Snake Game in Assembly

Learning Goal: Write an assembly program which runs on a Nios II processor.

Requirements: nios2sim Simulator, Gecko4Education-EPFL, multicycle Nios II processor.

1 Introduction

This lab guides you to write a simplified single-player version of the **snake** game in assembly. You can find an implementation of the game at this link, http://patorjk.com/games/snake/. Upon completing the lab, you should be able to play the game on the **Gecko4EPFL**.

You also have at your disposal the **nios2sim** simulator, which has a consistent behavior with the Gecko4EPFL and should help you test your program. You can download it from the moodle page of the course. **Note that the simulator does not support `ldb` and `stb` instructions.**

1.1 About the game

The game consists of a *snake* and *food*. The player can change the direction of the snake's movement to avoid obstacles or reach food. If the snake eats food, then the player's score increases; if the snake hits an obstacle, then the player loses. The goal of the player is to maximize her or his score.

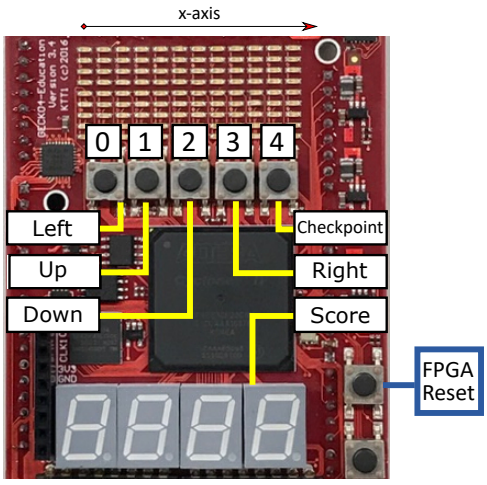

Figure 1: The **Snake** game inputs displayed on the Gecko4EPFL.

We use the upper eight rows of the LED array on Gecko4EPFL as our display. We draw a snake by turning on LEDs in our display which represent the snake. Initially the snake has length one, corresponding to a single LED. We also draw food using an LED in the display.

The snake has a head and a tail. At the beginning of the game, the head and tail correspond to the same LED position. Later, the head moves in some direction. We say the snake *hits* the food if its head's neighbor LED in the direction of the movement is the food. The snake *eats* the food after hitting it: When the snake hits the food, its length increases by one through setting the position of the food as the new position of its head while keeping the position of its tail unchanged. After the snake eats food, new food appears at a random location outside the snake's body. The player can change the movement direction using push buttons labelled 0 to 3 in Figure 1: left, up, down, and right. When the snake dies, the player loses and the game restarts.

At each point in time, the game is described by the snake position, the direction of the snake's movement, the position of the food, and the score. To allow a player to resume from a previous game state, the entire game state (a checkpoint) is saved whenever the player's score becomes a multiple of 10 (i.e., 10, 20, 30, etc.). Only the most recent checkpoint is kept in memory. After the checkpoint is saved, all 7-segments blink to inform the player. To restore the last checkpoint, the player can press button 4 in Figure 1. Here again, all 7-segments blink once the checkpoint is restored, to inform the player.

The state of the game display (LEDs) is described in a so-called game-state array (GSA), which is kept in memory starting from the address `0x1014` (see Table 1). The addressing of the upper 96 LEDs (8 rows $\times$ 12 columns) 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 are switched off. The address mapping between the two-dimensional LED display and the one-dimensional GSA array, as well as the details on filling in the GSA, will be described later.

> 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.

In your assembly, you will call procedures, so the stack pointer register has to be correctly initialized `sp` so that you can use the stack to backup registers (`ra`, `s0`, etc.) in case they are overwritten. For instance, every `call` instruction will change the value of `ra` register. If in procedure `A` you call another procedure `B`, the `ra` register will be updated with the returning address of `B`, making `A` unable to return to its caller. In this case, you need to use the stack to back up the value of `ra` before calling `B`, and restore its value after `B` returns.

Since the stack grows from high address to low address, in the template file, we have initialized its value to the bottom of the RAM, which is the starting address of `LEDS` (`0x2000`).

To improve the readability of your code, you can associate symbols to values with the **`.equ`** statement. **`.equ`** statement takes a symbol and a value as arguments. Example:

```
.equ  SCORE, 0x1010 ; score address
```

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

```
ldw  t1, SCORE (zero)    ; load the score in t1
```

> **Note for grading:** The addresses represented by macros HEAD_X, HEAD_Y, TAIL_X, TAIL_Y, SCORE, GSA, SEVEN_SEGS, CP_VALID, CP_HEAD_X, CP_HEAD_Y, CP_TAIL_X, CP_TAIL_Y, CP_SCORE, CP_GSA, LEDS, RANDOM_NUM, and BUTTONS must match those in Table 1 for correct grading. For convenience, we provide a template file with all the macros already defined.

