A Snake Game in Assembly Language

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

Requirements: nios2sim Simulator (Java 10), Gecko4Education-EPFL, multicycle Nios II processor.

1 Introduction

During this lab, you will implement a simplified version of the well-known **snake** game in assembly language. An example of the game can be found here: http://patorjk.com/games/snake/. At the end of the lab, you should be able to play the game on the **Gecko4EPFL**.

1.1 About the game

In this simplified implementation, snake is a single-player game. It consists of a *snake* and *food*. Snake grows as it eats food and dies when it hits a boundary or itself.

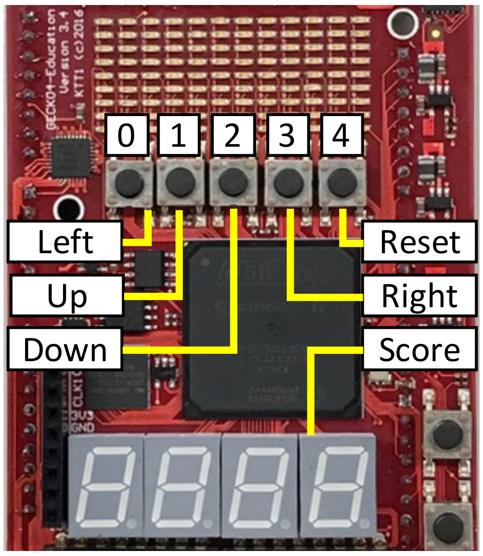


Figure 1: The **Snake** game inputs displayed on the Gecko4EPFL.

The game is to be displayed on the LEDs of the Gecko4EPFL. The initial length of the snake is one. Each time the snake eats food, its length increases by one and the score increases by one. Food is represented by a single, random LED (a single pixel on the LED screen). When the snake's head hits the food, the snake's length increases by setting the position of the food as the new position of the snake's head, while keeping unchanged the position of the snake's tail. (Snake's tail is the last pixel of snake's body away from its head.) After the snake has eaten the food, new food appears at another random location that does not overlap with the body of the snake. The player controls the movement of the head of the snake by using the push buttons on the board.

The left four push buttons of the Gecko4EPFL control the motion of the snake (see Figure 1): left, up, down, and right. The right most push button is used to initialize or restart the game, i.e. clear the score and display the initial state of the game.

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 upper 96 LEDs (8 rows $\times$ 12 columns), given in Figure 2, is consistent with the mapping of LEDs in the **nios2sim** simulator. The top left pixel denotes the origin of the LED coordinate system; x-axis grows rightwards while the y-axis grows downwards. Initially, all LEDs should be switched off.

The current state of the game is defined by the location of the snake (it's entire body), the direction of the snake's movement, and the location of the food. This state of the game is stored in memory and contains 96 32-bit words, corresponding to 96 LED pixels. The address mapping between the two-dimensional LED display and the one-dimensional GSA array is discussed later (Figure 3). The details of populating the GSA are also described later (Section 3 and Section 4).

Table 1 shows the precise memory arrangement you **must use**. In addition to the GSA, the state of the game also includes the location of the snake's head and tail, and the current score. Besides the LED array, the addresses for interfacing with the 7-segment display and the buttons are also fixed. This memory arrangement will be used to test (and grade) your assembly code.

`0x1000`	HEAD_X: Snake head position on **x-axis**
`0x1004`	HEAD_Y: Snake head position on **y-axis**
`0x1008`	TAIL_X: Snake tail position on **x-axis**
`0x100C`	TAIL_Y: Snake tail position on **y-axis**
`0x1010`	SCORE
`0x1014`	GSA: Game State Array
`0x1018`	containing 96 32-bit words
...	...
`0x1198`	SEVEN_SEGS[0]
	SEVEN_SEGS[1]
	SEVEN_SEGS[2]
	SEVEN_SEGS[3]
...	...
`0x2000`	LEDS[0]
	LEDS[1]
	LEDS[2]
...	...
`0x2010`	RANDOM_NUM
...	...
`0x2030`	BUTTONS
`0x2034`	
...	...

Table 1: A structure for storing the current state of the game.

