A Game of life 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

In this lab, the goal is to implement a simplified version of the **`Game of Life`** in assembly language. At the end of the lab, you should be able to play it on **Gecko4EPFL** board.

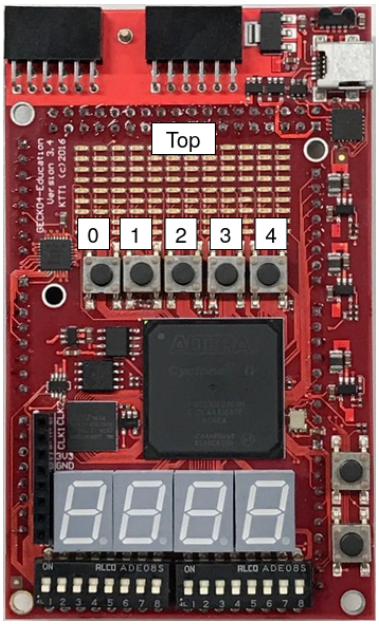


Figure 1: Gecko4EPFL

1.1 General Description

The **Game of Life** is a cellular automaton devised by British mathematician John Conway in 1970. The game requires no players: its evolution is determined by its initial state (also called the seed of the game). The playing field of the game is an infinite two-dimensional grid of cells, where each cell is either alive or dead. At each time step, the game evolves following this set of rules:

- **Underpopulation**: any living cell dies if it has (strictly) fewer than two live neighbours.
- **Overpopulation**: any living cell dies if it has (strictly) more than three live neighbours.
- **Reproduction**: any dead cell becomes alive if it has exactly three live neighbours.
- **Stasis**: Any live cell remains alive if it has two or three live neighbours.

When your game will be complete, you will able to have behaviours similar to the one shown in fig. 2.

The goal of this lab will be to implement an assembly version of the game of life. In addition to the previous rules, we will add some control functions to the game, as well as walls, where no cell could ever be alive in them. We first describe the conventions that your code needs to follow to meet the grading requirements, then we give a high level description of the code organisation, and finally, detail the functions you should implement.

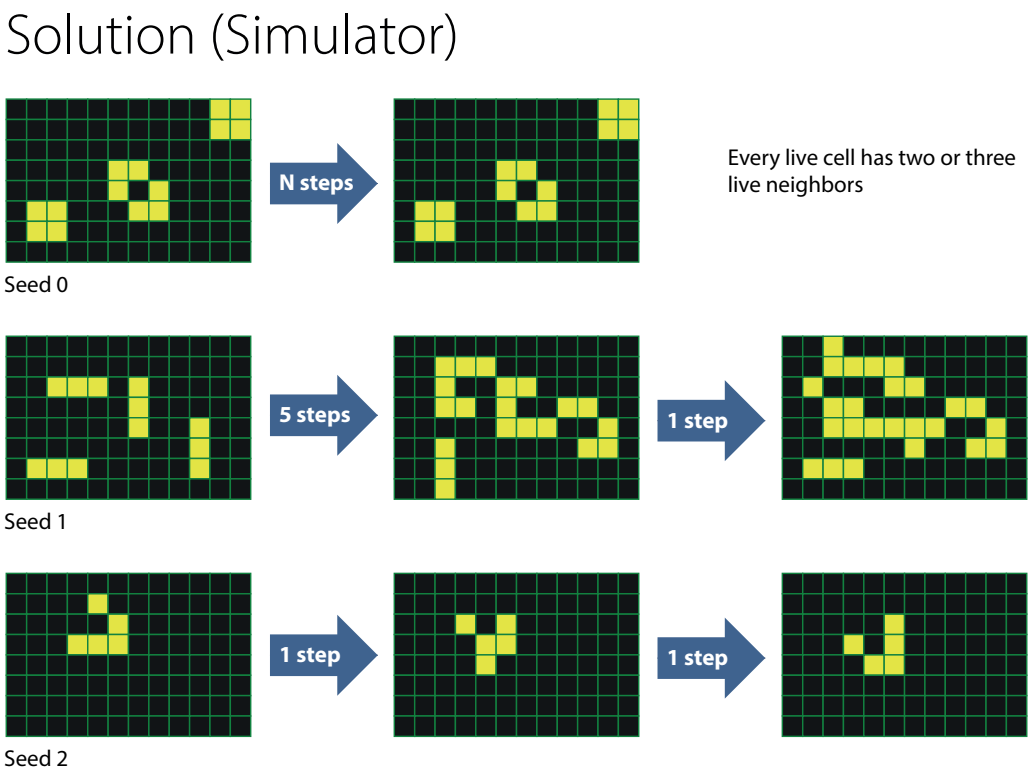


Figure 2: Example of a game iteration from left to right.