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]
...	...
0x1200	CP_VALID: Whether the checkpoint is valid
0x1204	CP_HEAD_X: Snake head position on **x-axis**
0x1208	CP_HEAD_Y: Snake head position on **y-axis**
0x120C	CP_TAIL_X: Snake tail position on **x-axis**
0x1210	CP_TAIL_Y: Snake tail position on **y-axis**
0x1214	CP_SCORE
0x1218	CP_GSA: Checkpoint Game State Array
0x121C	containing 96 32-bit words
...	...
...	...
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. CP stands for checkpoint.

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`, `init_game`, `blink_score`, `save_checkpoint`, and `restore_checkpoint`. Please pay attention to spelling and spacing of the opening and closing macros.**

- If your procedure makes calls to auxiliary procedures, then those auxiliary procedures must also be entirely enclosed. You may have auxiliary helper procedures which are *shared* among procedures when developing your code, but when submitting, you need to make sure **each procedure has a copy of the shared auxiliary procedure with proper renaming**. Not the best practice for do-not-repeat-yourself software principle, but it is **necessary for grading**. The auxiliary procedures may have any name except the ones above (clear_leds, etc.).

```
; 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. Our grading system checks each procedure independently: suppose procedure B calls procedure A, if there is a mistake in procedure A, then you can still get full credits for procedure B.

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 bits (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.
The 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

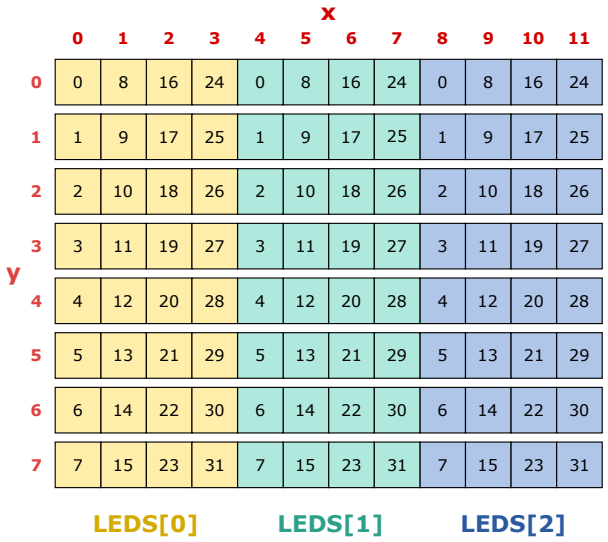

Figure 2: Translating the LED **x** and **y** coordinates into the corresponding bit in the LED array. For example, **x** = 5 and **y** = 3 correspond to the bit 11 in the word LEDS[1].

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 your `snake.asm` file from the template we provide. The file encoding should be UTF-8 (`https://en.wikipedia.org/wiki/UTF-8`). French accents should not be used.
- 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 four procedures: `get_input`, which captures the pressed button, `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 element 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 buttons (Figure 1) and updates snake's head direction accordingly if a direction button is pressed. The five 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.

The purpose of `get_input` is to analyze the `edgecapture`, clear it, and change snake's head direction in GSA if a direction button was pressed. It should check each bit of the `edgecapture` and find the pressed button. If the pressed button turns out to be a direction button, you need to change snake's head direction accordingly. However, according to the rules of the game, no direction change should be made if the current head's moving direction is opposite to the pressed button. For instance, no update should happen if the snake's head is moving towards right and the leftwards button is pressed, even if the length of the body is only one pixel.

When grading, we don't consider the case where multiple direction buttons were pressed (in other words, you are free to choose which button should have the highest priority in such a case). However, if the `checkpoint` button was pressed together with any other button, this procedure should return that only the `checkpoint` button was pressed.

3.1.1 Arguments

- None.

3.1.2 Return Values

- register `v0`: Which button is pressed. The return value is indicated in table 4.

value	Button pressed
0	none
1	left
2	up
3	down
4	right
5	checkpoint

Table 4: Snake movement.

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 time 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).

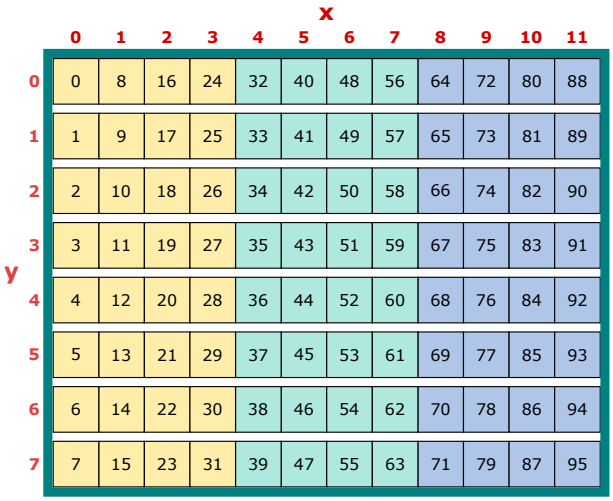

Figure 3: Mapping of a two-dimensional LED display to the one-dimensional Game State Array (GSA).

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 should be called to activate the corresponding LED and display the snake (or the food).

3.3.1 Arguments

- None.

3.3.2 Return Values

- None.

3.4 Exercise

- Implement `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 have length one and 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 valid (meaning that it is within the limits of the two-dimensional LED display and does not overlap with the snake), it creates food. Otherwise, the random number is read again (new number will be available) until a valid food location is obtained.

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`.

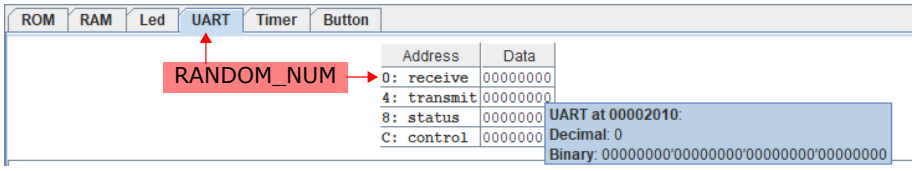

Figure 4: The location where you should write a random number for simulating the game in **nios2sim**.

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.
  - 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 LED 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`.

> **Note:** For simplicity, we assume that the collision test is carried out *before* moving the snake's body. Therefore, if the next position happens to overlap with the snake's tail, collision will be detected instead of allowing the snake to move. Our automated grader will not test this particular case, but you are free (if you wish) to find and implement a solution to this particular corner case.

5.1 Procedure hit_test

This procedure tests whether or not the new element being drawn as 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.

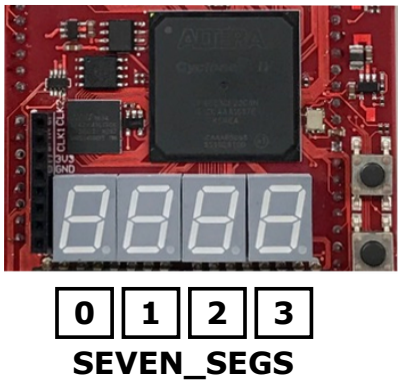

Figure 5: Seven segment displays.

Figure 6 shows `digit_map` table which defines digits. 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.

