A Game of life in Assembly Language

Learning Goal: Write a complete program in assembly language and run it on a RISC-V processor.

Requirements: Multicycle RISC-V processor, Gecko5 Simulator board (VSCode Extension).

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 the **Gecko5** board.

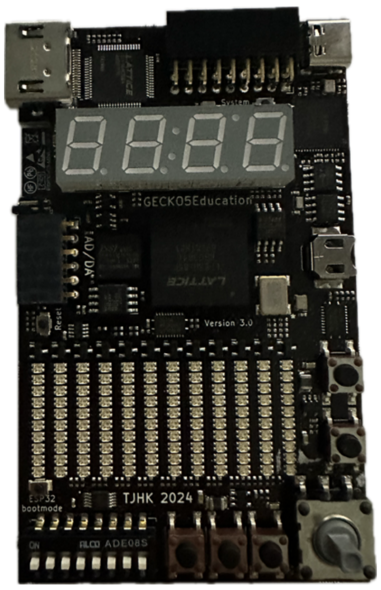


Figure 1: Gecko5

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.

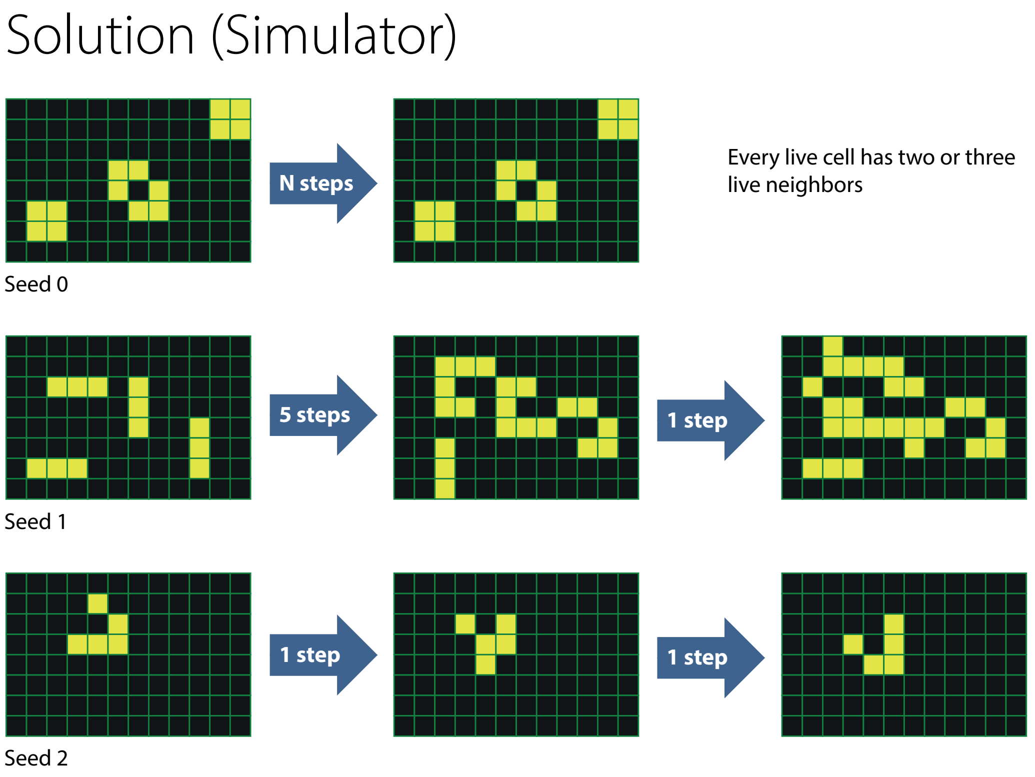


Figure 2: Example of a game iteration from left to right.