1.2 Constants

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 line below

```
.equ LEDS,  0x2000
```

will associate value `0x2000` to symbol `LEDS`. In the code, whenever you write `LEDS`, that will have the same effect as if you write `0x2000`, but your code will be much more readable and easier to update. Example of use:

```
ldw  t1,   LEDS (zero)    ; load the LEDS in t1
```

We have prepared for you a list of useful symbol/value pairs in a template file which will be provided to you. **For the correct grading of the game, we strongly advise you to use the list below, without any modification! If you choose different symbol names or values, the grader may fail and you may lose points.**

```
;;    game state memory location
.equ CURR_STATE, 0x1000       ; Current state of the game
.equ GSA_ID, 0x1004           ; ID of the GSA holding the current state
.equ PAUSE, 0x1008            ; Is the game paused or running
.equ SPEED, 0x100C            ; Current speed of the game
.equ CURR_STEP,  0x1010       ; Game current step
.equ SEED, 0x1014             ; Which seed was used to start the game
```

```
.equ GSA0, 0x1018              ; Game State Array 0 starting address
.equ GSA1, 0x1038              ; Game State Array 1 starting address
.equ SEVEN_SEGS, 0x1198        ; 7-segment display addresses
.equ CUSTOM_VAR_START, 0x1200 ; Free range of addr. for custom variables
.equ CUSTOM_VAR_END, 0x1300    ; End of the free range of addresses
.equ LEDS, 0x2000              ; LEDs
.equ RANDOM_NUM, 0x2010        ; Random number generator
.equ BUTTONS, 0x2030           ; Buttons

;; states
.equ INIT, 0
.equ RAND, 1
.equ RUN, 2

;; constants
.equ N_SEEDS, 4            ; number of available seeds
.equ N_GSA_LINES, 8        ; number of gsa lines
.equ N_GSA_COLUMNS, 12     ; number of gsa columns
.equ MAX_SPEED, 10         ; maximum speed
.equ MIN_SPEED 1           ; minimum speed
.equ PAUSED, 0x00          ; game paused value
.equ RUNNING, 0x01         ; game running value
```

1.3 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. **Please pay attention to spelling and spacing of the opening and closing macros.**

- If your procedure makes calls to other, auxiliary procedures, all those auxiliary procedures must also be entirely enclosed between `; BEGIN:helper` and `; END:helper` comments. The auxiliary procedures may have whatever name you choose. You can also add any newly defined constants in the helper block.

```
; BEGIN:helper
my_helper_procedure_name1:
; your implementation code
ret

my_helper_procedure_name2:
; your implementation code
ret
; END:helper
```

```
; BEGIN:procedure_name:
procedure_name:
; your implementation code
call my_helper_procedure_name1
; your implementation code
ret
; END:procedure_name
```

- Have all the procedures inside a **single** `.asm` file.

Our grading system will check each procedure individually and separately from the rest of your assembly code.

Finally, in order to ensure a correct grading, please **respect the coding conventions: especially when pushing and popping from the stack: it must grow downward**.

2 Game mechanics

This section gives a high level description of the different components of the game and their interactions.

2.1 Terminology

The game is displayed on a LED array, where each pixel is a **Cell**. Each **Cell** can either be in the **dead** state or the **alive** state. A **wall** is an always-dead cell.

A **seed** is an initial state of the game.

A **step** is the result of applying the game rules from one game state to the next.

2.2 Game representation

The game display is a LED array of 8×12 pixels. The top left corner of the array is the coordinate system's origin. The x-axis grows rightward while the y-axis grows downward. An example configuration can be seen in Fig. 3. The game display is a torus, which means that two cells (x_1, y_1) and (x_2, y_2) are considered neighbours if one of the following conditions is satisfied (symbol $==$ stands for equality):

- $(x_1 + 1 \mod 12,\ y_1 + 1 \mod 8) == (x_2, y_2)$
- $(x_1 + 1 \mod 12,\ y_1 \mod 8) == (x_2, y_2)$
- $(x_1 \mod 12,\ y_1 + 1 \mod 8) == (x_2, y_2)$
- $(x_1 - 1 \mod 12,\ y_1 - 1 \mod 8) == (x_2, y_2)$
- $(x_1 - 1 \mod 12,\ y_1 \mod 8) == (x_2, y_2)$
- $(x_1 \mod 12,\ y_1 - 1 \mod 8) == (x_2, y_2)$
- $(x_1 - 1 \mod 12,\ y_1 + 1 \mod 8) == (x_2, y_2)$
- $(x_1 + 1 \mod 12,\ y_1 - 1 \mod 8) == (x_2, y_2)$

In order to display elements on the LEDs, we will use two representations. One will be the game state array (GSA), a convenient representation on which the game operations are easily performed. The other one will be the display representation, a more compact but directly displayable representation. We then describe how these two representations work and interact.

2.2.1 GSA

The GSA is made of *GSA elements*. Each GSA element is a horizontal line on the screen and is stored in a single word in memory. Indeed, a word is 32 bits and there are only 12 cells per line. As each cell has either the value 0 (dead) or 1 (alive), the cell sate can be stored in a bit. The leftmost cell of a row is mapped to the least significant bit of the corresponding GSA word, and going rightwards goes to higher significant bits. This is visualized in Fig. 4 which shows how a line is represented in the GSA. The bits having an index higher than 11 are not used and must then be 0. 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. Finally, the complete GSA is made out of 8 GSA elements. The GSA element of Fig. 4 is line three in the exemplary GSA of Fig. 3.

