A Pong Game in Assembly Language

Learning Goal: Write a complete program in assembly language and run it on your own processor.

Requirements: nios2sim Simulator (Java 7 or 8), Gecko4Education-EPFL, multicycle Nios II processor.

1 Introduction

During this lab, you will implement a simplified version of the well-known **Pong** game in assembly language. At the end of the lab, you should be able to play the game on the **Gecko4EPFL**.

1.1 About the game

Pong is a game for *two players* inspired by table tennis. It consists of a *table*, *two paddles* and a *ball*. Each player controls a paddle at one end of the table. Using the paddle, a player can hit the ball back to his opponent. When a player misses the ball, his opponent wins one point and the ball is re-engaged and the paddle positions are re-initialized.

The left four push buttons of the Gecko4EPFL control the movement of the paddles (see Figure 1). There are two buttons to control each paddle: one to move it up; the other to move it down. The game is displayed on the LEDs of the Gecko4EPFL, as illustrated in Figure 1.

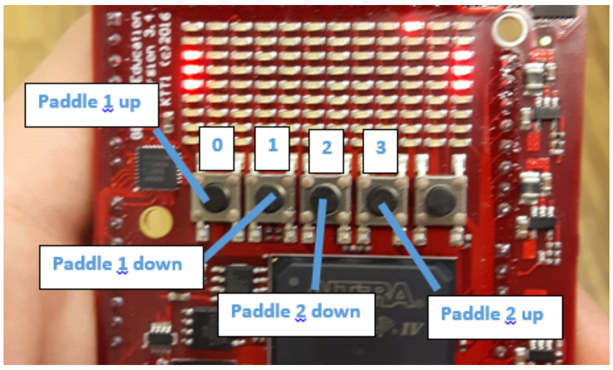


Figure 1: The game **Pong** displayed on the Gecko4EPFL.

The game uses the upper 8 rows of LEDs; to have a convenient mapping from memory to LEDs, we chose not to use the bottom row of LEDs, i.e. the single row that is just above the buttons. The addressing of the remaining 96 LEDs (8 rows × 12 columns), given in Figure 2, is consistent with the mapping of LEDs in the **nios2sim** simulator.

The current state of the game, defined by the position and the velocity of the ball, the position of the paddles and the score, is stored in the RAM. Table 1 shows the precise memory arrangement you **must use** to store the current state of the game. This memory arrangement will be used to test your assembly code.

Table 1: A structure for storing the current state of the game in the RAM. All values are 32-bits long. Ball position is determined using its x- and y-coordinates. Ball velocity is stored as a vector composed of an x- and a y-component. Positive velocity means that the ball is moving in the direction of increasing coordinates. Figure 1 shows that the **origin** of the x-y coordinate system is the **top left** corner of the LED array, that the x-axis **grows rightwards** and that the y-axis **grows downwards**.

`0x1000`	Ball position on **x-axis**
`0x1004`	Ball position on **y-axis**
`0x1008`	Ball velocity along **x-axis**
`0x100C`	Ball velocity along **y-axis**
`0x1010`	Paddle 1 position (left)
`0x1014`	Paddle 2 position (right)
`0x1018`	Score of player 1 (left)
`0x101C`	Score of player 2 (right)

To improve the readability of your code, you can associate symbols to values with the `.equ` statement. The `.equ` statement takes a symbol and a value as arguments. For example, the address structure of internal game state and peripherals can be hard-coded macros as given below. **The addresses represented by macros BALL, PADDLES, SCORES, LEDS, and BUTTONS must match those in Table 1 for correct grading.**

```
.equ    BALL,     0x1000 ; ball state (its position and velocity)
.equ    PADDLES,  0x1010 ; paddles position
.equ    SCORES,   0x1018 ; game scores

.equ    LEDS,     0x2000 ; LED addresses
.equ    BUTTONS,  0x2030 ; Button addresses
```

These symbols can replace any numeric value of your code (like `#define` directive in programming languages). For example, you can also use arithmetic expressions, as shown below:

```
stw     zero, BALL (zero)   ; set ball x-coordinate to 0
stw     zero, BALL+4 (zero) ; set ball y-coordinate to 0
stw     t0, LEDS+8 (zero)   ; set leds[2] to t0 (see Figure 2 for the details)
ldw     t1, SCORE+4 (zero)  ; load the score of player 2 in t1
```