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,  0x50000000
```

will associate value `0x50000000` to symbol `LEDS`. In the code, whenever you write `LEDS`, that will have the same effect as if you write `0x50000000`, but your code will be much more readable and easier to update. Example of use:

```
li t1, LEDS
lw t1, 0(t1)    /* load the LEDS address 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 locations */
.equ CURR_STATE, 0x90001000       /* Current state of the game */
.equ GSA_ID, 0x90001004           /* ID of GSA holding the curr state */
.equ PAUSE, 0x90001008            /* Is the game paused or running */
.equ SPEED, 0x9000100C            /* Current speed of the game */
.equ CURR_STEP,  0x90001010       /* Current step of the game */
```

```
.equ SEED, 0x90001014                /* Seed used to start the game */
.equ GSA0, 0x90001018                /* Game State Array 0 starting addr */
.equ GSA1, 0x90001058                /* Game State Array 1 starting addr */
.equ CUSTOM_VAR_START, 0x90001200 /* Start of addresses for custom vars */
.equ CUSTOM_VAR_END, 0x90001300   /* End of addresses for custom vars */
.equ RANDOM, 0x40000000              /* Random number generator address */
.equ LEDS, 0x50000000                /* LEDs address */
.equ SEVEN_SEGS, 0x60000000          /* 7-segment display addresses */
.equ BUTTONS, 0x70000004             /* Buttons address */

/* States */
.equ INIT, 0
.equ RAND, 1
.equ RUN, 2

/* Constants */
.equ N_SEEDS, 4               /* Number of available seeds */
.equ N_GSA_LINES, 10           /* 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 Gecko5 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 the same BEGIN and END.

- **Please do not modify the provided constants, as they are used for grading.** The undefined behavior caused by modifying them is more likely to affect your grade negatively!

- Have all the procedures inside a **single** `.s` 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**.

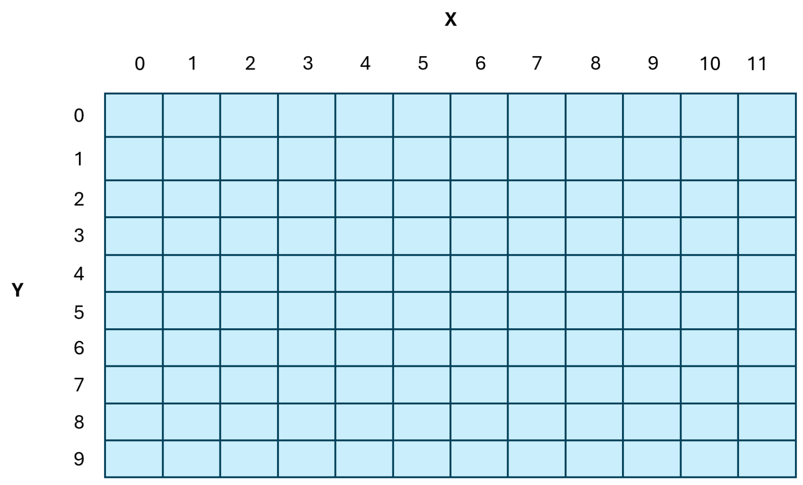


Figure 3: Coordinate system

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 10×12 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):

- $(x_1 + 1 \mod 12,\ y_1 + 1 \mod 10) == (x_2, y_2)$
- $(x_1 + 1 \mod 12,\ y_1 \mod 10) == (x_2, y_2)$
- $(x_1 \mod 12,\ y_1 + 1 \mod 10) == (x_2, y_2)$
- $(x_1 - 1 \mod 12,\ y_1 - 1 \mod 10) == (x_2, y_2)$
- $(x_1 - 1 \mod 12,\ y_1 \mod 10) == (x_2, y_2)$
- $(x_1 \mod 12,\ y_1 - 1 \mod 10) == (x_2, y_2)$
- $(x_1 - 1 \mod 12,\ y_1 + 1 \mod 10) == (x_2, y_2)$
- $(x_1 + 1 \mod 12,\ y_1 - 1 \mod 10) == (x_2, y_2)$

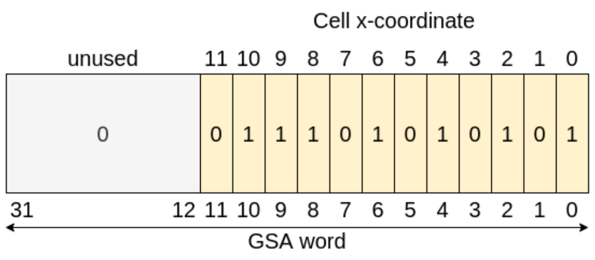


Figure 4: GSA element

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 10 GSA elements. The GSA element of Fig. 4 is line three in the exemplary GSA of Fig. 3.