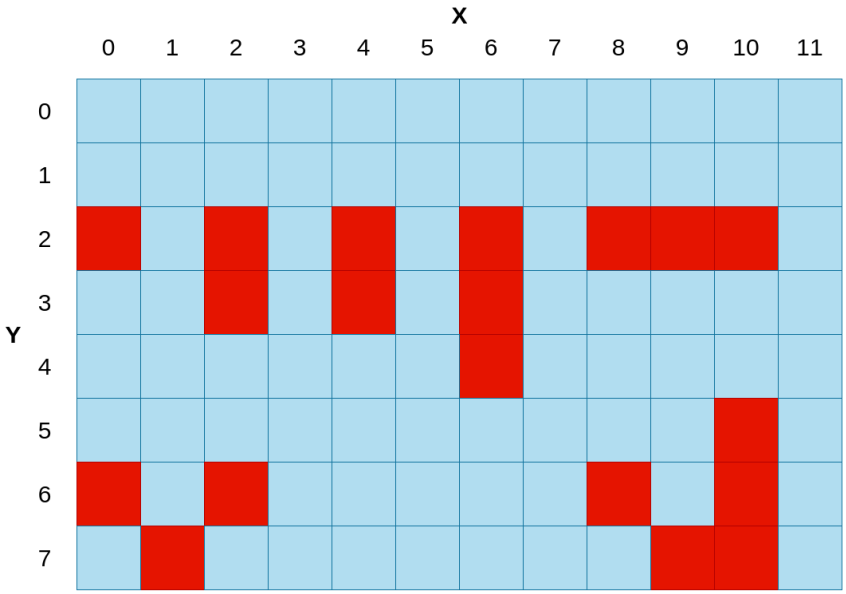


Figure 3: Coordinate system

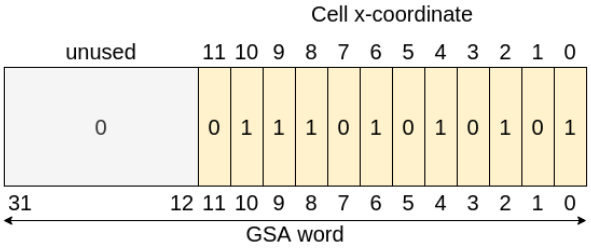


Figure 4: GSA element

2.2.2 GSA step

The **`Game of Life`** rules (presented in section 1.1) are applied to the GSA. The next state must only depend on the current one, and hence every cell dying/coming to life must not have any influence on the current state. An easy way of ensuring this is to have a pair of GSAs. One will hold the current valid GSA state, and the other one will hold the next value for the GSA. This way, when performing a step, values are read from the current GSA, and written to the next one. At the end of each update, the GSAs are inverted: current GSA become the next GSA and next GSA become the current one.

Finally, in order to apply the **`Game of Life`** rules, the neighbours of a cell must be known. They are defined in Fig. 5: considering the target cell, each pixel that is at most 1 jump away from it in either or both x and y directions. A jump is defined as adding or removing 1 from either x or y coordinate. Of course, one needs to take into account the torus topology and hence the required modulo.

2.2.3 LED array

The LED array representation consists of three memory mapped registers, where each bit in the registers is mapped to a LED. If it is one then the corresponding LED is lit, otherwise the corresponding LED is off. Fig. 6 shows the mapping between register bits and LEDs. A single function will be in charge of mapping the content of the GSA to the LED array. This helps isolating the complexity of the mapping to one place while providing an easy internal representation.

