1 Introduction

Breakout[1] is an iconic arcade video game similar to the classic game *pong*. The game begins with rows of bricks at the top of the screen, a ball at the center, and a paddle at the bottom. A ball moves around the screen and bounces off the screen boundaries, the bricks, and the paddle. The player must prevent the ball from touching the bottom of the screen by using the paddle that moves left and right. If the paddle misses the ball on rebound, the game is lost. Using the ball, the player must knock down all the bricks at the top of the screen to win the game.

In this TP, we will implement a simplified version of the game on the Gecko4Education board.

2 System Specifications

The input/output mapping on the Gecko4Education board is shown in Figure 1. The game is shown on the LEDs. A turned-on LED may represent the wall, a brick, the ball, or the paddle. The paddle moves in the bottom row of the LEDs by pressing button SW1 to move it left or button SW5 to move it right. The two buttons are used also to start the game. SW7 is instead used to restart the game and acts also as the general reset. Four DIP switches regulate the difficulty of the game. Each one is connected to a brick and defines if the brick is in the game.

The game is composed of 4 bricks, numbered from 1 to 4, of size 2×1 (#rows × #colums), a ball at the center of size 1×1, and a centered paddle at the bottom of size 3×1. Walls cover the external layer of the LEDs, except for the bottom of the screen which is only partially covered. These elements are arranged as shown in Figure 2.

The player may use the leftmost row of the DIP switches to enter which bricks are active at the beginning of the game. For example, if only the leftmost DIP switch (number 1) is on, only brick number one is active at the beginning of the game. The DIP switches cannot change the configuration of the bricks while playing the game or at the end of it.

The game ends when either:

- The player knocks down all the bricks (when there is at least one brick at the beginning of the game)
- The ball reaches the bottom row of the display without bouncing off the paddle or the walls

[1] `https://en.wikipedia.org/wiki/Breakout_(video_game)`

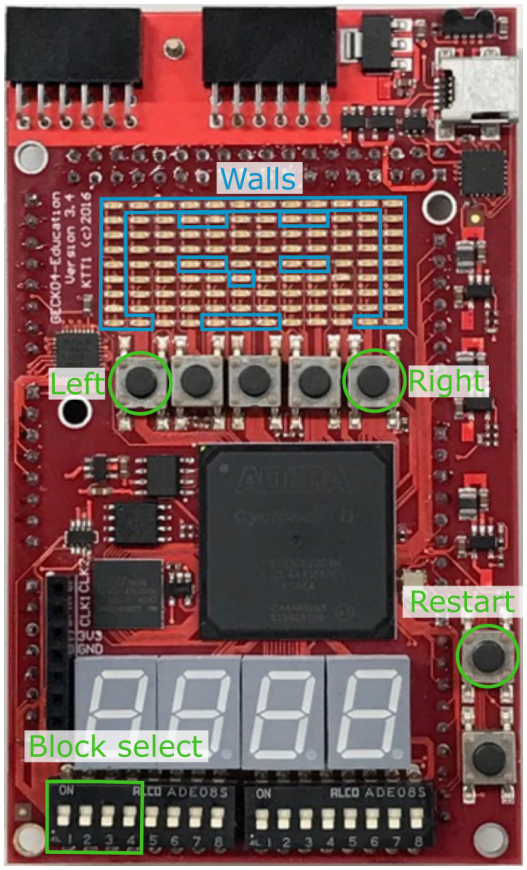

Figure 1: I/O mapping of the Gecko4Education board.

- The reset button is pressed

If all the switches are off, i.e., there are no bricks to break, the game starts normally with the difference that it cannot be won.

2.1 Ball specifications

The ball can freely move around the unoccupied positions. At the beginning of the game, the ball is at the center of the LED matrix and starts by going down. The ball moves by one block at a time in 6 possible directions as shown in Table 1. The ball can never enter a state in which it moves horizontally.

The ball is described by its position and its velocity. The position is defined using $x \in [1, 10]$ and $y \in [0, 7]$ coordinates. For instance, position $(5, 4)$ describes the original position of the ball. The original position of the center of the paddle is instead $(5, 0)$. The ball velocity is represented by (v_x, v_y) that takes the values in Table 1. The ball's initial velocity is $(0, -1)$.