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.

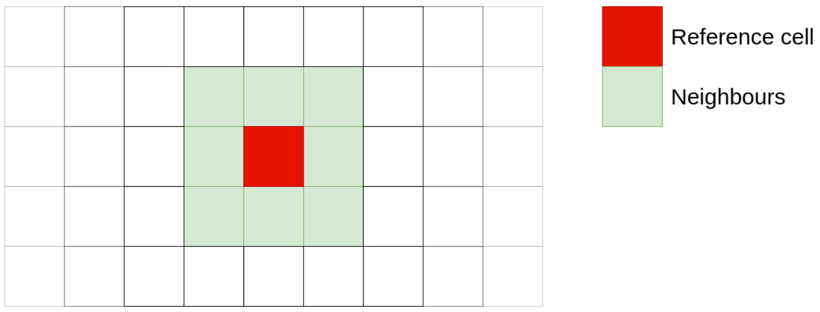


Figure 5: Neighbors of a cell.

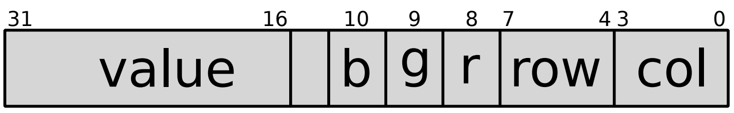


Figure 6: Memory map led register.

2.2.3 LED array

The RGB-LED array representation consists of a memory mapped register accessible at address 0x50000000 and depicted in Figure 6. Reading it returns 0, as the software should properly track LED states without depending on the hardware. The meaning of each field is presented in the next table.

Led Control Register		
Name	**Bit Field**	**Description**
col	[3:0]	Column selection. 4b1111 means all columns are selected.
row	[7:4]	Row selection. 4b1111 means all rows are selected
r	[8]	Updating red LEDs
g	[9]	Updating green LEDs
b	[10]	Updating blue LEDs
unused	[15:11]	–
value	[31:16]	New state of LEDs. • If no column or row is selected, all LEDs change their states to [16] • If a single column (but not row) is selected, all LEDs in that column change their states to [25:16] • If a single row (but not column) is selected, all LEDs in that row change their states to [27:16] • If both column and row are selected, the LED changes its state to [16]

For example, to change the states of all the red LEDs in column 3 to 0001111101 (from row 0 to row 9), one can write the following data to 0x50000000:

```
data[3:0] = 3 (or 4'0011); // for column 3
data[7:4] = 15 (or 4'1111); // for all rows in columns 3
data[10:8] = 1 (or 3'001); // for red
data[31:11] = 10b'1011111000
```

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. Wall pixels on the alive state would then never be seen nor impact the game. We will also make the walls visible for game clarity by painting them in BLUE, as seen in Fig. 7.

Masking is an easy procedure that complies with our game representation. Masks will have 10 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. 8.