```
digit_map:
    .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
```

Figure 6: digit_map

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 `digit_map` table in Figure 6 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). We assume that the score can never be higher than 99, so the two leftmost 7-segment modules should always show zero. However, we don't enforce the behavior of this procedure when the score is larger than 100. For example, you can choose to display the complete score, or the score modulo 100.

6.1.1 Arguments

- None.

6.1.2 Return Values

- None.

6.2 Procedure blink_score

The `blink_score` procedure serves for blinking 7-segment display. Blinking means that the 7-segment display periodically shows the score and switches off. In other words, you should clear the 7-segment display, wait (see the procedure `wait` in Section 9.), and call `display_score`. You are free to blink the score arbitrary number of times.

Note for grading: This procedure is not graded automatically; it will be graded by the teaching assistants when they will try to play your game on Gecko4EPFL.

6.2.1 Arguments

- None.

6.2.2 Return Values

- None.

6.3 Exercise

- Copy the content of `digit_map` 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, then update the score and create new food.
  - Update the position of the snake.

7 Initialize the Game

The game state should be initialized when the player loses the game or when the system is reset (by pressing the FPGA reset button in the Figure 1). You should write a procedure called `init_game` to implement this task.

The initial state of the game is defined by the snake of length one, 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.

Please note that the checkpoint described in Section 8 should not be influenced by this procedure.

7.0.1 Arguments

- None.

7.0.2 Return Values

- None.

7.1 Exercise

The `main` procedure should perform the following operations in a loop:

- Initialize the game.
- Get the input for snake movement.
- Check for collision.
- If snake's head collides with food, then update the score and create new food.
- Get the input. If the checkpoint button is pressed, do nothing, otherwise check for collision and move the snake.
- If the game finishes, repeat all previous steps and restart the game.

8 Checkpoint

There are two operations regarding the checkpoint mechanism: save and restore. You're requested to implement these two procedures.

8.1 Checkpoint Initialization

In the beginning, there is no available checkpoint, so you need to set `CP_VALID` to 0 before the game is initialized.

8.2 Memory copy

Inside saving or restoring the checkpoint, there will be plenty of memory copy operations. For convenience, it is **highly suggested (not compulsory)** that you should implement a procedure for copying a memory region from one address to another address. This procedure should be flexible enough to be reused when saving and restoring the game.

8.3 save_checkpoint

This procedure will first check whether the score is a multiple of 10 and then, if it is, set the `CP_VALID` to one and save the current game state to the checkpoint memory region specified in Table 1.

8.3.1 Arguments

- None.

8.3.2 Return Values

- `v0`: 1 if a checkpoint is created. Otherwise, 0.

8.4 restore_checkpoint

Procedure `restore_checkpoint` should be called when the player presses the `checkpoint` button to restore a game checkpoint. The procedure should first check whether the current checkpoint is valid by reading the value at `CP_VALID`. If it is valid (the value is one), the procedure should overwrite the current game state with the one in the checkpoint.

8.4.1 Arguments

- None.

8.4.2 Return Values

- register `v0`: 1 if the checkpoint is valid. Otherwise, 0.

8.5 Exercise