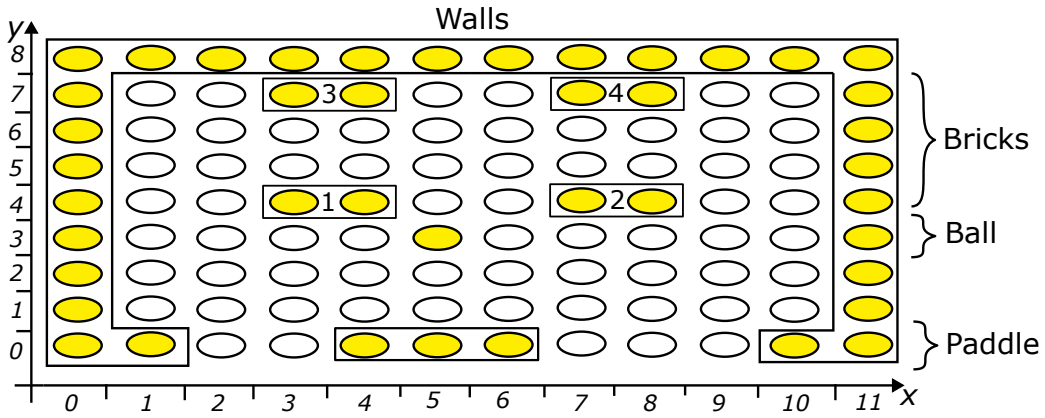

Figure 2: Display of the complete game on LEDs.

Table 1: Possible ball moves.

(v_x, v_y)	LEFT	CENTER	RIGHT
UP	(-1, 1)	(0, 1)	(1, 1)
DOWN	(-1, -1)	(0, -1)	(1, -1)

During the game, the ball hits and bounces off the walls, the paddle, and the bricks. Hits are identified using the next position of the ball (e.g., $p_x + v_x$). If the ball hits a vertical wall (its position is $p_x = 1$ or $p_x = 10$), the velocity is updated as follows:

$$v'_x = -v_x$$
$$v'_y = v_y$$

If the ball hits a horizontal wall, the velocity is updated as follows:

$$v'_x = v_x$$
$$v'_y = -v_y$$

Every second, the ball updates its position as follows:

$$p'_x = p_x + v'_x$$
$$p'_y = p_y + v'_y$$

Note that the new position is directly updated using the new velocity. For example, if the position is $(1, 4)$ with velocity $(-1, 1)$, after bouncing against the vertical left wall the next position will be $(2, 5)$ with velocity $(1, 1)$.

The paddle is composed of 3 segments. When the ball hits one of them, it bounces off with the following velocities shown in Table 5.

Table 2: Ball updates when bouncing off the paddle

		Velocity (v_x, v_y)		
	(v'_x, v'_y)	$(-1, -1)$	$(0, -1)$	$(1, -1)$
Hit position	Left corner	$(-1, 1)$	$(-1, 1)$	$(-1, 1)$
	Segment	$(v_x, 1)$	$(v_x, 1)$	$(v_x, 1)$
	Right corner	$(1, 1)$	$(1, 1)$	$(1, 1)$

For clarity, a left corner hit is shown in Figure 3. A right corner hit works equivalently.

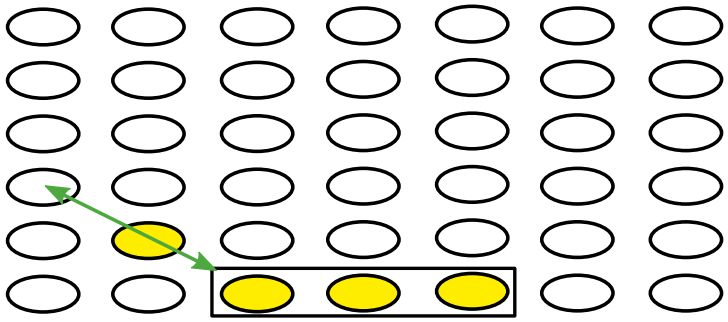

Figure 3: Left corner hit

The bricks behave similarly to the paddle. They are composed of two segments and the behavior is the same as Table 5. A brick is destroyed when the ball hits it in one of the two segments.

