_state <= read_memory_player_0;
        end if;

      when want_to_read_1 =>
        -- let the memory module know information needs to be read
           from the next position of player 1
        state_vga           <= "111";
        write_enable        <= '0';
        write_memory          <= "00000000";
        address           <= next_position_1(9 downto 0);
        go_to             <= '1';
```

```
908         busy_counter_reset      <= '0';
909         clear_memory        <= '0';
910 
911         e_position_0        <= '0';
912         e_position_1        <= '0';
913         e_wallshape_0       <= '0';
914         e_wallshape_1       <= '0';
915         e_read_memory_0       <= '0';
916         e_read_memory_1       <= '0';
917         e_next_position_0     <= '0';
918         e_next_position_1     <= '0';
919         e_direction_0       <= '0';
920         e_direction_1       <= '0';
921         e_next_direction_0      <= '0';
922         e_next_direction_1      <= '0';
923         e_player_0_state      <= '0';
924         e_player_1_state      <= '0';
925 
926         d_position_0        <= (others => '0');
927         d_position_1        <= (others => '0');
928         d_wallshape_0       <= (others => '0');
929         d_wallshape_1       <= (others => '0');
930         d_read_memory_0       <= (others => '0');
931         d_read_memory_1       <= (others => '0');
932         d_next_position_0     <= (others => '0');
933         d_next_position_1     <= (others => '0');
934         d_direction_0       <= (others => '0');
935         d_direction_1       <= (others => '0');
936         d_next_direction_0      <= (others => '0');
937         d_next_direction_1      <= (others => '0');
938         d_player_0_state      <= (others => '0');
939         d_player_1_state      <= (others => '0');
940 
941         -- the next state is 'read_memory_player_1'
942         new_state <= read_memory_player_1;
943 
944       when read_memory_player_1 =>
945         -- read the data from the address of the next position of
              player 1
946         state_vga           <= "111";
947         write_enable        <= '0';
948         write_memory          <= "00000000";
949         address           <= next_position_1(9 downto 0);
950         go_to             <= '0';
951         busy_counter_reset      <= '0';
952         clear_memory        <= '0';
953 
```

```
954         e_position_0        <= '0';
955         e_position_1        <= '0';
956         e_wallshape_0       <= '0';
957         e_wallshape_1       <= '0';
958         e_read_memory_0       <= '0';
959         e_read_memory_1       <= '1';
960         e_next_position_0     <= '0';
961         e_next_position_1     <= '0';
962         e_direction_0       <= '0';
963         e_direction_1       <= '0';
964         e_next_direction_0      <= '0';
965         e_next_direction_1      <= '0';
966         e_player_0_state      <= '0';
967
968         d_position_0        <= (others => '0');
969         d_position_1        <= (others => '0');
970         d_wallshape_0       <= (others => '0');
971         d_wallshape_1       <= (others => '0');
972         d_read_memory_0       <= (others => '0');
973         d_read_memory_1       <= (others => '0');
974         d_next_position_0     <= (others => '0');
975         d_next_position_1     <= (others => '0');
976         d_direction_0       <= (others => '0');
977         d_direction_1       <= (others => '0');
978         d_next_direction_0      <= (others => '0');
979         d_next_direction_1      <= (others => '0');
980         d_player_0_state      <= (others => '0');
981
982         -- wait till the memory module is done with processing the
               information to go to the next state: 'check_collision'
983         if (memory_ready = '1') then
984           -- when there is already data on the next position of
                 player 1, player 1 collides against wall
985           if (read_memory = "00000000") then
986             e_player_1_state <= '0';
987             d_player_1_state <= (others => '0');
988           else
989             e_player_1_state <= '1';
990             d_player_1_state <= "00";
991           end if;
992           new_state <= check_collision;
993         else
994           e_player_1_state <= '0';
995           d_player_1_state <= (others => '0');
996           new_state <= read_memory_player_1;
997         end if;
998
```