1.2 Formatting rules

In the rest of the assignment, you will be asked to write several procedures in assembly language. If you implement them all correctly, you will be able to play the game using your Gecko4EPFL board. **To enable correct automatic grading of your code, you must follow all the instructions below:**

- surround every procedure with BEGIN and END commented lines as follows:

```
; BEGIN:procedure_name
procedure_name:
    ; your implementation code
    ret
; END:procedure_name
```

Of course, replace the `procedure_name` with the correct name. **The only allowed procedure names are `clear_leds`, `set_pixel`, `move_ball`, `move_paddles`, `draw_paddles`, `hit_test`, and `display_score`. Please pay attention to spelling and spacing of the opening and closing macros.**

- If your procedure makes calls to another, auxiliary procedures, then those auxiliary procedures must also be entirely enclosed:

```
; BEGIN:procedure_name
procedure_name:
    ; your implementation code
    call my_helper_procedure_name
    ; your implementation code
    ret

my_helper_procedure_name:
    ; your implementation code
    ret
; END:procedure_name
```

  Of course, replace the `procedure_name` and `my_helper_procedure_name` with the right names. **The only allowed procedure names are `clear_leds`, `set_pixel`, `move_ball`, `move_paddles`, `draw_paddles`, `hit_test`, and `display_score`.** However, the auxiliary procedures may have whatever name you choose.

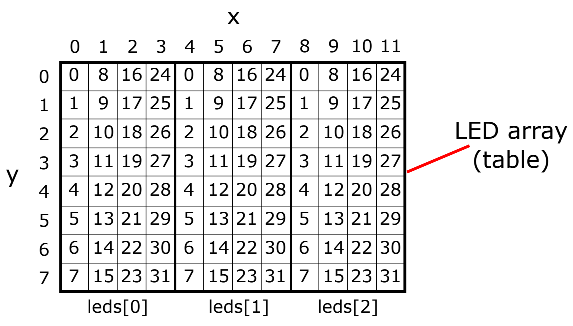
- Have all the procedures inside a **single** `.asm` file. Regardless of that, our grading system will check each procedure individually and separately from the rest of your assembly code.

2 Drawing Using LEDs

Your first exercise is to implement the following two procedures for controlling the LEDs:

- `clear_leds`, which initializes the display by switching off all the LEDs, and
- `set_pixel`, which turns on a specific LED.

The LED array has 96 pixels (LEDs). Figure 2 translates the pixel **x**- and **y**-coordinate into a 32-bit word (leds[0], leds[1], or leds[2]) and a position of the bit inside the word (0 – 31). The words leds[0], leds[1], and leds[2] are stored at the addresses `LEDS`, `LEDS+4`, and `LEDS+8`, respectively. As you are accustomed to, the most significant bit of a byte is referred with the highest index, e.g. 0-th bit is the rightmost and 7-th is the leftmost bit. Bytes are stored in memory in little endian fashion.



y \ x	0	1	2	3	4	5	6	7	8	9	10	11
0	0	8	16	24	0	8	16	24	0	8	16	24
1	1	9	17	25	1	9	17	25	1	9	17	25
2	2	10	18	26	2	10	18	26	2	10	18	26
3	3	11	19	27	3	11	19	27	3	11	19	27
4	4	12	20	28	4	12	20	28	4	12	20	28
5	5	13	21	29	5	13	21	29	5	13	21	29
6	6	14	22	30	6	14	22	30	6	14	22	30
7	7	15	23	31	7	15	23	31	7	15	23	31
	leds[0]				leds[1]				leds[2]			

Figure 2: Translating the LED **x** and **y** coordinates into the corresponding bit in the LED array. For example, **x** = 5 and **y** = 3 correspond to the bit 11 in the word leds[1].

Next two sections describe these two procedures. Section 2.3 describes the steps to follow in this exercise.

2.1 Procedure clear_leds

The `clear_leds` procedure initializes all LEDs to 0 (zero). You should call `clear_leds` before drawing every new position of the ball and/or paddles.

2.1.1 Arguments

- None

2.1.2 Return Values

- None.

2.2 Procedure set_pixel

The `set_pixel` procedure takes two coordinates as arguments and turns on the corresponding pixel on the LED display. When this procedure turns on a pixel, it must keep the state of all the other pixels **unmodified**.