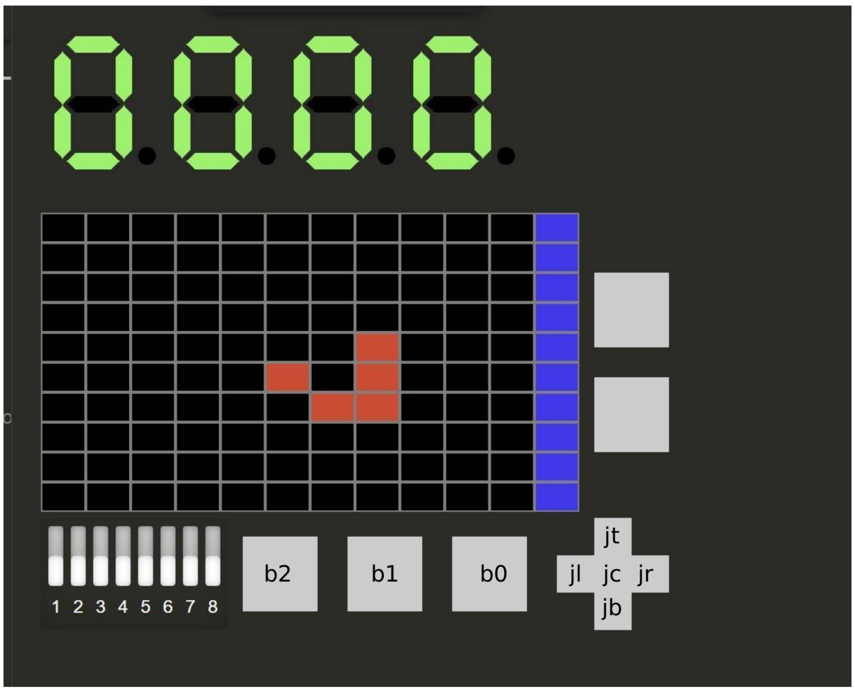


Figure 7: Our walls will be displayed in blue in the simulator

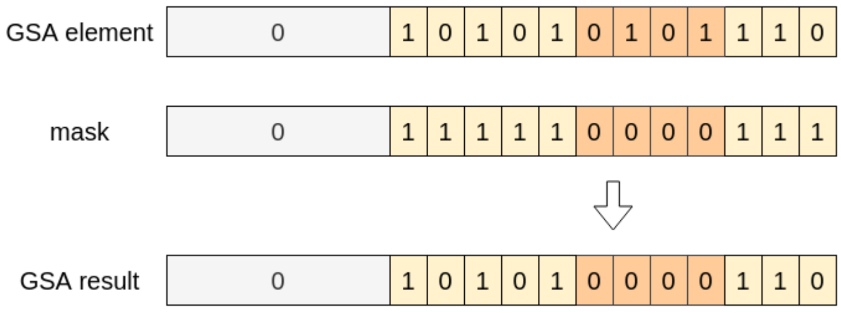


Figure 8: GSA masking

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 select 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:

- `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 `jc` *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 `jc`.

The button mapping is the following:

- **`jc`**: By pushing `jc`, 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 `jc` again selects the next seed mask pair. When `jc` has been pushed N times, it triggers a transition to the state `RAND`.
- **`jr`**: Starts the game from the selected initial state for the desired amount of steps.
- **`buttons 0-1-2`**: 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 0` configures the units, `button 1` 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 1` 6 times and `button 0` 15 times. 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:

- **`jc`**: Pushing it again triggers the generation of a new random game state.
- **`jr`**: Starts the game from the selected random game state for the amount of steps selected.
- **`button 0-1-2`**: 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 0` configures the units, `button 1` 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 1` 6 times and `button 0` 15 times. 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 `jr`. 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:

- **`jc`** is the start/pause button. If pressed, the game toggles between play and pause.
- **`jr`** increases the speed of the game.

- **`jl`** decreases the speed of the game.
- **`jb`** is the reset button. It clears the initial board selection, the number of steps, and stops the game.
- **`jt`** 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, `jt` can replace the GSA with a more interesting configuration.

2.4.4 State machine

Fig. 9 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).