```
999       when check_collision =>
1000        -- check whether or not players collide with eachother
1001        state_vga           <= "111";
1002        write_enable        <= '0';
1003        write_memory          <= "00000000";
1004        address             <= "0000000000";
1005        go_to             <= '0';
1006        busy_counter_reset      <= '0';
1007        clear_memory        <= '0';
1008
1009        e_position_0        <= '0';
1010        e_position_1        <= '0';
1011        e_wallshape_0       <= '0';
1012        e_wallshape_1       <= '0';
1013        e_read_memory_0       <= '0';
1014        e_read_memory_1       <= '0';
1015        e_next_position_0     <= '0';
1016        e_next_position_1     <= '0';
1017        e_direction_0       <= '0';
1018        e_direction_1       <= '0';
1019        e_next_direction_0      <= '0';
1020        e_next_direction_1      <= '0';
1021
1022        d_position_0        <= (others => '0');
1023        d_position_1        <= (others => '0');
1024        d_wallshape_0       <= (others => '0');
1025        d_wallshape_1       <= (others => '0');
1026        d_read_memory_0       <= (others => '0');
1027        d_read_memory_1       <= (others => '0');
1028        d_next_position_0     <= (others => '0');
1029        d_next_position_1     <= (others => '0');
1030        d_direction_0       <= (others => '0');
1031        d_direction_1       <= (others => '0');
1032        d_next_direction_0      <= (others => '0');
1033        d_next_direction_1      <= (others => '0');
1034
1035        -- when a player collides change its player state
              accordingly
1036        if (next_position_0 = next_position_1) then -- collide at
              eachother at middle of square
1037          e_player_0_state <= '1';
1038          e_player_1_state <= '1';
1039          d_player_0_state <= "00";
1040          d_player_1_state <= "00";
1041        elsif (position_0 = next_position_1) and (position_1 =
              next_position_0) then -- collide at eachother at border
1042          e_player_0_state <= '1';
```

```
1043            e_player_1_state <= '1';
1044            d_player_0_state <= "01";
1045            d_player_1_state <= "01";
1046          elsif (position_0 = next_position_1) then -- player 1
                 collides at the wall of player 0 made the previous time
1047            e_player_0_state <= '0';
1048            e_player_1_state <= '1';
1049            d_player_0_state <= (others => '0');
1050            d_player_1_state <= "00";
1051          elsif (position_1 = next_position_0) then -- player 0
                 collides at the wall of player 1 made the previous time
1052            e_player_0_state <= '1';
1053            e_player_1_state <= '0';
1054            d_player_0_state <= "00";
1055            d_player_1_state <= (others => '0');
1056          else                      -- otherwise do not change the
                 state of the player
1057            e_player_0_state <= '0';
1058            e_player_1_state <= '0';
1059            d_player_0_state <= (others => '0');
1060            d_player_1_state <= (others => '0');
1061          end if;
1062
1063          -- go to 'want_to_write_0' state next
1064          new_state<= want_to_write_0;
1065
1066        when want_to_write_0 =>
1067          -- check if writing is needed for player 0 and let the
                 memory know
1068          state_vga           <= "111";
1069
1070          busy_counter_reset      <= '0';
1071          clear_memory        <= '0';
1072
1073          e_position_0        <= '0';
1074          e_position_1        <= '0';
1075          e_wallshape_0       <= '0';
1076          e_wallshape_1       <= '0';
1077          e_read_memory_0       <= '0';
1078          e_read_memory_1       <= '0';
1079          e_next_position_0     <= '0';
1080          e_next_position_1     <= '0';
1081          e_direction_0       <= '0';
1082          e_direction_1       <= '0';
1083          e_next_direction_0      <= '0';
1084          e_next_direction_1      <= '0';
1085          e_player_0_state      <= '0';
```