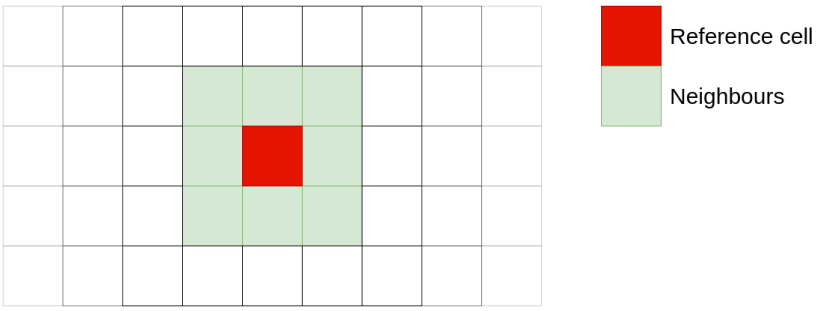
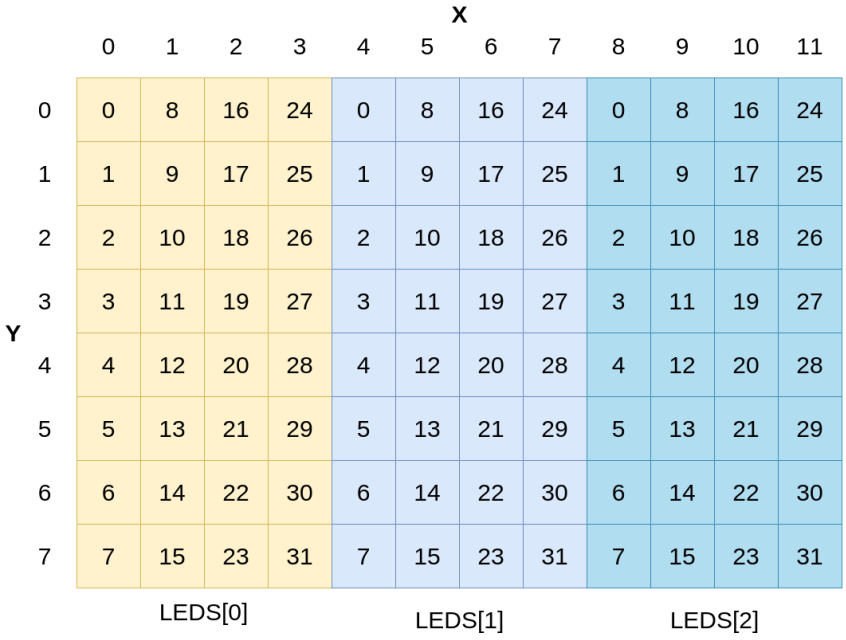


Figure 5: Neighbours



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

Figure 6: LED Mapping

2.3 Walls

Finally, let us explain the mechanism behind walls: masking. As the next state depends only on the current one (Section 2.2.2), one can first apply the **`Game of Life`** rules as if there were no walls, and, before drawing the result back on the screen, set all wall locations to the dead state. Any potential wall pixel on the alive state would then never be seen nor impact the game. Masking is an easy procedure that complies with our game representation. Masks will have 8 words, similar to a GSA, consisting of `0`s and `1`s where a `0` bit means a wall is at this location. Applying the wall mask is then as simple as AND-ing all GSA word with the corresponding in-use mask. An example of this procedure can be seen in Fig. 7.

2.4 Control extensions

Control extensions are the procedures allowing the user to setup the game parameters, change them while the game is running. These procedures are the interface to the game and they are accessible through 5 buttons on the board. In this part, we describe the action of each button. These actions depend on the current states of the game which can be:

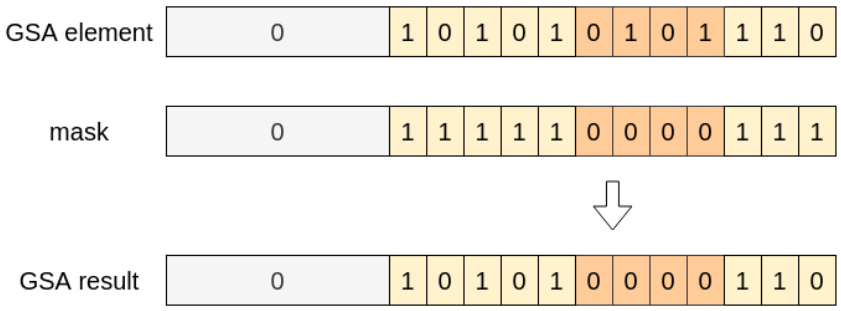


Figure 7: GSA masking

- `INIT`: This is the starting state and the game will come back to it after each run or upon reset. In this state, the seed and the game duration is configured from predefined ones.
- `RAND`: This state is reached from the initial state by pushing `button 0` *N* times where N is an integer representing the number of seeds. In this state, the seed and game duration is initialized to a random seed.
- `RUN`: In this state, the game runs and the user has a few possibilities to change the way the game runs.

Each state is explained more in the following:

2.4.1 `INIT` state

In the `INIT` state, the player can select the seed and set the number of steps the game will run for. The preconfigured seeds are already stored in memory, and should simply be displayed one after the other when pressing `button 0`.

The button mapping is the following:

- **`button 0`**: By pushing `button 0`, the user will go through the predefined seeds, one after the other. N seed and mask pairs are available. By default, seed 0 is displayed, and if the game is launched from this configuration, mask 0 must be used for masking. Pushing `button 0` again selects the next seed mask pair. When `button 0` has been pushed N times, it triggers a transition to the state `RAND`.
- **`button 1`**: Starts the game from the selected initial state for the desired amount of steps.
- **`button 2-3-4`**: These buttons are used to set the number of steps the game will run for by configuring the last three digits of the LCD display. The first digit can be initialized to any value, while `button 4` configures the units, `button 3` the tens, and `button 2` the hundreds. The number of steps the game will run is in hexadecimal. For example, if the number displayed on the LCD is 870, this in fact means that the game will run 2160 steps. Moreover, to set the number to 870 a player would need to push `button 2` 8 times and `button 3` 7 times and `button 4` 15 times (to overflow the intial 1 to 0). By default the game runs for 1 step.

