# ECSE 324: Computer Organization
# Lab 4: Keyboard and Display

**Due Wednesday, December 3, at 11:59 pm**

**Abstract**

In this lab, you will explore the more complex input/output capabilities of the DE1-SoC: the PS/2 keyboard port, and VGA display. You will begin by writing a driver for each of these devices, and then put them together into an application: the Game of Life. While the drivers will be written in ARMv7 assembly, the game logic can be written in C.

**Summary of Deliverables**

- Source code for:
  - An assembly library implementing a complete VGA driver
  - An assembly library implementing a complete PS/2 driver
  - A C application that implements the Game of Life on the VGA screen using PS/2 input
- Reports, two, no longer than one page each (10 pt font, 1” margins)
  - Part 3a: Performance analysis of Game of Life grid drawing subroutines
  - Part 3b: Performance analysis of Game of Life game state and grid update process

Please submit your source code to **MyCourses** in a single .zip archive, using the following file naming conventions:

- Archive: `StudentID_FullName_Lab4_src.zip`
- Code: `part1.s`, `part2.s`, `part3.c`

Please submit your reports to **Crowdmark** as PDFs, using the following file naming convention:

- Part 3a: `StudentID_FullName_Lab4_report_p3a.pdf`
- Part 3b: `StudentID_FullName_Lab4_report_p3b.pdf`

**Grading Summary**

- 50% Software test cases
- 50% Reports

**Changelog**

- 12-Nov-2025 Initial release.

## Overview

In this lab we will use the high-level I/O capabilities of the DE1-SoC simulator to

1. display pixels and characters using the VGA controller, and
2. accept keyboard input via the PS/2 port.

For each of these topics, we will create a driver. We will test the drivers both individually and in tandem by means of test applications.

## Part 1: Drawing things with VGA

The DE1-SoC computer has a built-in VGA controller that can render pixels, characters or a combination of both. The authoritative resource on these matters is Sections 4.2.1 and 4.2.4 of the DE1-SoC Computer Manual. This section of the lab provides a quick overview that should suffice for the purpose of completing this lab.

To render pixels, the VGA controller continuously reads the pixel buffer, a region in memory starting at `0xc8000000` that contains the color value of every pixel on the screen. Colors are encoded as 16-bit integers that reserve 5 bits for the red channel, 6 bits for the green channel, and 5 bits for the blue channel. That is, every 16-bit color is encoded like so:

| 15 … 11 | 10 … 5 | 4 … 0 |
|---|---|---|
| Red | Green | Blue |

The pixel buffer is 320 pixels wide and 240 pixels high. Individual pixel colors can be accessed at `0xc8000000 | (y << 10) | (x << 1)`, where `x` and `y` are valid *x* and *y* coordinates on the screen (i.e., $0 \le x < 320$, and $0 \le y < 240$).

As previously noted, we can also render characters. To do so, we will use the character buffer, which is analogous to the pixel buffer, but for characters. The device's VGA controller continuously reads the character buffer and renders its contents as characters in a built-in font. The character buffer itself is a buffer of byte-sized ASCII characters at `0xc9000000`. The buffer has a width of 80 characters and a height of 60 characters. An individual character can be accessed at `0xc9000000 | (y << 7) | x`.

*Getting Started: VGA driver*

To provide a slightly higher-level layer over the primitive functionality offered by the pixel and character buffers, we will create a driver, i.e., a set of functions that can be used to control the screen.

To help get you started, we created an application that uses such functions to draw a testing screen. Your job is to create a set of driver functions to support the application. Download vga.s and augment it with the following four functions. Note the arguments and associated data types in the following C prototypes.

```
void VGA_draw_point_ASM(int x, int y, short c);
void VGA_clear_pixelbuff_ASM();
void VGA_write_char_ASM(int x, int y, char c);
void VGA_clear_charbuff_ASM();
```