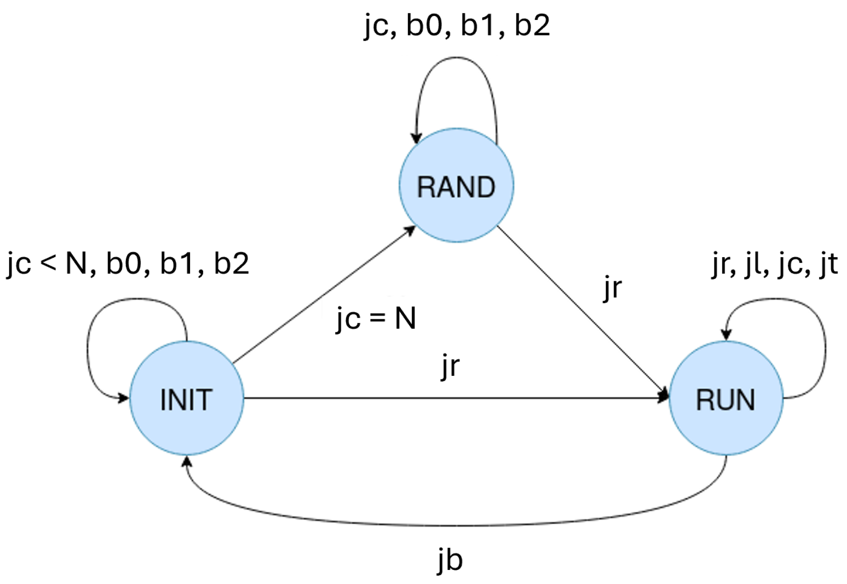


Figure 9: State machine

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.

`0x40000000`	RANDOM
...	...
`0x50000000`	LEDS
...	...
`0x60000000`	SEVEN_SEGS
...	...
`0x70000004`	BUTTONS
...	...
`0x90001000`	Game Current State
`0x90001004`	GSA ID
`0x90001008`	Game paused
`0x9000100C`	Game speed
`0x90001010`	Game current step
`0x90001014`	Game seed
`0x90001018`	Game State Array 0 (GSA)
`0x9000101C`	10 words (32b each)
...	...
`0x9000103C`	
`0x90001040`	Unused, allows for GSA0
`0x90001044`	and GSA1 to be 64 bytes apart
...	...
`0x90001058`	Game State Array 1 (GSA)
`0x9000105C`	10 words (32b each)
...	...
`0x90001074`	
`0x90001200`	Custom Variables
...	...
`0x90001300`	

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

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 using the memory mapped led register. We only use the red color in this lab.
3. `wait`, which creates an execution delay.

The LED array has 120 pixels (LEDs).

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.

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 in red. 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.

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 Gecko5 board this can be done using an initial counter value of 2^{10}. You may get a slight difference in this value depending on how you implement the wait loop. 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 the **cs200 extension simulator**, the `wait` procedure can cause the simulation to run too slow. In this case, it is best 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 $0 \leq y \leq 9$ 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 `a0`: 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`

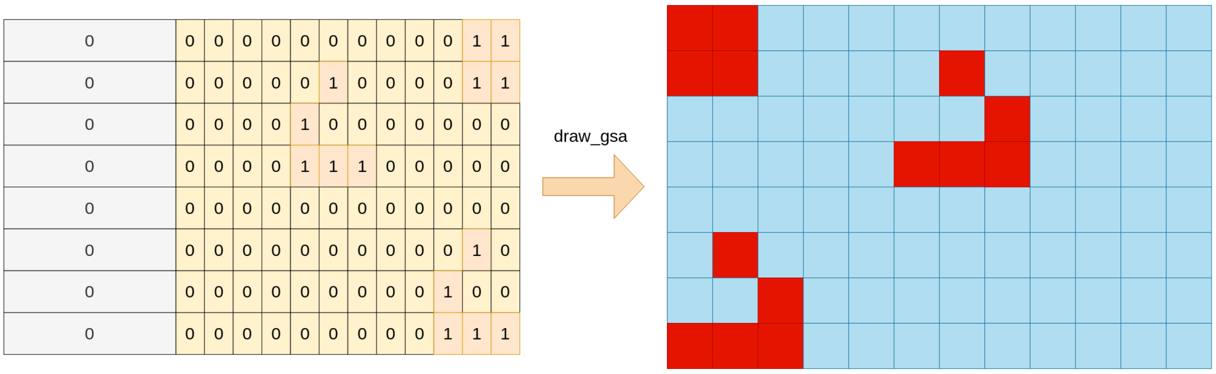
The `draw_gsa` procedure takes the GSA currently in use and reproduce it on the LEDs.