2.4.2 `RAND` state

When the state transitions to the `RAND` state from the `INIT` state, a random seed must be generated. A random seed is defined as each cell being put in the alive or dead state randomly. To avoid meaningless configurations, the random seed is associated with always the same mask: mask N+1. There are then N predefined seeds and N+1 predefined masks, counting the one for the random state. As with the

predefined seeds, the random mask must be used for one whole run.

In this state, the button mapping is the following:

- **`button 0`**: Pushing it again triggers the generation of a new random game state.
- **`button 1`**: Starts the game from the selected random game state for the amount of steps selected.
- **`button 2-3-4`**: These buttons are used to set the number of steps the game will run for by configuring the last three digits of the LCD display. The first digit can be any value, while `button 4` configures the units, `button 3` the tens, and `button 2` the hundreds. The number of steps the game will run is in hexadecimal. For example, if the number displayed on the LCD is `870`, this in fact means that the game will run 2160 steps. Moreover, to set the number to `870` a player would need to push `button 2` 8 times and `button 3` 7 times and `button 4` 15 times (to overflow the intial 1 to 0). By default the game runs for 1 step.

2.4.3 `RUN` state

This state is reached from the `INIT` or `RAND` state by pressing `button 1`. When entering this state, the game will be automatically set to run (Game paused = 1), and will play the game until either the pause button is pressed or until we play for the selected number of steps. It offers the player a few control elements, listed bellow:

- **`button 0`** is the start/pause button. If pressed, the game toggles between play and pause.
- **`button 1`** increases the speed of the game.
- **`button 2`** decreases the speed of the game.
- **`button 3`** is the reset button. It clears the initial board selection, the number of steps, and stops the game.
- **`button 4`** replaces the current game state with a new random one

When the game hangs on a configuration where nothing happens anymore or the screen becomes empty, `button 4` can replace the GSA with a more interesting configuration.

2.4.4 State machine

Fig. 8 summarizes the state transition. In this figure, bX shows a button. $b0 = N$ is the event of pushing button b0 N times in the `INIT` state, and $b0 < N$ is the reverse (button b0 is pushed less than N times).

2.5 Memory layout

Table 1 shows the precise RAM memory layout for the game. Please keep **exactly** this memory layout as otherwise you will lose points from the automated grading. In Table 1, we have considered a free space, Custom Variables, for the variables you may define when implementing the game. In Section 3, each memory location is explained in more details.

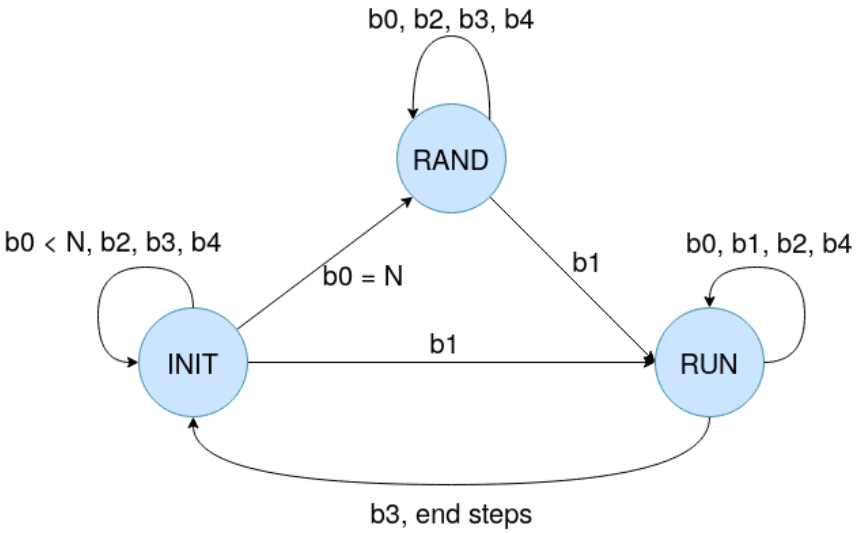


Figure 8: State machine

3 Implementation

We will now present the functions required to run the game, which you should implement.

3.1 Drawing using the LEDs

Your first exercise is to implement the following two procedures for controlling the LEDs, and a procedure to add execution delay:

1. `clear_leds`, which initializes the display by switching off all the LEDs,
2. `set_pixel`, which turns on a specific LED. This function is here purely for exercise, and will most probably not be used anywhere in the code
3. `wait`, which creates an execution delay.

The LED array has 96 pixels (LEDs). Fig. 9 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.

3.1.1 Procedure `clear_leds`

The `clear_leds` procedure initializes all LEDs to 0 (zero). You should call `clear_leds` before drawing a new GSA on the screen (see the algorithm at the end of this document).

Arguments

- None

Return Values

- None.