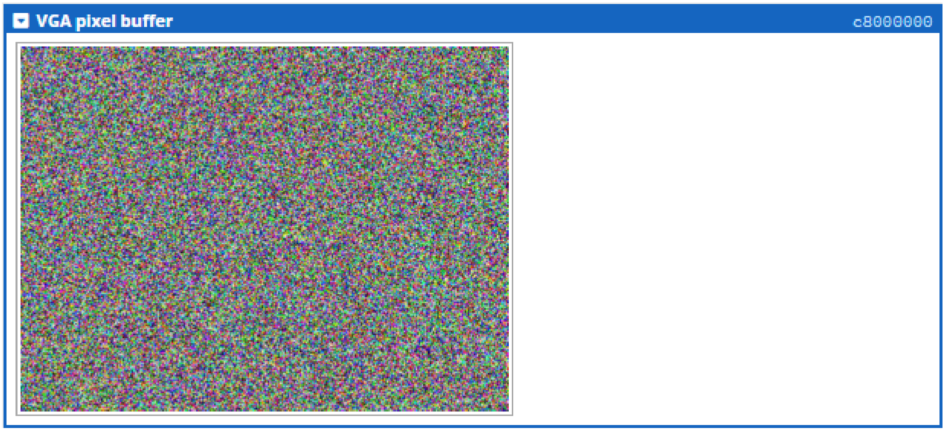
- `VGA_draw_point_ASM` draws a point on the screen at the specified (*x*, *y*) coordinates in the indicated color *c*. The subroutine should check that the coordinates supplied are valid, i.e., *x* in [0, 319] and *y* in [0, 239]. Hint: This subroutine should only access the pixel buffer.
- `VGA_clear_pixelbuff_ASM` clears (sets to 0) all the valid memory locations in the pixel buffer. It takes no arguments and returns nothing. Hint: You can implement this function by calling `VGA_draw_point_ASM` with a color value of zero for every valid location on the screen.
- `VGA_write_char_ASM` writes the ASCII code *c* to the screen at (*x*, *y*). The subroutine should check that the coordinates supplied are valid, i.e., *x* in [0, 79] and *y* in [0, 59]. Hint: This subroutine should only access the character buffer.
- `VGA_clear_charbuff_ASM` clears (sets to 0) all the valid memory locations in the character buffer. It takes no arguments and returns nothing. Hint: You can implement this function by calling `VGA_write_char_ASM` with a character value of zero for every valid location on the screen.

Notes:

- Use suffixes `B` and `H` with the assembly memory access instructions in order to read/modify bytes/half-words in memory.
- You must follow the conventions taught in class. If you do not, then the testing code in the next section will be unlikely to work.

*Testing the VGA driver*

To test your VGA driver, run your finished assembly file. You can inspect the VGA output visually using the VGA pixel buffer tab under the Devices panel of the simulator.




If you implemented your driver correctly, compiling and running the program will draw the following image.

## Part 2. Reading keyboard input

For the purpose of this lab, here's a high level description of the PS/2 keyboard protocol. For a more comprehensive resource, see Section 4.5 (pp. 24-25) of the DE1-SoC Computer Manual.

The PS/2 bus provides data about keystroke events by sending hexadecimal numbers called scan codes, which for this lab will vary from 1-3 bytes in length. When a key on the PS/2 keyboard is pressed, a unique scan code called the make code is sent, and when the key is released, another scan code called the break code is sent. The scan code set used in this lab is summarized by the table below. (Originally taken from Baruch Zoltan Francisc's page on PS/2 scan codes.)