```
1086          e_player_1_state      <= '0';
1087
1088          d_position_0        <= (others => '0');
1089          d_position_1        <= (others => '0');
1090          d_wallshape_0       <= (others => '0');
1091          d_wallshape_1       <= (others => '0');
1092          d_read_memory_0       <= (others => '0');
1093          d_read_memory_1       <= (others => '0');
1094          d_next_position_0     <= (others => '0');
1095          d_next_position_1     <= (others => '0');
1096          d_direction_0       <= (others => '0');
1097          d_direction_1       <= (others => '0');
1098          d_next_direction_0      <= (others => '0');
1099          d_next_direction_1      <= (others => '0');
1100          d_player_0_state      <= (others => '0');
1101          d_player_1_state      <= (others => '0');
1102
1103          -- when player 0 collided on border, there shall be no
                writing and the next state is 'want_to_write_1'
1104          if (player_0_state = "01") then
1105            write_enable        <= '0';
1106            write_memory        <= "00000000";
1107            address           <= "0000000000";
1108            go_to             <= '0';
1109            new_state           <= want_to_write_1;
1110          -- if player 0 did not collide the next state will be
                'write_memory_player_0'
1111          else
1112            write_enable        <= '1';
1113            write_memory(7 downto 3)  <= "00000";
1114            write_memory(2 downto 0)  <= wallshape_0;
1115            address           <= position_0(9 downto 0);
1116            go_to             <= '1';
1117            new_state           <= write_memory_player_0;
1118          end if;
1119
1120        when write_memory_player_0 =>
1121          -- send to the memory module the wall shape of player 0 on
                the address of its position
1122          state_vga           <= "111";
1123          write_enable        <= '1';
1124          write_memory(7 downto 3)  <= "00000";
1125          write_memory(2 downto 0)  <= wallshape_0;
1126          address           <= position_0(9 downto 0);
1127          go_to             <= '0';
1128          busy_counter_reset      <= '0';
1129          clear_memory        <= '0';
```

```
1130
1131        e_position_0        <= '0';
1132        e_position_1        <= '0';
1133        e_wallshape_0       <= '0';
1134        e_wallshape_1       <= '0';
1135        e_read_memory_0       <= '0';
1136        e_read_memory_1       <= '0';
1137        e_next_position_0     <= '0';
1138        e_next_position_1     <= '0';
1139        e_direction_0       <= '0';
1140        e_direction_1       <= '0';
1141        e_next_direction_0      <= '0';
1142        e_next_direction_1      <= '0';
1143        e_player_0_state      <= '0';
1144        e_player_1_state      <= '0';
1145
1146        d_position_0        <= (others => '0');
1147        d_position_1        <= (others => '0');
1148        d_wallshape_0       <= (others => '0');
1149        d_wallshape_1       <= (others => '0');
1150        d_read_memory_0       <= (others => '0');
1151        d_read_memory_1       <= (others => '0');
1152        d_next_position_0     <= (others => '0');
1153        d_next_position_1     <= (others => '0');
1154        d_direction_0       <= (others => '0');
1155        d_direction_1       <= (others => '0');
1156        d_next_direction_0      <= (others => '0');
1157        d_next_direction_1      <= (others => '0');
1158        d_player_0_state      <= (others => '0');
1159        d_player_1_state      <= (others => '0');
1160
1161        -- wait until the memory is ready to go to the next state
               'want_to_write_1'
1162        if (memory_ready = '1') then
1163          new_state <= want_to_write_1;
1164        else
1165          new_state <= write_memory_player_0;
1166        end if;
1167
1168      when want_to_write_1 =>
1169        -- check if writing is needed for player 0 and let the
               memory know
1170        state_vga           <= "111";
1171
1172        busy_counter_reset      <= '0';
1173        clear_memory        <= '0';
1174
```

```
1175        e_position_0         <= '0';
1176        e_position_1         <= '0';
1177        e_wallshape_0        <= '0';
1178        e_wallshape_1        <= '0';
1179        e_read_memory_0        <= '0';
1180        e_read_memory_1        <= '0';
1181        e_next_position_0      <= '0';
1182        e_next_position_1      <= '0';
1183        e_direction_0        <= '0';
1184        e_direction_1        <= '0';
1185        e_next_direction_0       <= '0';
1186        e_next_direction_1       <= '0';
1187        e_player_0_state       <= '0';
1188        e_player_1_state       <= '0';
1189
1190        d_position_0         <= (others => '0');
1191        d_position_1         <= (others => '0');
1192        d_wallshape_0        <= (others => '0');
1193        d_wallshape_1        <= (others => '0');
1194        d_read_memory_0        <= (others => '0');
1195        d_read_memory_1        <= (others => '0');
1196        d_next_position_0      <= (others => '0');
1197        d_next_position_1      <= (others => '0');
1198        d_direction_0        <= (others => '0');
1199        d_direction_1        <= (others => '0');
1200        d_next_direction_0       <= (others => '0');
1201        d_next_direction_1       <= (others => '0');
1202        d_player_0_state       <= (others => '0');
1203        d_player_1_state       <= (others => '0');
1204
1205      -- when player 1 collided on a border, there shall be no
             writing and the next state will be 'change_data'
1206      if (player_1_state = "01") then
1207        write_enable         <= '0';
1208        write_memory         <= "00000000";
1209        address            <= "0000000000";
1210        go_to              <= '0';
1211        new_state            <= change_data;
1212      -- if player 1 did not collide, the next state will be
             'write_memory_player_1'
1213      else
1214        write_enable         <= '1';
1215        write_memory(7 downto 3)   <= "00001";
1216        write_memory(2 downto 0)   <= wallshape_1;
1217        address            <= position_1(9 downto 0);
1218        go_to              <= '1';
1219        new_state            <= write_memory_player_1;
```