2.2.1 Arguments

- register `a0`: the pixel's **x**-coordinate.
- register `a1`: the pixel's **y**-coordinate.

2.2.2 Return Values

- None.

2.3 Exercise

- Create a new `pong.asm` file.
- Implement the `clear_leds` and `set_pixel` procedures.
- Implement a `main` procedure that calls `clear_leds` and `set_pixel` in the following manner.
  - First, call the `clear_leds` to initialize the display.
  - Then, call the `set_pixel` several times with different parameters to turn on some pixels.
- Simulate your program in **nios2sim**.
- If you want to run this program on your Gecko4EPFL board, follow the instructions in Section 7.

3 Displaying and Controlling the Ball

In this section, you will implement two procedures, `hit_test` and `move_ball`, which control the ball. For the moment, ignore the paddles and make the ball bounce when it reaches the limits of the table. The current state of the ball is represented by its position and its velocity vector:

- The position of the ball is specified using its **x**- and **y**-coordinates, which relate to a single pixel.
- The velocity vector specifies in which direction the ball is moving (up/down, left/right). It has two components: x- and y-direction velocity. For this simple version of the **Pong**, each velocity component can only take one of the following two integer values: -1 or 1. See Table 2.

  Section 3.3 describes the steps to follow in this exercise.

Table 2: Ball movement.

x-velocity	y-velocity	ball direction
1	1	rightwards and downwards
1	-1	rightwards and upwards
-1	1	leftwards and downwards
-1	-1	leftwards and upwards

3.1 Procedure hit_test

The `hit_test` procedure checks whether or not the ball hits the table boundary. If it hits the table boundary, then it must modify the velocity vector to make the ball bounce off the border. This test can be done independently for each ball position.

Let us look to the example in Figure 3:

1. The initial ball position is (x,y) = (3,1) and the velocity vector is (x,y) = (1,-1).
2. Then, the ball reaches the position (x,y) = (4, 0), where `hit_test` detects that the ball touched the upper bound of the table and modifies the ball's velocity vector. The ball shall bounce off with the new velocity vector (x,y) = (1,1).
3. The next the ball position, after it bounced off, is (x,y) = (5,1).

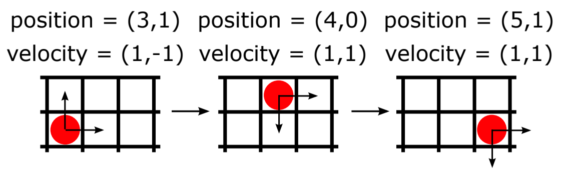


Figure 3: The velocity vector before and after the ball has hit the boundary.

3.1.1 Arguments

- None.

3.1.2 Return Values

- None.

3.2 Procedure move_ball

The `move_ball` procedure moves the ball depending on its velocity vector. It computes the next position of the ball by adding the velocity vector to the current position vector of the ball.

3.2.1 Arguments

- None.

3.2.2 Return Values

- None.

3.3 Exercise

- Implement the `hit_test` and `move_ball` procedures in your `pong.asm` file.
- Modify the `main` procedure. First, it should initialize the ball position and its velocity vector to the values you choose. Then, it should perform the following steps in an infinite loop.
  - Call `clear_leds` to initialize the LEDs.
  - Call `move_ball`.
  - Call `set_pixel` with the ball coordinates as arguments.
  - Call `hit_test`.
- Simulate your program to verify it.

4 Moving and Displaying the Paddles

In this section you will write two procedures, `move_paddles` and `draw_paddles`, which control and display the paddles, respectively. Section 4.3 describes the steps to follow in this exercise.

4.1 Procedure move_paddles

The `move_paddles` procedure reads the state of the push buttons (Figure 1) and moves the paddles accordingly. The paddles move only along the **y** axis, i.e., up or down. Paddles are three-pixels long. Their position is determined only by the **y**-coordinate of the **center** of the paddle. Their **x** coordinates are constants: 0 for the first paddle (on the left) and 11 for the second paddle (on the right).

The y-coordinate of the left paddle center is stored at the address `PADDLES`. The y-coordinate of the right paddle center is stored at the address `PADDLES+4`. See Table 1.