Arguments

- none

Return Value

- None



0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1

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 the **cs200 extension simulator**, you can read a value from the `RANDOM_NUM`. Consequent reads will return the next random value, based on hardcoded seeds. It is therefore expected that the random number generator will return the same sequence of numbers each time the program is run.

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 1` represents the tens and `button 0` 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 b0 pressed, 0 otherwise
- register `a1`: 1 if b1 pressed, 0 otherwise
- register `a2`: 1 if b2 pressed, 0 otherwise

Return Value

- None

3.5.4 Procedure `set_seed`

The `set_seed` procedure uses the input argument (the current Seed ID) to set the current GSA to the predefined seed state associated with it (check the SEEDS section in the template file).

Arguments

- register `a0`: the current seed ID value

Return Value

- None

3.5.5 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` (or the current seed ID is less than the number of seeds), it increments the `Game seed` by one, and copies the new seed in the current GSA. If the game state is `RAND` (or the current seed ID is greater or equal to the number of seeds), 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

3.5.6 Procedure `update_state`

The `update_state` procedure checks if the `BUTTONS` 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`: `BUTTONS`

Return Value

- None

3.5.7 Procedure `select_action`

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

Arguments

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

- register `a0`: 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 `a0`: number of living neighbours.
- register `a1`: 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 `mask` 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.

This procedure should also be in charge of drawing the walls on the screen in blue, in a similar way to the `draw_gsa` procedure.

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 10 push buttons of the Gecko5 are read through the **Buttons** module, which is memory mapped (see Table 1). Its content is described in Table 2.

Address	**31 . . . 10**	**9 . . . 0**
`BUTTONS`	*Reserved*	*State of each button*

Table 2: The first word of the **Buttons** module.

The button register contains the information whether the button i (i = 0, 1, 2, ..., 9) was pressed. If the button i changed its state from released to pressed, i.e. a falling edge was detected, the register 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 the register, regardless of the value written, the entire register will be cleared; there is no possibility to clear its individual bits.

In the **cs200 extension** simulator, you can observe the behavior of buttons module by opening clicking on the buttons and reading from the BUTTONS memory location. For simplification, we renamed the useful buttons as per the following figure (Figure 11). You will find equivalent names in the provided code template as .equ directives.