Based on your understanding, write a complete `main` procedure for Snake game using procedures defined before. Figure 7 gives a flowchart of a reference `main` procedure. However, you can implement your own `main` procedure as long as the game flow performs like what a snake game should perform.

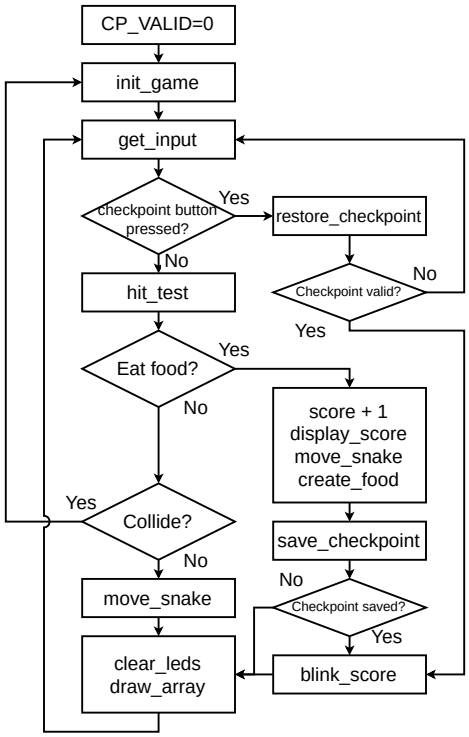

Figure 7: Program flowchart of `main` procedure.

Simulate your program to verify it. Before playing the game on Gecko4EPFL, read Section 9.

9 Running your Program on the Gecko4EPFL

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

9.1 Wait Procedure

You need to implement `wait` procedure to give some time to the player to respond to the update of the game state. The explicit wait is necessary, because the execution time of game procedures is short. For instance, if FPGA's clock speed is 50 MHz, and if it takes at most 1,000 instructions to update the game state during each iteration, the total time will amount to only 20 us. In contrast, an appropriate time for each iteration of the game should be 0.5 s to 1 s.

The goal of the `wait` procedure is to simply do nothing useful for a certain period of time. For example, inside the procedure you can initialize a large counter, decrease it slowly with a loop, and return when the counter reaches 0. The initial value of the counter can be estimated based on the FPGA clock frequency and your expected latency.

You can easily figure out where to call this procedure by yourself, but we suggest that the following places are considered:

- The start of the game iteration, e.g., before `get_input`.
- After the player loses the current game.
- Inside `blink_score`, you should use the `wait` procedure between turning off the 7-segment display and calling `display_score`.

> **Note**: 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.

9.2 Workflow

- Please use the **Nios II** CPU provided in the Quartus project `quartus/GECKO.qpf` in the template.
- 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

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 the loss of points in the final grading.

You are expected to submit your complete code as a single .asm file. The automated grader will look for and test the following API procedures: `clear_leds`, `set_pixel`, `get_input`, `move_snake`, `draw_array`, `hit_test`, `display_score`, `create_food`, `init_game`, `save_checkpoint` and `restore_checkpoint`. Make sure that you follow the formatting instructions detailed in Section 1.2.

Each of the above listed procedures is tested independently. If a procedure calls a helper procedure which is not in the above list, please make sure to enclose the definition of the helper function inside your procedure (see Section 1.2).

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 listed above by analyzing their effect on memory contents and registers. The final feedback will resemble the following: *Procedure procedure_name passed/failed the test*. There are two potential error types you may encounter:
  - `failed` means your procedure did not implement the functionality we expect. The issue can be solved by testing more use cases, as well as carefully looking into the log and comparing what is required with what you have implemented.
  - `build or execution failed` means either you have compiling errors, or your program did not end normally. In this case, first check if there is a compilation error (preliminary grader can be used for this purpose). After that, verify if your code has an infinite loop or tries to access an invalid memory address.

If your code passes all the tests in **snake-final**, you will obtain the maximum score of 80%. For the remaining 20%, we will need to see a successful live demonstration of the game on Gecko4EPFLboard. It will be the teaching assistants who will play the demo using the **last** submission made to **snake-final** prior to the deadline.