The four push buttons of the Gecko4EPFL are read through the **Buttons** module. This module has two 32-bit words, called `status` and `edgecapture`, described in Table 3. To implement `move_paddles`, you will need to use `edgecapture`.

Table 3: The two words of the **Buttons** module.

Address	Name	31 . . . 4	3 . . . 0
`BUTTONS`	`status`	*Reserved*	State of the Buttons
`BUTTONS+4`	`edgecapture`	*Reserved*	Falling edge detection

The `status` contains the current state of the push buttons: if the bit at the position i is 1, the button i is *currently* released, otherwise (when $i = 0$) the button i is *currently* pressed.

The `edgecapture` contains the information whether the button i ($i = 0, 1, 2, 3$) was pressed. If the button i changed its state from released (1) to pressed (0), i.e. a falling edge was detected, `edgecapture` will have the bit i set. The bit i stays at 1 until it is explicitly cleared to 0 by the program. In the **nios2sim** simulator, you can observe the behavior of this module by opening the **Button** window and clicking on the buttons. In the simulator, the buttons are numbered from 1 to 4.

The `move_paddles` procedure must ensure that the paddles do not leave the table boundaries. Since the paddles are three pixels long and the paddle position is its **center** y-coordinate, the `move_paddles` must ensure the paddle position remains between 1 and 6 (see Figure 2).

4.1.1 Arguments

- None.

4.1.2 Return Values

- None.

4.2 Procedure draw_paddles

The `draw_paddles` procedure draws the paddles on the display. Using the `set_pixel` procedure, it turns on the three pixels that represent the paddle. Once again, the paddle position refers to the y-coordinate of its center.

The `draw_paddles` calls another procedure, thus it has to save some registers on the stack using the **Stack Pointer** register (`sp`). Do not forget to initialize `sp` register at the beginning of your `main` procedure. The `sp` points to the last occupied memory word. Stack grows towards lower addresses. You may initialize `sp` to LEDS, for example.

4.2.1 Arguments

- None.

4.2.2 Return Values

- None.

4.3 Exercise

- Implement the `move_paddles` and `draw_paddles` procedures in your `pong.asm` file.
- Modify the `main` procedure as described below.
  - Initialize the `sp` register.
  - Initialize the position of the paddles.
  - Call `move_paddles` and `draw_paddles` in a loop.
- Simulate your program to verify it.

5 Testing if the Ball Hits a Paddle

In this section, you will modify the `hit_test` procedure to take the paddles into account. The `hit_test` procedure will now return a value in `v0` telling whether a player missed the ball. If the ball hits one of the paddles or/and walls, then this procedure updates the ball's velocity vector.

In order to grasp the intuition behind the ball-paddle interaction, it is best to imagine them as real physical objects. A hit on a paddle occurs if the ball is on the collision path to the paddle. In other words, the ball and the paddle would share the same pixel position if the paddle was not there, so the ball must bounce back. Few examples regarding the expected returns for given current states are given on Table 4 and correspond to the illustrations below. In order to eliminate confusion, visit each and every one of those particular cases and see how paddles are interacting with the ball.

If a paddle misses the ball, the procedure has to return the winner's ID in `v0` (i.e., 1 for the Player 1, and 2 for the Player 2, see Table 1) and keep the position and the velocity of the ball unchanged. Otherwise, `v0` should be 0 for all other cases.

Table 4: Few examples for the behavior of testing hits. Ball position is represented as (x, y) pair. The same applies to ball's velocity. In all the examples, the paddle **y**-center positions are fixed to 2 and 5 for Player 1 and Player 2, respectively.

Figure #	**Iteration #1** Pos. / Vel.	**Iteration #2** Pos. / Vel.	**Iteration #3** Pos. / Vel.
4	$(2,3)$ / $(-1,1)$	$(1,4)$ / $(-1,1)$	—
5	$(2,2)$ / $(-1,1)$	$(1,3)$ / $(-1,1)$	$(2,4)$ / $(1,1)$
6	$(2,1)$ / $(-1,1)$	$(1,2)$ / $(-1,1)$	$(2,3)$ / $(1,1)$
7	$(2,5)$ / $(-1,-1)$	$(1,4)$ / $(-1,-1)$	$(2,5)$ / $(1,1)$
8	$(2,1)$ / $(-1,-1)$	$(1,0)$ / $(-1,-1)$	$(2,1)$ / $(1,1)$