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 HEAD_X, HEAD_Y, TAIL_X, TAIL_Y, SCORE, GSA, LEDS, SEVEN_SEGS, BUTTONS, and RANDOM_NUM must match those in Table 1 for correct grading.**

```
.equ   HEAD_X,       0x1000 ; snake head's position on x-axis
.equ   HEAD_Y,       0x1004 ; snake head's position on y-axis
.equ   TAIL_X,       0x1008 ; snake tail's position on x-axis
.equ   TAIL_Y,       0x100C ; snake tail's position on y-axis
.equ   SCORE,        0x1010 ; score address
.equ   GSA,          0x1014 ; game state array
.equ   LEDS,         0x2000 ; LED addresses
.equ   SEVEN_SEGS,   0x1198 ; 7-segment display addresses
.equ   RANDOM_NUM,   0x2010 ; Random number generator address
.equ   BUTTONS,      0x2030 ; Button addresses
```

These symbols can replace any numeric value of your code (like `#define` directive in programming languages). Example:

```
ldw  t1,    SCORE (zero)     ; load the score 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`, `draw_array`, `get_input`, `move_snake`, `create_food`, `hit_test`, `display_score` and `restart_game`. 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. The auxiliary procedures may have whatever name you choose.

  ```
  ; 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
  ```

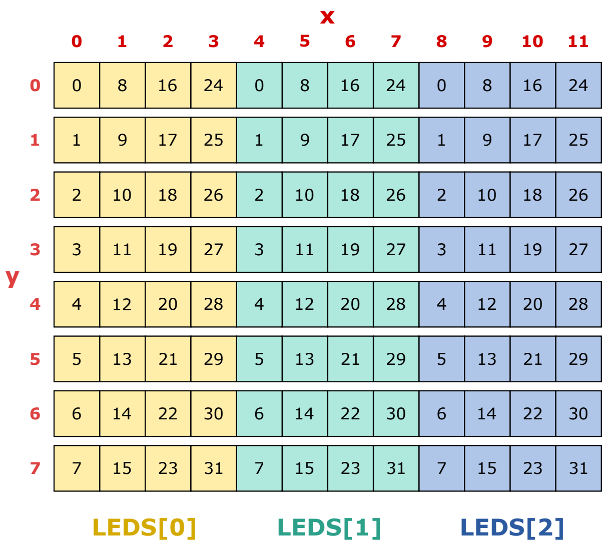
- Have all the procedures inside a **single** `.asm` file. Regardless of that, our grading system will check each procedure individually and separately from the rest of your assembly code.

2 Drawing Using LEDs

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

1. `clear_leds`, which initializes the display by switching off all the LEDs, and
2. `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 and a position of the bit inside that word (0 – 31). The words LEDS[0], LEDS[1], and LEDS[2] are stored consecutively in memory as illustrated in Table 1. As you are accustomed to, the most significant bit of a byte bears 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.



x

y	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 snake and/or food.

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 `snake.asm` file.
- Implement the `clear_leds` and `set_pixel` procedures.
- Implement a `main` procedure that first calls the `clear_leds` to initialize the display and then calls 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 9.

3 Displaying and Controlling the Snake

In this section, you will implement three procedures: `get_input`, which captures the control inputs, `move_snake`, which controls the snake, and `draw_array`, which displays the game. For the moment, ignore any collisions. The current state of the snake is represented by its position and its direction vector:

- The position of the snake is specified in GSA (game state array), using the **x**- and **y**-coordinates of every element of the snake body (every LED pixel occupied by the snake). Game state array will contain the information regarding the food as well; that will be described in Section 4.
- Each member of the game state array (GSA) should have a value between zero and five (inclusive). Zero indicates that the element is not occupied, 1-4 indicate the snake and 5 indicates the food. The different values of the elements occupied by the snake inherently indicate the snake's direction.
- Snake can move in four directions: up/down/left/right. For this simple version of the **Snake**, an element occupied by a snake can only take one of the following four integer values: 1, 2, 3 or 4. See Table 2.

value	**snake direction**
1	leftwards
2	upwards
3	downwards
4	rightwards

Table 2: Snake movement.

Section 3.4 describes the steps to follow in this exercise.

3.1 Procedure get_input

The `get_input` procedure reads the state of the push buttons (Figure 1) and updates the direction of the snake accordingly. The snake moves only along the **x/y** axis, i.e., up/down/left/right.

The push buttons are used to directly modify the direction vector of the head of the snake. To easily move the snake, the position of the snake's head and tail should be tracked separately. The x-coordinate of the head is stored at the address `HEAD_X`. The y-coordinate of the head is stored at the address `HEAD_Y`. Similarly, the x-coordinate of the tail is stored at the address `TAIL_X`. The y-coordinate of the tail is stored at the address `TAIL_Y`. See Table 1.

The five 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 `get_input`, you will need to use `edgecapture`.

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