```
1220        end if;
1221
1222      when write_memory_player_1 =>
1223        -- send to the memory module the wall shape of player 1 on
             the address of its position
1224        state_vga           <= "111";
1225        write_enable        <= '1';
1226        write_memory(7 downto 3)  <= "00001" ;
1227        write_memory(2 downto 0)  <= wallshape_1;
1228        address           <= position_1(9 downto 0);
1229        go_to             <= '0';
1230        busy_counter_reset      <= '0';
1231        clear_memory        <= '0';
1232
1233        e_position_0        <= '0';
1234        e_position_1        <= '0';
1235        e_wallshape_0       <= '0';
1236        e_wallshape_1       <= '0';
1237        e_read_memory_0       <= '0';
1238        e_read_memory_1       <= '0';
1239        e_next_position_0     <= '0';
1240        e_next_position_1     <= '0';
1241        e_direction_0       <= '0';
1242        e_direction_1       <= '0';
1243        e_next_direction_0      <= '0';
1244        e_next_direction_1      <= '0';
1245        e_player_0_state      <= '0';
1246        e_player_1_state      <= '0';
1247
1248        d_position_0        <= (others => '0');
1249        d_position_1        <= (others => '0');
1250        d_wallshape_0       <= (others => '0');
1251        d_wallshape_1       <= (others => '0');
1252        d_read_memory_0       <= (others => '0');
1253        d_read_memory_1       <= (others => '0');
1254        d_next_position_0     <= (others => '0');
1255        d_next_position_1     <= (others => '0');
1256        d_direction_0       <= (others => '0');
1257        d_direction_1       <= (others => '0');
1258        d_next_direction_0      <= (others => '0');
1259        d_next_direction_1      <= (others => '0');
1260        d_player_0_state      <= (others => '0');
1261        d_player_1_state      <= (others => '0');
1262
1263        -- wait till the memory is finished before going to the
             next state 'change_data'
1264        if (memory_ready = '1') then
```

```
1265            new_state <= change_data;
1266          else
1267            new_state <= write_memory_player_1;
1268          end if;
1269
1270        when change_data =>
1271          -- change the data that is going to the graphics engine
                and update data in the register
1272          state_vga           <= "111";
1273          write_enable        <= '0';
1274          write_memory        <= "00000000";
1275          address           <= "0000000000";
1276          go_to             <= '0';
1277          busy_counter_reset      <= '0';
1278          clear_memory        <= '0';
1279
1280          e_wallshape_0       <= '0';
1281          e_wallshape_1       <= '0';
1282          e_read_memory_0       <= '0';
1283          e_read_memory_1       <= '0';
1284          e_next_position_0     <= '0';
1285          e_next_position_1     <= '0';
1286          e_next_direction_0      <= '0';
1287          e_next_direction_1      <= '0';
1288          e_player_0_state      <= '0';
1289          e_player_1_state      <= '0';
1290
1291          d_wallshape_0       <= (others => '0');
1292          d_wallshape_1       <= (others => '0');
1293          d_read_memory_0       <= (others => '0');
1294          d_read_memory_1       <= (others => '0');
1295          d_next_position_0     <= (others => '0');
1296          d_next_position_1     <= (others => '0');
1297          d_next_direction_0      <= (others => '0');
1298          d_next_direction_1      <= (others => '0');
1299          d_player_0_state      <= (others => '0');
1300          d_player_1_state      <= (others => '0');
1301
1302          e_direction_0 <= '1';
1303          e_direction_1 <= '1';
1304
1305          d_direction_0 <= next_direction_0;
1306          d_direction_1 <= next_direction_1;
1307
1308          -- if player 0 collides against a border do not change its
                position, otherwise do
1309          if (player_0_state = "01") then
```