Due to the complexity of realizing the ball component by yourself, we provide you with the complete VHDL code. Your design should correctly interface with the ball block.

3 VHDL Module Interfaces

This section describes the structure of the provided VHDL code skeleton and gives you some hints on how the complicated system can be divided into small modules. Each task (module) is tested separately in the auto-grader,

so you can conquer them one by one. Before submitting any code, simulate it with Modelsim as in TP5 and observe the waveform to make sure it works correctly.

3.1 Data Types

We provide custom data types in `utils-package.vhd` to simplify the module interfaces. First, `array_loc_x` and `array_loc_y` are aliases of an array of 4-bit vectors of size 4 which hold the positions of the bricks. Second, `array_loc_valid` is an array of 4 bits which can contain the bricks validity information. Then, `refresh_count` is a constant that controls the refresh rate of the display. Finally, `debounce_count` is another constant to be used for the timer of the debouncer. These data types are summarized in Table 3.

Table 3: Data types for modules' interface.

Name	Content	Meaning
`array_loc_x`	`array(0 to 3) of std_logic_vector(3 downto 0)`	Position of the bricks on the x axis
`array_loc_y`	`array(0 to 3) of std_logic_vector(3 downto 0)`	Position of the bricks on the y axis
`array_loc_valid`	`array(0 to 4) of std_logic`	Validity signals of the bricks
`refresh_count`	integer	Refresh rate of the display
`debounce_count`	integer	Timer setting of the debouncer

The package also contains the function `coord_to_ind`. You may use it to map the (x, y) coordinates of the display to the correct LED in the display.

3.2 Input Interfaces

There are two types of inputs on the board that we use in this TP: the DIP switches and the buttons. Both of them are active low, meaning that when turned on (for switches) or pressed (for buttons), they give a low (logic 0) signal. So, we first negate them in the top-level design (`breakout-rtl.vhd`). The switches are usually stable, so we can use them directly. The buttons, on the other hand, need to be debounced.

There are two problems with the physical buttons that we must address here: 1) the user is free to press the button at any moment they desire, which

may fall into the $[T_{su}, T_h]$ window of some flip-flop, leading it to metastability and 2) due to imperfection of the metal surfaces of the switch contacts and/or tilting of the contact surfaces on the spring, it can happen that while the user is applying force to the button, the current flow gets established and broken multiple times, resulting in multiple transitions of the signal at the output; this phenomenon which is called *switch bouncing* is illustrated in Figure 4.[2] The circuit of Figure 5 solves both of these issues.

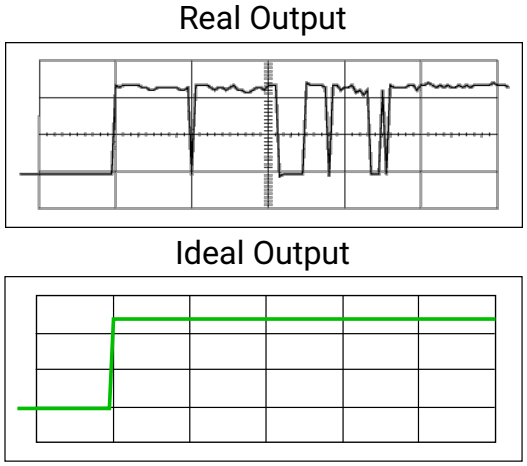

Figure 4: Output of a real mechanical switch illustrating the phenomenon of switch bouncing.

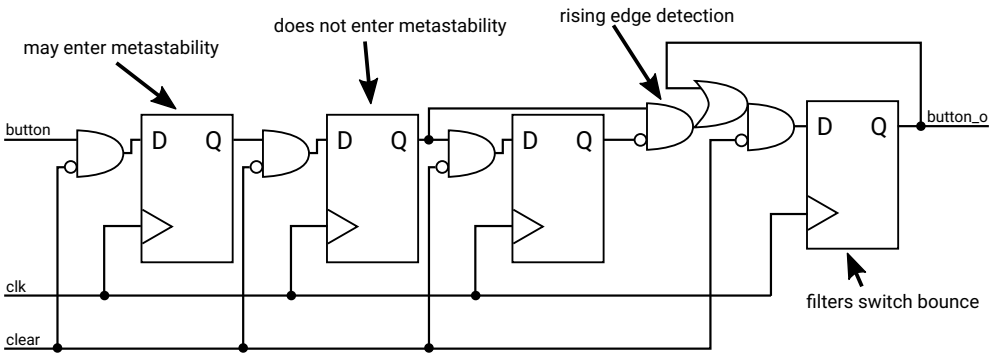

Figure 5: Switch synchronization and debouncing circuit.