Address	Name	31 . . . 5	4 . . . 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, 4) was pressed. If the button i changed its state from released to pressed, 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 by your program. Mind that when you attempt to write something in `edgecapture`, regardless of the value **the entire** `edgecapture` **will be cleared**; there is no possibility to clear its individual bits.

In the **nios2sim** simulator, you can observe the behavior of buttons module by opening the **Button** window and clicking on the buttons. In the simulator, the buttons are numbered from 0 to 4.

Once a player presses a push button, your game should update the direction vector of the snake's head in the GSA. However, there is one exception to be considered. If the player requests changing the snake's head direction to the **opposite** of the current snake's head direction, e.g. from left to right or from down to up, then the player's command should be ignored and the direction of the snake's head should not be updated.

3.1.1 Arguments

- None.

3.1.2 Return Values

- None.

3.2 Procedure move_snake

Once you have the new direction vector of snake's head, you can calculate the new position of the snake. The new head element can be easily determined given the current position of the head and its direction vector. The direction vector of the new head should be same as the direction vector of the old head. Note, however, the direction of the new head is subject to change if a new button press is detected afterwards. After creating the new head, the HEAD_X and HEAD_Y values must also be updated.

As snake's body moves together as a whole, the only change other than snake's head is its tail. If the length of the snake stays the same, then you need to remove its tail element. Using the tail coordinates, you can find the tail direction vector in the GSA, which can help determining the next tail element. Then, you clear the old tail element and update the tail coordinates TAIL_X and TAIL_Y to reflect the position of the new tail element.

In the later part of the game development, move_snake will also take input from collision detection. If the snake's head collides with the food item, the snake will eat the food; the snake's length will increase, but the tail will remain at the same position.

3.2.1 Arguments

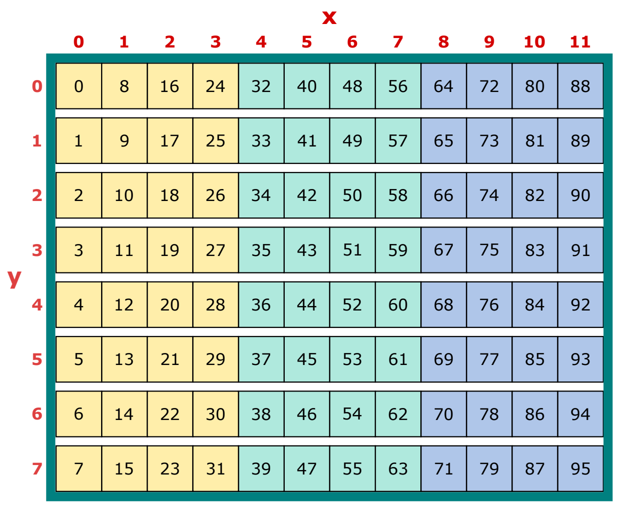
- register a0 = 1 if the snake's head collides with the food, else a0 = 0.

3.2.2 Return Values

- None.

3.3 Procedure draw_array

The `draw_array` procedure draws the game (snake and the food) by reading the contents of the GSA. The game should be redrawn on the screen at regular intervals to depict the movement of the snake. This can be done with a wait procedure (for details on how a wait procedure could be implemented, please see Section 9).



y \ x	0	1	2	3	4	5	6	7	8	9	10	11
0	0	8	16	24	32	40	48	56	64	72	80	88
1	1	9	17	25	33	41	49	57	65	73	81	89
2	2	10	18	26	34	42	50	58	66	74	82	90
3	3	11	19	27	35	43	51	59	67	75	83	91
4	4	12	20	28	36	44	52	60	68	76	84	92
5	5	13	21	29	37	45	53	61	69	77	85	93
6	6	14	22	30	38	46	54	62	70	78	86	94
7	7	15	23	31	39	47	55	63	71	79	87	95

Figure 3: Mapping of a two-dimensional LED display to the one-dimensional Game State Array (GSA).

In `draw_array`, every non-zero element of the GSA which depicts a snake (or food, as described later in Section 4) results in an LED pixel being lit on the LED screen. For every element of the GSA that corresponds to the snake (or the food), the `set_pixel` procedure can be called to activate the corresponding LED and display the snake (or the food).

Since `set_pixel` is another procedure, you may need to consider saving some registers on the stack. Prior to using stack, remember to initialize the **Stack Pointer** register (`sp`) at the beginning of your `main` procedure. Note that the `sp` points to the last occupied memory word and that the stack grows towards lower addresses. You may initialize `sp` to LEDS, for example.

3.3.1 Arguments

- None.

3.3.2 Return Values

- None.

3.4 Exercise

- Implement the `get_input`, `move_snake`, and `draw_array` procedures in your `snake.asm` file.
- Modify the `main` procedure. First, it should initialize the snake position and its direction vector: snake should be of length 1, should appear at the upper left corner of the screen and move rightwards. Then, it should perform the following steps in an infinite loop:
  - Call `clear_leds`.
  - Call `get_input`.
  - Call `move_snake`.