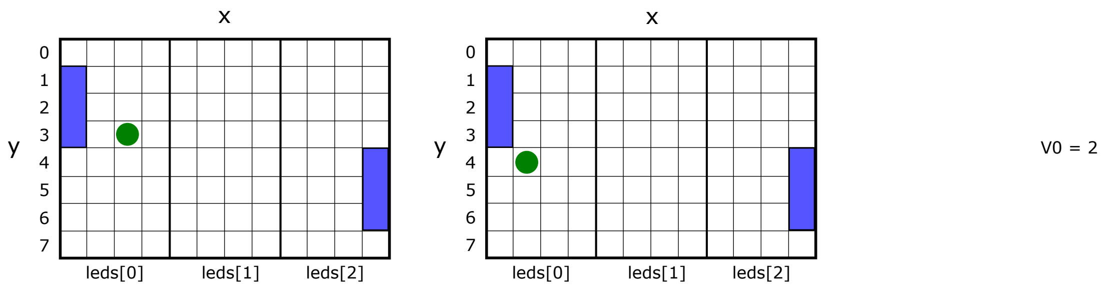


Figure 4: Example of a miss. Left is the ball's initial position. In the middle is its next position. Right is the expected value of the register `v0`.

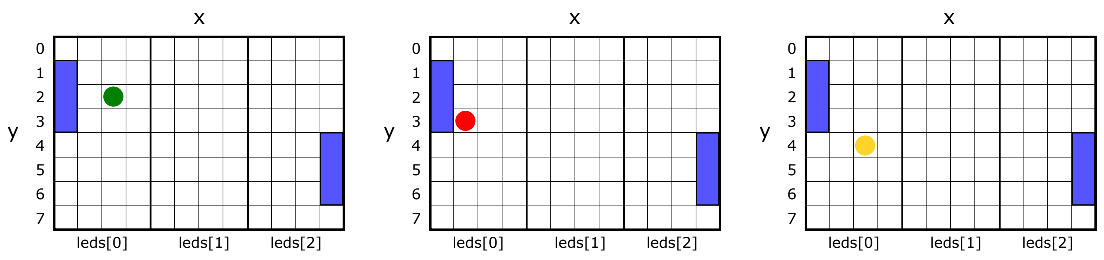


Figure 5: Example of a hit. Left is the ball's initial position. In the middle is its next position resulting in a hit. Right is the expected ball's position after it bounces off the paddle.

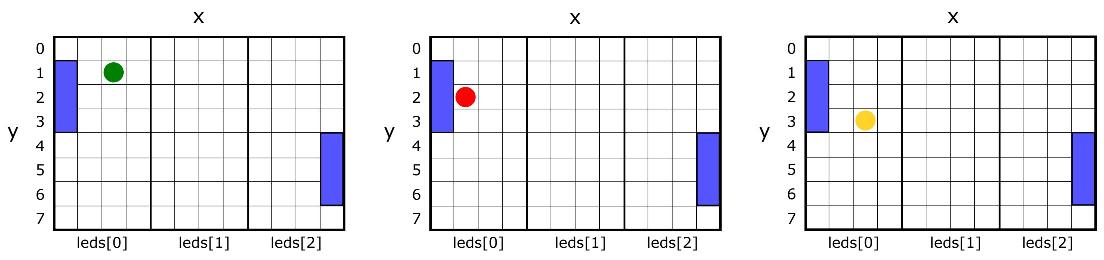


Figure 6: Example of a hit. Left is the ball's initial position. In the middle is its next position resulting in a hit. Right is the expected ball's position after it bounces off the paddle.