0x1000	Game Current State
0x1004	GSA ID
0x1008	Game paused
0x100C	Game speed
0x1010	Game current step
0x1014	Game seed
0x1018	Game State Array 0 (GSA)
0x101C	8 words (32b each)
...	...
0x1034	
0x1038	Game State Array 1 (GSA)
0x103C	8 words (32b each)
...	...
0x1054	
...	...
0x1198	SEVEN_SEGS[0]
	SEVEN_SEGS[1]
	SEVEN_SEGS[2]
0x11A4	SEVEN_SEGS[3]
0x1200	Custom Variables
...	...
0x1300	
...	...
0x2000	LEDS[0]
	LEDS[1]
0x2008	LEDS[2]
...	...
0x2010	RANDOM_NUM
...	...
0x2030	BUTTONS
0x2034	
...	...

Table 1: RAM memory organization for keeping the current state of the game.

3.1.2 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**.

Arguments

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

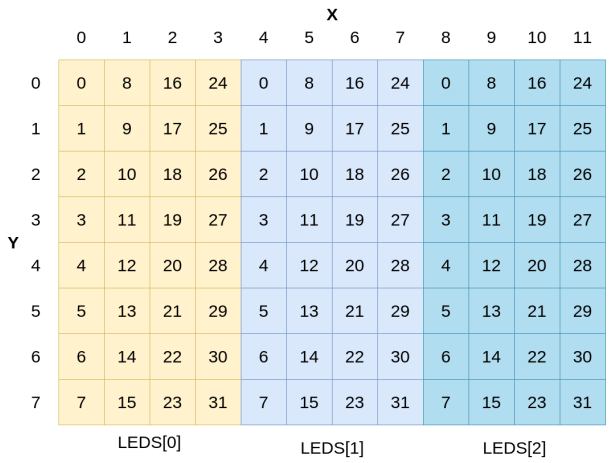


Figure 9: Translating the LED **x** and **y** coordinates into the corresponding bit in the LED array. For example, **x** = 5 and **y** = 3 correspond to the bit 11 in the word `LEDS[1]`.

Return Values

- None.

3.1.3 Procedure `wait`

The `wait` procedure serves to add a delay to the execution of the program. This delay can be created by initializing a very large counter value in a register and decrementing it in a loop. This way, the execution time needed to decrement the counter to zero will create a time delay. A good value for the delay is approximately 1s, and for the Gecko4EPFL board this can be done using an initial counter value of 2^{19}. However, the `Game speed` variable from the RAM has to be taken into account: it specifies how fast the game executes. This variable can take values between 1 and 10. If 1, the game runs at the regular speed with the delay of approximately 1 s, while if it is 10, the game runs at the maximum speed. Depending on the value of the `Game speed` variable in the RAM, the original game speed is increased `Game speed` times.

Beware that when simulating the assembly program in **nios2sim**, the `wait` procedure can cause the simulation to run too slow. In this case, it is best to either test the `wait` procedure directly on Gecko4EPFL, or to significantly reduce the initial value of the counter for simulation.

Arguments

- None.

Return Values

- None.

3.2 GSA handling procedures

Setting and getting a GSA word will be frequent operations, hence, we define two helper procedures doing the work.

- `get_gsa`: gets an element from the GSA
- `set_gsa`: sets an element of the GSA

3.2.1 Procedure `get_gsa`

This procedure gets as the argument a line location, y, where $0 \leq y \leq 7$ and returns the GSA element at the location of y. This procedure must take into account the `GSA ID` location in RAM because this flag indicates which GSA is currently in use. The `GSA ID` flag should always be either 0 or 1.

Arguments

- register `a0`: line y-coordinate

Return Value

- register `v0`: Line at location y in the GSA

3.2.2 Procedure `set_gsa`

This procedure gets a line as the argument and sets it at the specified location in the GSA. It must also use the `GSA ID` flag, similar to the `get_gsa` function.

Arguments

- register `a0`: the line
- register `a1`: y-coordinate

Return Values

- None.

3.3 From the GSA to the LEDs

3.3.1 Procedure `draw_gsa`

The `draw_gsa` procedure takes the GSA currently in use and reproduce it on the LEDs.