Even after synchronization and debouncing, there is yet another problem. The system operates at 12 MHz clock frequency, leaving little time for the user to lift their finger from the button before the current press is detected

[2]Courtesy of Maxim Integrated (`https://www.maximintegrated.com/en/design/technical-documents/app-notes/2/287.html`).

again. Thus, we would like the `clear` input to the debouncer be held 1 for a longer time. For this purpose, we need to integrate a timer to delay the button press detection. The *soft_button* module, that wraps the debouncer and is already implemented for you, converts then the `button_out` signal into a one-cycle strobe (button stays up for only one clock cycle).

Task 1 Press debouncing and delaying (15pts)

- Implement the debouncer as in Figure 5.
- Implement a timer that counts up to the value `count`, a parameter defined in the generic field of the timer entity.
- Integrate the timer inside the debouncer module so that when the *clear* input is asserted by the system, it remains active until the timer is up, thus giving the user enough time to lift the finger before making another press.

3.3 Output Interfaces

The game presents its state to the player using the LED matrix. The LED module *display* in Table 4 interprets the data of the game and outputs the LED signals as a vector of bits. Specifically, the LED driver maps the meaningful game data of different types into a vector of 108 bits connected to the board's LED matrix.

Task 2 Display the game (10pts)

- The LEDs show each active brick according to their status (`brick_loc_valid_arr`).
- Show the walls.
- Show the ball according to its location.
- Show the paddle according to its location.

3.4 Paddle control

The position of the paddle is controlled using the buttons. A specific module named *paddle* keeps track of the position and updates it on buttons' events.

Table 4: Interfaces of the LED driver module.

I/O	Pin name	Type	Description
I	`clk`	`std_logic`	Clock signal
I	`n_reset`	`std_logic`	Active low reset
I	`game_ctl`	`std_logic`	Encoding of the game status (0: stop, 1: play)
I	`ball_loc_x`	`std_logic_vector(3 downto 0)`	Position of the ball on the X-axis
I	`ball_loc_y`	`std_logic_vector(3 downto 0)`	Position of the ball on the Y-axis
I	`paddle_loc_x`	`std_logic_vector(3 downto 0)`	Position of the center of paddle on the X-axis
I	`brick_loc_x_arr`	`array_loc_x`	Array of brick x positions (left segment)
I	`brick_loc_y_arr`	`array_loc_y`	Array of brick y positions
I	`brick_loc_valid_arr`).	`array_loc_valid`	Array that indicates if each brick should be displayed
O	`led_array`	`std_logic_vector (0 to 107)`	The signals connected to the board's LED matrix

The module is directly connected to the LED interface that shows the paddle on the LED matrix. Table 5 describes the paddle interface.

Task 3 Control of the paddle (10pts)

- The paddle moves left and right according to when the control buttons are pressed.
- The paddle stays within the limits of the screen and the walls.
- The paddle is correctly displayed on the screen for each possible position.

3.5 Brick control

The brick module is responsible for the state of a brick. In particular, it manages two tasks. First, it checks if a brick is enabled according to the corresponding DIP switch before the start of the game. Second, it checks if a collision with the ball happened while playing the game. The collision is

Table 5: Interfaces of the paddle module.

I/O	Pin name	Type	Description
I	`clk`	`std_logic`	Clock signal
I	`n_reset`	`std_logic`	Active low reset
I	`game_ctl`	`std_logic)`	Encoding of the game status (0: stop, 1: play)
I	`but_left`	`std_logic`	Button left pressed
I	`but_right`	`std_logic`	Button right pressed
O	`paddle_loc_x`	`std_logic_vector(3 downto 0)`	Position of the center of paddle on the X-axis

computed by combining the position and velocity of the ball. The module should update the validity of the brick accordingly. The update is committed only in specific time frames which are defined by signal `sync`. The interface is described in Table 6.