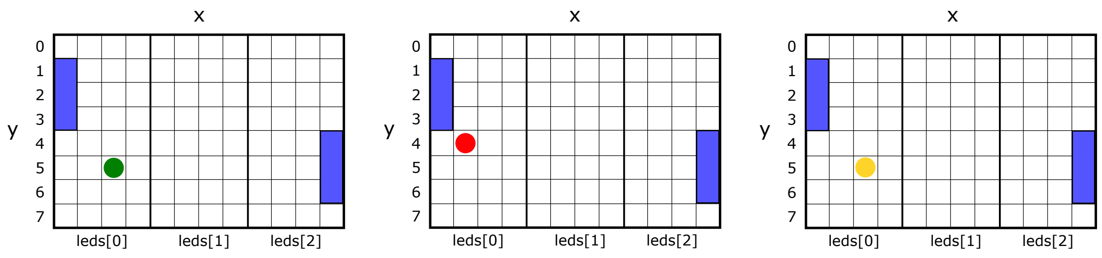


Figure 7: Example of a hit. Left is the ball's initial position. In the middle is its next position resulting in a hit. Right is the expected ball's position after it bounces off the paddle.

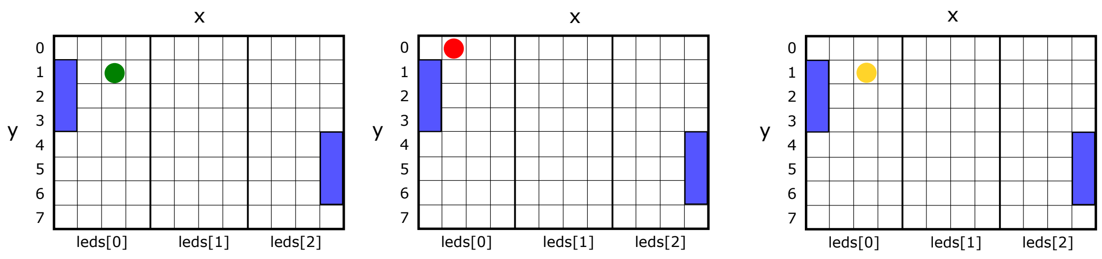


Figure 8: Example of a hit. Left is the ball's initial position. In the middle is its next position resulting in a hit. Right is the expected ball's position after it bounces off the paddle.

5.1 hit_test (final version)

This procedure tests whether or not the ball hits the boundaries of the table or a paddle and modifies its velocity vector accordingly. If there is a winner, `v0` returns the winner's ID **without changing the position and the velocity of the ball**. The velocity of the ball should be modified only if it hits a wall or a paddle.

5.1.1 Arguments

- None.

5.1.2 Return Values

- `v0`: The winner's ID, if there is any; otherwise 0.

5.2 Exercise

- Modify the `hit_test` as described.
- Modify the `main` procedure to make at least the following calls:
  - Call `hit_test`.
  - Call `move_paddles`.
  - Call `move_ball`.
  - Display the game (`clear_leds`, `set_pixel`, `draw_paddles`).

  and to read the value returned by `hit_test` and stop the game when a player misses the ball.
- Simulate your program to verify it.

6 Updating and Displaying the Score

In this section, you will implement a `display_score` procedure to display the score on the LEDs. The following `font_data` section contains a font definition for hexadecimal characters. Each `.word` statement defines the font of the character given in the comments.

```
font_data:
    .word 0x7E427E00 ; 0
    .word 0x407E4400 ; 1
    .word 0x4E4A7A00 ; 2
    .word 0x7E4A4200 ; 3
    .word 0x7E080E00 ; 4
    .word 0x7A4A4E00 ; 5
    .word 0x7A4A7E00 ; 6
    .word 0x7E020600 ; 7
    .word 0x7E4A7E00 ; 8
    .word 0x7E4A4E00 ; 9
    .word 0x7E127E00 ; A
    .word 0x344A7E00 ; B
    .word 0x42423C00 ; C
    .word 0x3C427E00 ; D
    .word 0x424A7E00 ; E
    .word 0x020A7E00 ; F
    .word 0x00181800 ; separator
```

6.1 Procedure display_score

The `display_score` procedure draws the current score on the display. To draw a character on the LEDs, you must load the corresponding word from `font_data` section and store it into the LEDs module. For example, if the score is 3-8, you must perform the following operations.

- Load word 0x7E4A4200 and store it in leds[0] (4 leftmost pixel columns) to draw the digit 3.
- Load word 0x7E4A7E00 and store it in leds[2] (4 rightmost pixel columns) to draw the digit 8.
- Load word 0x00181800 and store it in leds[1] (4 center pixel columns) to draw a separator.