Arguments

- none

Return Value

- None

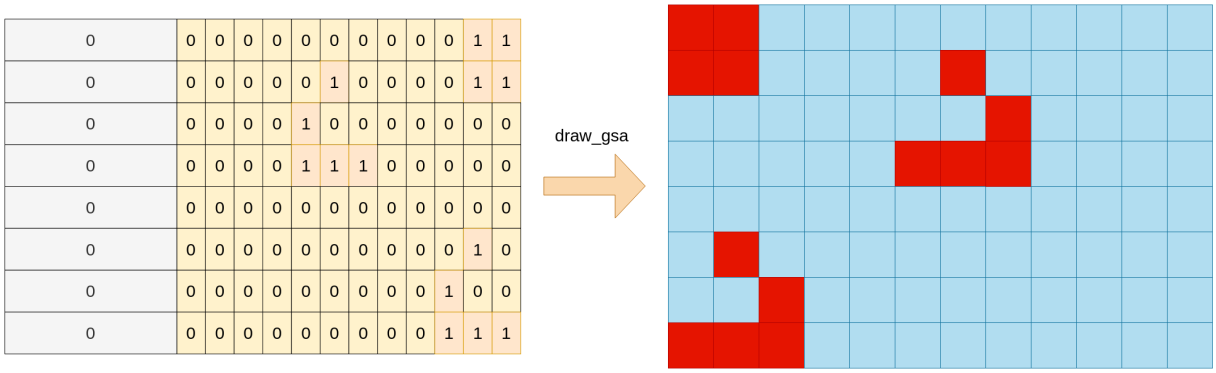


Figure 10: Example of draw_gsa application

3.4 Using a random GSA

3.4.1 Procedure `random_gsa`

The `random_gsa` procedure initialize the current GSA to a random state. Again, this procedure knows which GSA is currently in use thanks to the `GSA ID` flag. For each pixel, it must draw a random value from the `RANDOM_NUM` location, a random number generator, and convert it to either dead or alive. As the returned random value is 32 bits, we must convert it to one bit. Many possibilities exist, but here we will take the modulo 2 operation as it is very easy to perform. If the modulo two value is 0, then the cell will be dead, otherwise it will be alive.

Arguments

- none

Return Value

- None

To simulate random number generation in **nios2sim**, you can write a value of your choice at the location `RANDOM_NUM`. Inside **nios2sim** this memory location overlaps with the address space reserved for UART (Fig. 11); 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`. Whichever value you write here will be read as a random number from the location `RANDOM_NUM`.

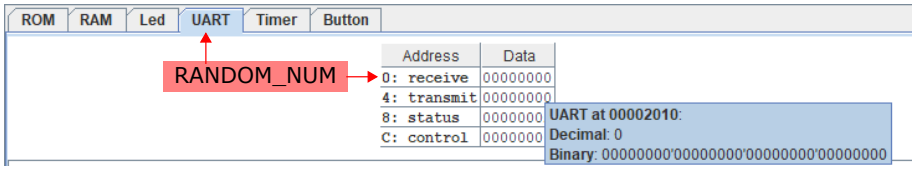


Figure 11: The location where you should write a random number for simulating the game in **nios2sim**.

3.5 Action functions

This section introduces all the control functions of the game.

3.5.1 Procedure `change_speed`

The `change_speed` procedure increases or decreases the game speed depending on the argument and updates the value of the game speed in the RAM. By default this value is 1, and it cannot be smaller. Each time it is increased, the speed value is incremented by one, up to 10. It cannot be bigger than 10. Similarly, when the speed is decreased, the game speed value is decremented by 1, down to 1.

Arguments

- register `a0`: 0 if increment, 1 if decrement.

Return Value

- None

3.5.2 Procedure `pause_game`

The `pause_game` procedure pauses or resumes the game depending on the current state by setting the `Game paused` variable in RAM. A value of 0 means that the game is paused, and 1 that it is running (Section 1.2). The pause procedure must invert the currently stored value of `Game paused`.

Arguments

- None

Return Value

- None

3.5.3 Procedure `change_steps`

The `change_steps` procedure changes the number of steps that the game will run based on the input arguments. As discussed in the Section 2, `button 2` is for the hundreds, `button 3` represents the tens and `button 4` the units. Each digit is represented in hexadecimal base (from `0` to `F`). Keep in mind that more than one of the arguments can be set to 1.

Arguments

- register `a0`: 1 if b4 pressed, 0 otherwise
- register `a1`: 1 if b3 pressed, 0 otherwise
- register `a2`: 1 if b2 pressed, 0 otherwise

Return Value

- None

3.5.4 Procedure `increment_seed`

The `increment_seed` procedure changes the value of `Game seed` based on the current state of the game. If the game state is `INIT`, it increments the `Game seed` by one, and copies the new seed in the current GSA. If the game state is `RAND`, this procedure must generate a new random GSA. Please note when starting from a seed, the mask associated to that specific seed needs to be used for the next steps. In the template file (Section 1.2), you will find 4 seeds and 5 masks. The first seed will be associated with the first mask, etc.. The last mask is the one associated to the random state.

Return Value

- None

3.5.5 Procedure `update_state`

The `update_state` procedure checks if the `edgcapture` register, which is given as the input, requires a change of state, and if necessary performs it. For any change of state from `RAND` or `RUN` to `INIT`, the `reset_game` procedure has to be called.

Arguments

- register `a0`: `edgecapture`

Return Value

- None

3.5.6 Procedure `select_action`

The `select_action` procedure calls the correct action function depending on the button pressed.

Arguments

- register `a0`: a copy of the `edgecapture` register

Return Value

- None

3.6 Updating the GSA

3.6.1 Procedure `cell_fate`

The `cell_fate` procedure returns the next state of a cell depending on the number of living neighbours which is passed to the procedure as an argument.

Arguments

- register `a0`: number of live neighbouring cells.
- register `a1`: examined cell state.

Return Value

- 1 if the cell is alive 0 otherwise

3.6.2 Procedure `find_neighbours`

The `find_neighbours` procedure takes a cell location as the argument and returns the number of this cell's living neighbours as well as the value of the cell itself.