Task 4 Control of the brick (20pts)

- The brick status is correctly displayed on the LEDs
- If the game is in status *stop* (logical zero) the brick is enabled/disabled according to its corresponding DIP switch.
- If the game is in status *play* (logical one) the brick is disabled after a ball collision and stays disabled until the game is restarted.
- If the game is in status *play* and the ball collides with the brick, the module communicates the status update to the display only when sync is active (active high).

3.6 Game Control

The game control module takes care of the operation of the entire game. The I/O interface of this module is summarized in Table 7. This module is in charge of the status of the game. In particular, it starts the game when a left or right button is pressed, and it stops the game on an end condition. A possible finite state machine is provided for your reference in Figure 6.

Table 6: Interfaces of the brick.

I/O	Pin name	Type	Description
I	`clk`	`std_logic`	Clock signal
I	`n_reset`	`std_logic`	Active low reset
I	`game_ctl`	`std_logic`	Encoding of the game status (0: stop, 1: play)
I	`sync`	`std_logic`	if '1', updates the status to the display module
I	`sw`	`std_logic`	Switch select
I	`ball_loc_x`	`std_logic_vector(3 downto 0)`	Position of the ball on the X-axis
I	`ball_loc_y`	`std_logic_vector(3 downto 0)`	Position of the ball on the Y-axis
I	`ball_vel_x`	`std_logic_vector(3 downto 0)`	Velocity of the ball on the X-axis
I	`ball_vel_y`	`std_logic_vector(3 downto 0)`	Velocity of the ball on the Y-axis
O	`brick_loc_x`	`std_logic_vector(3 downto 0)`	Brick x position of the left segment
O	`brick_loc_y`	`std_logic_vector(3 downto 0)`	Brick y position
O	`brick_loc_valid`	`std_logic`	Indicates if the brick should be displayed

Task 5 Game control (20pts)

This module controls the entire *breakout* game. The I/O interface is summarized in Table 7.

For reference, the top-level module `breakout` instantiates and connects the modules as shown in Figure 7.

4 Simulation and Hardware Testing

To synthesize the bitstream, follow the instructions in TP6 to create a new Quartus project. Choose `breakout/quartus/` as the working directory. The top-level design is `breakout`. Remember to add all the files in `breakout/entities/`, `breakout/architectures/` and `breakout/packages/`

Table 7: Interfaces of the game control.

I/O	Pin name	Type	Description
I	`clk`	`std_logic`	Clock signal
I	`n_reset`	`std_logic`	Active low reset
I	`but_left`	`std_logic`	Button left pressed
I	`but_right`	`std_logic`	Button right pressed
I	`ball_loc_invalid`	`std_logic`	The ball reaches the bottom row of the screen ($y = 0$)
I	`brick_loc_valid.`	`array_loc_valid`	Array that indicates if each brick is valid
O	`game_ctr_out`	`std_logic`	Encoding of the game status (0: stop, 1: play)

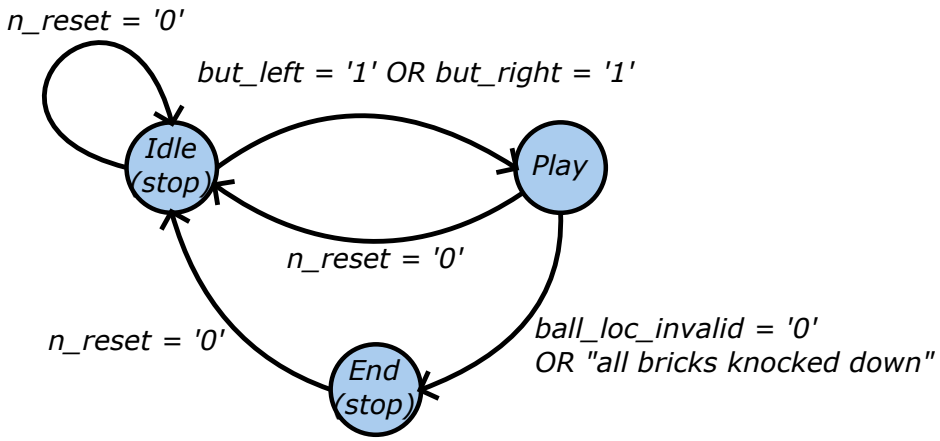

Figure 6: State diagram of the game control.

to the project. Choose the same board as in TP6 (`EP4CE30F23C8`). The needed TCL scripts are already provided, so there is no need to download from the website. Simply run `breakout/quartus/breakout.tcl` and the pin assignments should be done. Click on the compile button to run the synthesis and generate the bitstream file. If the compilation is successful, you will find the SOF file in `breakout/quartus/output_files/`.