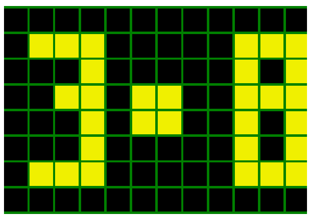


Figure 9: An example score 3 - 8 as shown on LEDs.

6.1.1 Arguments

- None.

6.1.2 Return Values

- None.

6.2 Exercise

- Copy the `font_data` section to the end of your code.
- Implement the `display_score` procedure.
- Modify the `main` procedure to implement the final behavior of the game. You are free to add any other procedure to implement it, provided that you follow all the formatting instructions described in Section 1.2. The `main` procedure should perform the following operations.
  - Initialize the game state.
  - Start a round.
  - Wait for a winner.
  - Update the score in the RAM.
  - Display the score on the LEDs.
  - Reinitialize the position of the ball and the paddles.
  - Start a new round and so on, until one of the player reaches the score of 10 points.
- Simulate your program to verify it.
- Follow the instructions of Section 7 to try the game on your Gecko4EPFL.
- Remember that the automatic grader will look for and test the following procedures: `clear_leds`, `set_pixel`, `move_ball`, `move_paddles`, `draw_paddles`, `hit_test`, and `display_score`. Make sure that you use those exact names in your code and that you enclose the procedures in the appropriate comments (see Section 1.2).
- While implementing the pong game, you might come accross with design choices that are not addressed specifically or left unclear in this document. For those cases, you can safely assume that whichever choice you deem fit necessary will be considered as valid and will not result in loss of points in the final grading. We only use the aformentioned units to test with very clear cases. For example, how the paddles and the ball should be initialized after a round is unspecified, and is not tested. Similarly, you are free to choose what should be the exact speed of the game.

7 Running your Program on the Gecko4EPFL

This section describes the necessary steps to run the program on the **Gecko4EPFL** board.

- You can either use your own working **Nios II** CPU from the **Multicycle Nios II Processor** lab or the **Nios II** CPU provided in the Quartus project `quartus/GECKO.qpf` in the template. If you use your own processor, you just need to replace the `LEDs.vhd` from the previous lab with the one provided in the current template.
- In your `pong.asm` program, add a `wait` procedure to slow down its execution speed.
  - For example, this procedure could initialize and decrement a large counter and return when the counter reaches 0.
  - In your `main`, add a call to your `wait` procedure. You can call it every time after having displayed the new position of the ball and/or paddles, for example.
  - Do not forget to comment the call to the `wait` procedure when going back to the simulation in **nios2sim**, as otherwise the simulation will run too slow.
- In the **nios2sim** simulator, assemble your program (Nios II → Assemble) and export the ROM content (File → Export to Hex File → Choose ROM as the memory module) as [template folder]/quartus/ROM.hex. **Do not modify anything else in the Quartus project folder.**

- Compile the Quartus project.
- Program the **FPGA**.
- Every time you modify your program, do not forget to regenerate the Hex file and to compile the Quartus project again before programming the **FPGA**.

8 Submission

You are expected to submit your complete code as a single .asm file. The automatic grader will look for and test the following procedures: `clear_leds`, `set_pixel`, `move_ball`, `move_paddles`, `draw_paddles`, `hit_test`, and `display_score`. Make sure that you use those exact names in your code and that you enclose these procedures between the appropriate comments (see Section 1.2).

Each of the above listed procedures is tested independently of the rest of your code; everything **around** the tested procedure the grader will replace with the default code. Therefore, you must enclose between appropriate comments all the auxiliary procedures your code calls (see Section 1.2). The only exception is when your procedure calls `set_pixel`: in that case, you do not need to enclose the body of `set_pixel`, because the grader will call the internal implementation of `set_pixel`.

There are two submission links: **pong-preliminary** and **pong-final**:

- You can use the preliminary test as many times as you wish until the deadline. The preliminary tests only checks if the grader found and parsed correctly all the procedures and if your assembly code compiles without errors. The preliminary feedback will refer to these checks only.
- The final test will assess the correctness of the procedures enlisted above by analyzing their effect on memory contents and registers. The final feedback will resemble the following: *Procedure procedure_name passed/failed the test.*

If your code passes all the tests in **pong-final**, you will obtain the maximum score of 80%. For the remaining 20%, you will need to make a successful live demonstration of the game on Gecko4EPFL to the teaching assistants on 29.11.2017.