  - Call `draw_array`.
- Simulate your program to verify it.

4 Creating and Displaying the Food

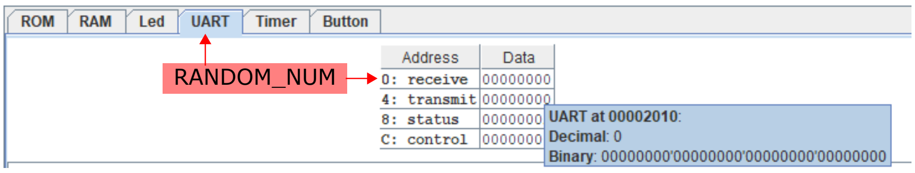
In this section you will write procedure `create_food`, which creates a new food item at a random location on the screen. The food size is always one (a single LED pixel), while its location must not overlap with the snake. You can differentiate between a snake and the food easily: GSA element representing the food has the value 5, while the GSA elements representing the snake have values 1-4. To display the food, `draw_array` can be used.

4.1 Procedure create_food

The `create_food` procedure creates a new food item. To get a random number when playing a game on your FPGA board, you can read the location RANDOM_NUM (Table 1) to which we have mapped a random number generator module designed in VHDL.

The `create_food` first takes **the lowest byte** of the value at RANDOM_NUM and then, if the suggested new food location is within the limits of the two-dimensional LED display and does not overlap with the snake, it creates food. The lowest byte of the value at RANDOM_NUM corresponds to the index of an element in GSA. For example, if the lowest byte value is 0x00 then the food should appear at the location corresponding to the very first element of the GSA.

To simulate random number generation in **nios2sim**, you can write a value of your choice at the location RANDOM_NUM. Inside the **nios2sim** this memory location overlaps with the address space reserved for UART (Figure 4); this poses no issues as your processor does not use UART. Therefore, to access the location RANDOM_NUM in nios2sim, open the UART tab and look for the field `receive`.



Address	Data
0: receive	00000000
4: transmit	00000000
8: status	0000000
C: control	0000000

UART at 00002010:
Decimal: 0
Binary: 00000000'00000000'00000000'00000000

Figure 4: The location where you should write a random number for simulating the game in nios2sim.

4.1.1 Arguments

- None.

4.1.2 Return Values

- None.

4.2 Exercise

- Implement the `create_food` procedure in your `snake.asm` file.
- Modify the `main` procedure as described below.
  - Initialize the `sp` register.
  - Call `clear_leds`.
  - Call `create_food`.
    - Get a random location for the food.
    - Check if the location is valid.
    - If it is not valid, get new random location.
  - Call `draw_array`.
- Simulate your program to verify it.

5 Collision Testing

Now that you have implemented the snake's movement and food generation, you need to detect any collisions between the snake and the surroundings. The snake can collide in three ways: (1) with the screen boundary, (2) with the food item, or (3) with its own body.

Collision with the boundary or the snake's body should **terminate the game**. Once the game is terminated, the contents of the GSA and SCORE must remain unchanged.

You should start by calculating the next position of the snake's head based on the direction vector of the snake's head. However, do not update the snake's head location in GSA as that is the task of `move_snake`.

5.1 Procedure hit_test

This procedure tests whether or not the new element being plotted for the snake's head collides with the screen boundary, the food, or the snake's own body. If there is a collision with the food, the procedure returns 1 indicating that the score needs to be incremented. If there is a collision with the screen boundary or the snake's body, the procedure returns 2 indicating the end of the game. If there is no collision, the procedure returns 0.

5.1.1 Arguments

- None.

5.1.2 Return Values

- `v0`: 1 for score increment, 2 for the game end, and 0 when no collision.

5.2 Exercise

- Write the `hit_test` as described.
- Modify the `main` procedure to make at least the following calls:
  - Call `clear_leds`.
  - Call `get_input`.
  - Call `hit_test`.
  - If `hit_test` returns 1, call `create_food`.
  - If `hit_test` returns 2, terminate the game.
  - If no collision, call `move_snake` and `draw_array`.
- Simulate your program to verify it.

6 Displaying the Game Score

In this section, you will implement a `display_score` procedure to show the game score on the 7-segment displays.

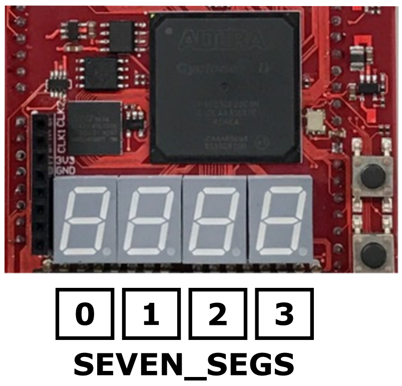


Figure 5: Seven segment displays.

You may use the following `font_data` table to define every digit you may need to display. Each `.word` statement defines a value which, if stored in one 7-segment display, results in it showing the decimal digit written in the comment.