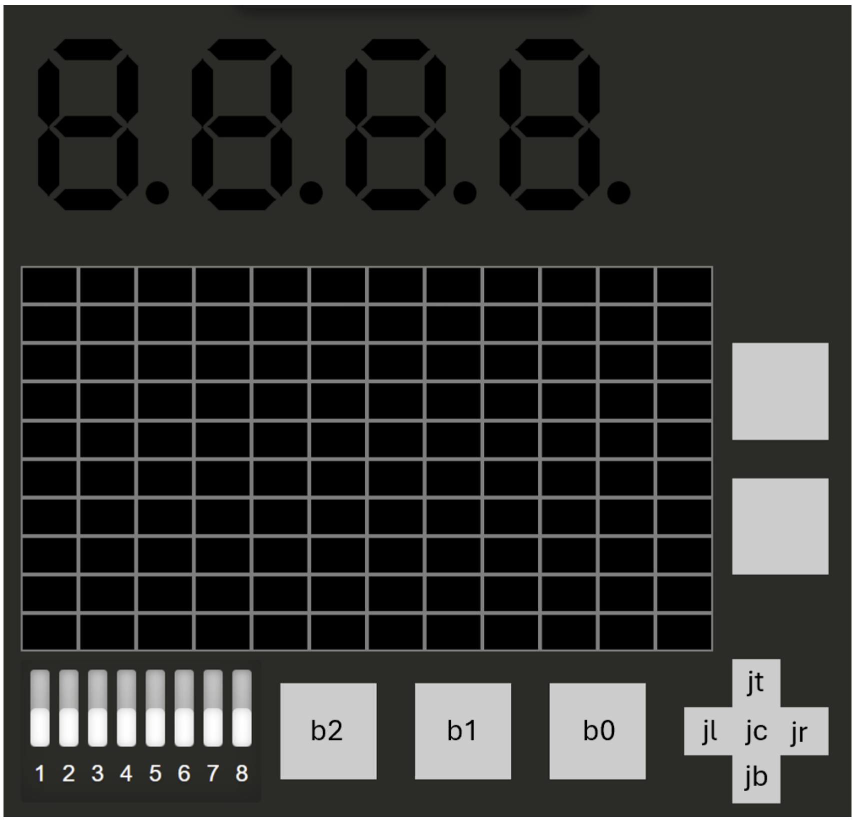


Figure 11: Buttons mapping

3.7.1 Procedure `get_input`

The `get_input` procedure reads the `BUTTONS` register and returns its value.

Arguments

- None

Return Value

- register `a0`: `BUTTONS` register

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

3.8 Game step handling

3.8.1 Procedure `decrement_step`

The current number of remaining steps in the game is display on a 4 digit 7 segment (7-SEG) display, depicted in Figure 12. The digits of the 7-SEG display are exposed through a memory mapped register at address 0x60000000. Each byte of that register configure a different digit. We already give you the font constants, defined in the lab template through the font_data array, needed to properly display digits on the 7-SEG.

7-SEG Control Register		
Name	**Bit Field**	**Description**
Digit 0	[7:0]	Content of digit 0 of the 7-SEG display.
Digit 1	[15:8]	Content of digit 1 of the 7-SEG display.
Digit 2	[23:16]	Content of digit 2 of the 7-SEG display.
Digit 3	[24:31]	Content of digit 3 of the 7-SEG display.

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.

Arguments

- None

Return Value

- register `a0` 1 if done 0 otherwise

```
  /* 7-segment display */
  font_data:
    .word 0x3F /* 0 */
    .word 0x06 /* 1 */
    .word 0x5B /* 2 */
    .word 0x4F /* 3 */
    .word 0x66 /* 4 */
    .word 0x6D /* 5 */
    .word 0x7D /* 6 */
    .word 0x07 /* 7 */
    .word 0x7F /* 8 */
    .word 0x6F /* 9 */
    .word 0x77 /* A */
    .word 0x7C /* B */
    .word 0x39 /* C */
    .word 0x5E /* D */
    .word 0x79 /* E */
    .word 0x71 /* F */

/* Loads font for digit 9 in t0 */
la t0, font_data
lw t0, 36(t0)
```

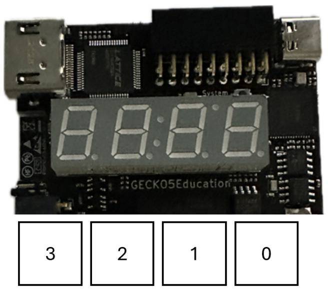


Figure 12: 7SEG mapping

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 and displayed on the leds
4. `GSA ID` is 0
5. The game is currently paused
6. The game speed is 1 (or, `MIN_SPEED`)

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 **cs200 extension** simulator. 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 assembly file. The automatic grader will look for and test the following procedures (all given in the template): `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; the grader will replace everything **around** the tested procedure with the reference code. Therefore, any auxiliary function you may require must be enclosed between the corresponding BEGIN and END (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. You will be able to see the feedback of the preliminary grading as soon as the grader is done processing your submission.
- The final test will assess the correctness of the procedures enlisted above by analyzing their effect on memory contents and registers. For each test, you will have access to an abstract description of the performed test as well as a pass/fail notification. You can submit your solution for the final grading only twice.

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 the extension. You will have one month, after the submission deadline, to come to a lab session and demonstrate that your game works.