Task 6 Hardware demonstration (25 points)

Implement the machine on Gecko4Education and demonstrate that it really works as expected. Set realistic values for the timers so that the

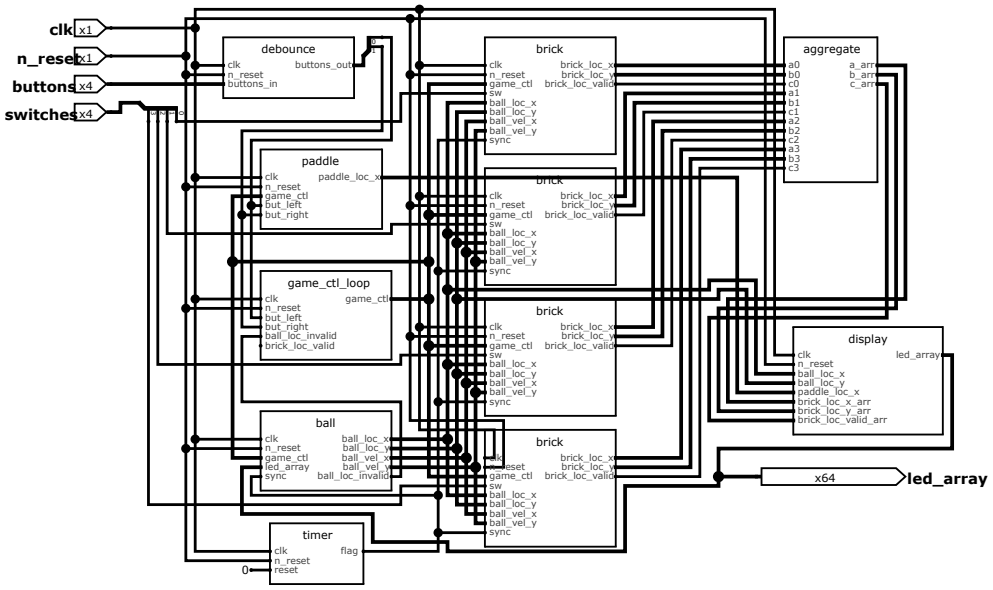

Figure 7: Top level view.

game runs in human time. For instance, the debounce timer clears after 125 milliseconds, and the display refreshes at 4Hz.

5 General Instructions and Rules

Please download the archive *breakout.zip*. It contains the entity files (suffixed with *-entity.vhd*) and architecture files (suffixed with *-rtl.vhd*) for all the modules used in this TP. To make sure that the automated grader recognizes the modules and interfaces correctly, please do not change any of the file, entity, architecture, or port names and write the architecture descriptions in the specified files only.

To submit the assignment, you must rename the provided template folder to "solution". Then, you need to zip it to "solution.zip" which you can submit to the grader at **https://digsys.epfl.ch**. WARNING: Using any other names for the directory or changing the structure of the sub-directories will lead to a build failure and it will count towards your allowed submissions. The total number of uploads allowed is 20, which makes 4 attempts per task, on average. We do not impose a hard limit on the number of attempts per module. The only constraint is that the total does not exceed 20.

Each module is graded separately on Jenkins, thus partial points are possible. The full score after passing all testbenches is 75.

Partial points on hardware demonstration (Task 6) may be given by

matching the result with one of the following cases:

1. The display module passes on Jenkins and the bricks are correctly enabled/disabled using the switches during the starting window of the game. (5 pts)

2. The previous item is fulfilled, the game starts after pressing the left or right button, and the paddle and the ball are shown correctly on the display during the game. (15 pts)

3. All modules pass on Jenkins and the entire system works correctly. (25 pts)