```
1310            e_position_0 <= '0';
1311            d_position_0 <= (others => '0');
1312          else
1313            e_position_0 <= '1';
1314            d_position_0 <= next_position_0;
1315          end if;
1316
1317          -- if player 1 collides against a border do not change its
                 position, otherwise do
1318          if (player_1_state = "01") then
1319            e_position_1 <= '0';
1320            d_position_1 <= (others => '0');
1321          else
1322            e_position_1 <= '1';
1323            d_position_1 <= next_position_1;
1324          end if;
1325
1326          -- check_who_won is the next state
1327          new_state <= check_who_won;
1328
1329        when check_who_won =>
1330          -- check who won based on the player states
1331          state_vga           <= "111";
1332          write_enable        <= '0';
1333          write_memory          <= "00000000";
1334          address             <= "0000000000";
1335          go_to             <= '0';
1336          busy_counter_reset      <= '0';
1337          clear_memory        <= '0';
1338
1339          e_position_0        <= '0';
1340          e_position_1        <= '0';
1341          e_wallshape_0       <= '0';
1342          e_wallshape_1       <= '0';
1343          e_read_memory_0       <= '0';
1344          e_read_memory_1       <= '0';
1345          e_next_position_0     <= '0';
1346          e_next_position_1     <= '0';
1347          e_direction_0       <= '0';
1348          e_direction_1       <= '0';
1349          e_next_direction_0      <= '0';
1350          e_next_direction_1      <= '0';
1351          e_player_0_state      <= '0';
1352          e_player_1_state      <= '0';
1353
1354          d_position_0        <= (others => '0');
1355          d_position_1        <= (others => '0');
```

```
1356          d_wallshape_0       <= (others => '0');
1357          d_wallshape_1       <= (others => '0');
1358          d_read_memory_0       <= (others => '0');
1359          d_read_memory_1       <= (others => '0');
1360          d_next_position_0     <= (others => '0');
1361          d_next_position_1     <= (others => '0');
1362          d_direction_0       <= (others => '0');
1363          d_direction_1       <= (others => '0');
1364          d_next_direction_0      <= (others => '0');
1365          d_next_direction_1      <= (others => '0');
1366          d_player_0_state      <= (others => '0');
1367          d_player_1_state      <= (others => '0');
1368
1369          -- if both players are still playing, go back to the
                'wait_state'
1370          if ((player_0_state = "11") and (player_1_state = "11"))
                then
1371            new_state <= wait_state;
1372          -- if only player 0 is still playing, player 0 won
1373          elsif (player_0_state = "11") then
1374            new_state <= player_0_won;
1375          -- if only player 1 is still playing, player 1 won
1376          elsif (player_1_state = "11") then
1377            new_state <= player_1_won;
1378          -- when both players collided, nobody won and it is a tie
1379          else
1380            new_state <= tie;
1381          end if;
1382
1383        when player_0_won =>
1384          -- player 0 won and tell that to the graphics engine
1385          state_vga           <= "010";
1386          write_enable        <= '0';
1387          write_memory        <= "00000000";
1388          address           <= "0000000000";
1389          go_to           <= '0';
1390          busy_counter_reset      <= '0';
1391          clear_memory        <= '0';
1392
1393          e_position_0        <= '0';
1394          e_position_1        <= '0';
1395          e_wallshape_0       <= '0';
1396          e_wallshape_1       <= '0';
1397          e_read_memory_0       <= '0';
1398          e_read_memory_1       <= '0';
1399          e_next_position_0     <= '0';
1400          e_next_position_1     <= '0';
```