Arguments

- register `a0`: x coordinate of examined cell
- register `a1`: y coordinate of examined cell

Return Value

- register `v0`: number of living neighbours for the cell at location `x` in the middle line
- register `v1`: state of the cell at location `(x, y)`, i.e., the cell for which we are counting the living neighbours.

3.6.3 Procedure `update_gsa`

The `update_gsa` procedure updates the next GSA according to the **`Game of Life`** rules. When the update done, this procedure must invert the `GSA ID` (As explained in Section 2.2.2). If the game is paused, this procedure should not do anything.

Arguments

- None

Return Value

- None

3.6.4 Procedure `mask`

The `maks` procedure applies the mask corresponding to the selected seed to the current GSA. Please note that the mask for the configuration with the random seed is the last mask stored in the code, so if N seeds are predefined, it will be the mask N+1. In our case, it will be mask4.

Arguments

- None

Return Value

- None

3.7 Inputs to the Game

Interacting with the game is the responsibility of the `get_input` procedure, which reads the state of the push buttons and returns it. This information will be passed to the action functions.

The five push buttons of the Gecko4EPFL are read through the **Buttons** module, which are memory mapped (see Table 1). This module has two 32-bit words, called `status` and `edgecapture`, described in Table 2. To implement `get_input`, you will need to use `edgecapture` (you can ignore `status`).

Address	Name	31 . . . 5	4 . . . 0
`BUTTONS`	`status`	*Reserved*	State of the Buttons
`BUTTONS+4`	`edgecapture`	*Reserved*	Falling edge detection

Table 2: The two words of the **Buttons** module.

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.

3.7.1 Procedure `get_input`

The `get_input` procedure reads the `edgecapture` register, returns its value and clears the `edgecapture` register.

Arguments

- None

Return Value

- register `v0`: `edgecapture`

In case multiple buttons are pressed simultaneously, you can decide the processing order. One possible option is to only concider the least significant bit of `edgecapture` which is set, while discarding the others.

3.8 Game step handling

3.8.1 Procedure `decrement_step`

If game state is `RUN` and the game is running, the `decrement_step` procedure checks if the current step is 0 or not. In case of 0, it returns 1, otherwise, decrements the number of steps, displays it on the 7-SEG display, and returns 0. If the game state is `INIT` or `RAND`, it displays the number of steps on the 7-SEG display and returns 0.

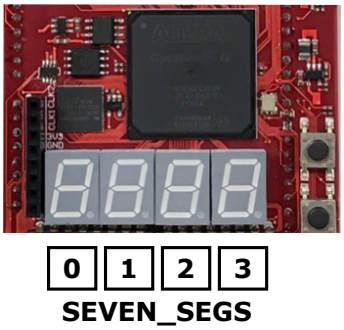


Figure 12: 7SEG mapping

Arguments

- None

Return Value

- register `v0` 1 if done 0 otherwise

In order to display one hexadecimal digit on the 7-segment display, you can use the following array (mapping), which is already specified for you in the template file (Section 1.2):

```
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
  .word 0xEE  ; A
  .word 0x3E  ; B
  .word 0x9C  ; C
  .word 0x7A  ; D
  .word 0x9E  ; E
  .word 0x8E  ; F
```

3.9 Reset

Finally, we need a function that resets the game to its default state: `reset_game`.

3.9.1 Procedure `reset_game`

The `reset_game` procedure puts the game in its default state. The default state is defined as follows:

1. Current step is 1 and displayed as such on the 7-SEG display
2. The seed 0 is selected
3. Game state 0 is initialized to the seed 0
4. `GSA ID` is 0
5. The game is currently paused
6. The game speed is 1

Arguments

- None

Return Value

- None

3.10 Putting everything together

The algorithm of the game is:

Algorithm 1: `Game of Life`

```
while True do
    reset_game()
    e ← get_input()
    done ← false
    while !done do
        select_action(e)
        update_state(e)
        update_gsa()
        mask()
        draw_gsa()
        wait()
        done ← decrement_steps()
        e ← get_input()
    end
end
```

4 Playing the Game

Now that you have implemented all the required core functionality, it is time to test if the game runs smoothly end to end. You can do this by simulating your program in the **nios2sim** simulator. A better way to do this is to run the program on the Gecko4EPFLboard.

The steps necessary to run the program on the **Gecko4EPFL** board are the following:

- 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.

While implementing the **`Game of Life`** game, you might have 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.

5 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`, `wait`, `set_gsa`, `get_gsa`, `draw_gsa`, `random_gsa`, `change_speed`, `mask`, `pause_game`, `change_steps`, `increment_seed`, `update_state`, `select_action`, `cell_fate`, `get_input`, `decrement_step`, `reset_game`, `find_neighbours`, and `update_gsa`. Make sure that you follow the formatting instructions detailed in Section 1.3.

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 undeclared procedures your code calls (see Section 1.3).

There are two submission links: **GoL-preliminary** and **GoL-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 **GoL-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 Gecko4EPFL. This year, exceptionally, it will be the teaching assistants who will run the demo using the last submission made to **GoL-final**.