```
font_data:
    .word 0xFC ; 0
    .word 0x60 ; 1
    .word 0xDA ; 2
    .word 0xF2 ; 3
    .word 0x66 ; 4
    .word 0xB6 ; 5
    .word 0xBE ; 6
    .word 0xE0 ; 7
    .word 0xFE ; 8
    .word 0xF6 ; 9
```

6.1 Procedure display_score

The `display_score` procedure draws the current score (in decimal representation) on the display. To draw a digit on the 7-segment display, you can load the corresponding word from `font_data` table and store it into the corresponding 7-segment display module. There are four 7-segment displays on the board, indexed from 0 to three as shown in Figure 5, and memory mapped starting from the address SEVEN_SEGS (Table 1). Since the score can never be higher than 99, the two leftmost 7-segment modules should always show zero.

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
  - Get the inputs for snake movement.
  - Check for collision.
  - If snake's head collided with food, update the score, create new food.
  - Update the position of the snake.

7 Restart the Game

As a final touch to our game, let us include the feature of restarting the game upon pressing the reset button (Figure 1). For that purpose, write the procedure `restart_game`.

7.1 restart_game

This procedure checks for the input from **the reset button** (Figure 1) and, if the reset/restart is detected, (re)initializes the game.

The initial state of the game is defined by the snake of length 1, appearing at the top left corner of the LED screen and moving towards right, while the food is appearing at a random location, and the score is all zeros.

7.1.1 Arguments

- None.

7.1.2 Return Values

- `v0`: 1 if the reset button was pressed and the game state (re)initialized, 0 otherwise.

7.2 Exercise

The `main` procedure should perform the following operations in a loop. Although the user may attempt to restart the game at any point, it is sufficient to test for reset only once per loop iteration.

- Check if reset/restart was requested; if so, initialize the game.
- If the game is finished, loop again.
- Get the inputs for snake movement.
- Check for collision.
- If snake's head collided with food, update the score, create new food.
- Update the position of the snake.

Simulate your program to verify it. Before playing the game on Gecko4EPFL, read Section 8.

8 Playing the Game

Now that you have have implemented all the required core functionality, test if the game runs smoothly end to end. Simulate your program to verify it. Follow the instructions of Section 9 to try the game on your Gecko4EPFL.

When playing the game on Gecko4EPFL, make sure that you add a call to `wait` between the call to `get_input` and the call to `restart_game`, and that `restart_game` is called near the end of every loop iteration. Otherwise it may happen that a press on the reset button is ignored by your program.

While implementing the snake game, you might come across 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 will be considered as valid and will not result in loss of points in the final grading.

9 Running your Program on the Gecko4EPFL

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

- Please use the **Nios II** CPU provided in the Quartus project `quartus/GECKO.qpf` in the template.
- In your `snake.asm`, add a `wait` procedure to slow down the execution speed of the program.
  - For example, the procedure `wait` could initialize a large counter and decrement it in a loop, returning when the counter reaches 0.
  - In your `main` procedure, call the `wait` procedure. For example, you can call it every time after having displayed the new position of the snake and/or food.
  - Remember 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, remember to regenerate the Hex file and to compile the Quartus project again before programming the FPGA.

10 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`, `get_input`, `move_snake`, `draw_array`, `create_food`, `hit_test`, `display_score`, and `restart_game`.Make sure that you follow the formatting instructions detailed in 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: **snake-preliminary** and **snake-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 **snake-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.

You are allowed (and encouraged) to add other features to the game, e.g., increasing the snake's speed after eating food each time. However, you must not submit an enhanced game to the automated grader, as it expects a basic game that conforms to the instructions detailed in this document. Teams that demonstrate the most interesting and complete game will have a chance to win **the ArchOrd Christmas gift**! So, be creative!