```
1401        e_direction_0       <= '0';
1402        e_direction_1       <= '0';
1403        e_next_direction_0      <= '0';
1404        e_next_direction_1      <= '0';
1405        e_player_0_state      <= '0';
1406        e_player_1_state      <= '0';
1407
1408        d_position_0        <= (others => '0');
1409        d_position_1        <= (others => '0');
1410        d_wallshape_0       <= (others => '0');
1411        d_wallshape_1       <= (others => '0');
1412        d_read_memory_0       <= (others => '0');
1413        d_read_memory_1       <= (others => '0');
1414        d_next_position_0     <= (others => '0');
1415        d_next_position_1     <= (others => '0');
1416        d_direction_0       <= (others => '0');
1417        d_direction_1       <= (others => '0');
1418        d_next_direction_0      <= (others => '0');
1419        d_next_direction_1      <= (others => '0');
1420        d_player_0_state      <= (others => '0');
1421        d_player_1_state      <= (others => '0');
1422
1423        -- stay in this state
1424        new_state <= player_0_won;
1425
1426      when player_1_won =>
1427        -- player 1 won and tell that to the graphics engine
1428        state_vga           <= "011";
1429        write_enable        <= '0';
1430        write_memory        <= "00000000";
1431        address           <= "0000000000";
1432        go_to           <= '0';
1433        busy_counter_reset      <= '0';
1434        clear_memory        <= '0';
1435
1436        e_position_0        <= '0';
1437        e_position_1        <= '0';
1438        e_wallshape_0       <= '0';
1439        e_wallshape_1       <= '0';
1440        e_read_memory_0       <= '0';
1441        e_read_memory_1       <= '0';
1442        e_next_position_0     <= '0';
1443        e_next_position_1     <= '0';
1444        e_direction_0       <= '0';
1445        e_direction_1       <= '0';
1446        e_next_direction_0      <= '0';
1447        e_next_direction_1      <= '0';
```

```
1448          e_player_0_state      <= '0';
1449          e_player_1_state      <= '0';
1450
1451          d_position_0        <= (others => '0');
1452          d_position_1        <= (others => '0');
1453          d_wallshape_0       <= (others => '0');
1454          d_wallshape_1       <= (others => '0');
1455          d_read_memory_0       <= (others => '0');
1456          d_read_memory_1       <= (others => '0');
1457          d_next_position_0     <= (others => '0');
1458          d_next_position_1     <= (others => '0');
1459          d_direction_0       <= (others => '0');
1460          d_direction_1       <= (others => '0');
1461          d_next_direction_0      <= (others => '0');
1462          d_next_direction_1      <= (others => '0');
1463          d_player_0_state      <= (others => '0');
1464          d_player_1_state      <= (others => '0');
1465
1466          -- stay in this state
1467          new_state <= player_1_won;
1468
1469        when tie =>
1470          -- both players lost and tell that to the graphics engine
1471          state_vga           <= "001";
1472          write_enable        <= '0';
1473          write_memory        <= "00000000";
1474          address           <= "0000000000";
1475          go_to           <= '0';
1476          busy_counter_reset      <= '0';
1477          clear_memory        <= '0';
1478
1479          e_position_0        <= '0';
1480          e_position_1        <= '0';
1481          e_wallshape_0       <= '0';
1482          e_wallshape_1       <= '0';
1483          e_read_memory_0       <= '0';
1484          e_read_memory_1       <= '0';
1485          e_next_position_0     <= '0';
1486          e_next_position_1     <= '0';
1487          e_direction_0       <= '0';
1488          e_direction_1       <= '0';
1489          e_next_direction_0      <= '0';
1490          e_next_direction_1      <= '0';
1491          e_player_0_state      <= '0';
1492          e_player_1_state      <= '0';
1493
1494          d_position_0        <= (others => '0');
```

```
1495          d_position_1         <= (others => '0');
1496          d_wallshape_0        <= (others => '0');
1497          d_wallshape_1        <= (others => '0');
1498          d_read_memory_0        <= (others => '0');
1499          d_read_memory_1        <= (others => '0');
1500          d_next_position_0      <= (others => '0');
1501          d_next_position_1      <= (others => '0');
1502          d_direction_0        <= (others => '0');
1503          d_direction_1        <= (others => '0');
1504          d_next_direction_0       <= (others => '0');
1505          d_next_direction_1       <= (others => '0');
1506          d_player_0_state       <= (others => '0');
1507          d_player_1_state       <= (others => '0');
1508
1509          -- stay in this state
1510          new_state <= tie;
1511      end case;
1512    end process;
1513  end behaviour;
```

VHDL/game_engine–behaviour.vhd