| KEY | MAKE | BREAK | KEY | MAKE | BREAK | KEY | MAKE | BREAK |
|---|---|---|---|---|---|---|---|---|
| A | 1C | F0,1C | 9 | 46 | F0,46 | [ | 54 | F0,54 |
| B | 32 | F0,32 | \` | 0E | F0,0E | INSERT | E0,70 | E0,F0,70 |
| C | 21 | F0,21 | - | 4E | F0,4E | HOME | E0,6C | E0,F0,6C |
| D | 23 | F0,23 | = | 55 | FO,55 | PG UP | E0,7D | E0,F0,7D |
| E | 24 | F0,24 | \ | 5D | F0,5D | DELETE | E0,71 | E0,F0,71 |
| F | 2B | F0,2B | BKSP | 66 | F0,66 | END | E0,69 | E0,F0,69 |
| G | 34 | F0,34 | SPACE | 29 | F0,29 | PG DN | E0,7A | E0,F0,7A |
| H | 33 | F0,33 | TAB | 0D | F0,0D | U ARROW | E0,75 | E0,F0,75 |
| I | 43 | F0,43 | CAPS | 58 | F0,58 | L ARROW | E0,6B | E0,F0,6B |
| J | 3B | F0,3B | L SHFT | 12 | FO,12 | D ARROW | E0,72 | E0,F0,72 |
| K | 42 | F0,42 | L CTRL | 14 | FO,14 | R ARROW | E0,74 | E0,F0,74 |
| L | 4B | F0,4B | L GUI | E0,1F | E0,F0,1F | NUM | 77 | F0,77 |
| M | 3A | F0,3A | L ALT | 11 | F0,11 | KP / | E0,4A | E0,F0,4A |
| N | 31 | F0,31 | R SHFT | 59 | F0,59 | KP * | 7C | F0,7C |
| O | 44 | F0,44 | R CTRL | E0,14 | E0,F0,14 | KP - | 7B | F0,7B |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| P | 4D | F0,4D | R GUI | E0,27 | E0,F0,27 | KP + | 79 | F0,79 |
| Q | 15 | F0,15 | R ALT | E0,11 | E0,F0,11 | KP EN | E0,5A | E0,F0,5A |
| R | 2D | F0,2D | APPS | E0,2F | E0,F0,2F | KP . | 71 | F0,71 |
| S | 1B | F0,1B | ENTER | 5A | F0,5A | KP 0 | 70 | F0,70 |
| T | 2C | F0,2C | ESC | 76 | F0,76 | KP 1 | 69 | F0,69 |
| U | 3C | F0,3C | F1 | 05 | F0,05 | KP 2 | 72 | F0,72 |
| V | 2A | F0,2A | F2 | 06 | F0,06 | KP 3 | 7A | F0,7A |
| W | 1D | F0,1D | F3 | 04 | F0,04 | KP 4 | 6B | F0,6B |
| X | 22 | F0,22 | F4 | 0C | F0,0C | KP 5 | 73 | F0,73 |
| Y | 35 | F0,35 | F5 | 03 | F0,03 | KP 6 | 74 | F0,74 |
| Z | 1A | F0,1A | F6 | 0B | F0,0B | KP 7 | 6C | F0,6C |
| 0 | 45 | F0,45 | F7 | 83 | F0,83 | KP 8 | 75 | F0,75 |
| 1 | 16 | F0,16 | F8 | 0A | F0,0A | KP 9 | 7D | F0,7D |
| 2 | 1E | F0,1E | F9 | 01 | F0,01 | ] | 5B | F0,5B |
| 3 | 26 | F0,26 | F10 | 09 | F0,09 | ; | 4C | F0,4C |
| 4 | 25 | F0,25 | F11 | 78 | F0,78 | ' | 52 | F0,52 |
| 5 | 2E | F0,2E | F12 | 07 | F0,07 | , | 41 | F0,41 |
| 6 | 36 | F0,36 | PRNT SCRN | E0,12, E0,7C | E0,F0,7C,E0,F0,12 | . | 49 | F0,49 |
| 7 | 3D | F0,3D | SCROLL | 7E | F0,7E | / | 4A | F0,4A |
| 8 | 3E | F0,3E | PAUSE | E1,14,77, E1,F0,14, F0,77 | | | | |

Two other parameters involved are the **typematic delay** and the **typematic rate**. When a key is pressed, the corresponding make code is sent, and if the key is held down, the same make code is repeatedly sent at a constant rate after an initial delay. The initial delay ensures that briefly pressing a key will not register as more than one keystroke. The make code will stop being sent only if the key is released or another key is pressed. The initial delay between the first and second make code is called the typematic delay, and the rate at which the make code is sent after this is called the typematic rate. The typematic delay can range from 0.25 seconds to 1.00 second and the typematic rate can range from 2.0 cps (characters per second) to 30.0 cps, with default values of 500 ms and 10.9 cps respectively.

*Getting started: PS/2 driver*

The DE1-SoC receives keyboard input from a memory-mapped PS/2 data register at address `0xff200100`. Said register has an RVALID bit that states whether or not the current contents of the register represent a new value from the keyboard. The `RVALID` bit can be accessed by shifting the data register 15 bits to the right and extracting the lowest bit, i.e., `RVALID = ((*(volatile int *)0xff200100) >> 15) & 0x1`. When `RVALID` is true, the low eight bits of the PS/2 data register correspond to a byte of keyboard data.

The hardware knows when you read a value from the memory-mapped PS/2 data register and will automatically present the next code when you read the data register again, *but this takes time.*

Note: Your function must wait for when `RVALID` is true again before reading subsequent bytes. Your function may work in single-step mode without this (because the simulator, when paused at a breakpoint, does not pause the simulation of I/O devices), but will fail in continuous execution if your function doesn't wait for each byte.

For more details, see Section 4.5 (pp. 24-25) of the DE1-SoC Computer Manual.

Download ps2.s. This assembly file implements a program that reads keystrokes from the keyboard and writes the PS/2 codes to the VGA screen using the character buffer. Copy your VGA driver into it. Then implement a function that adheres to the following specifications. Note the argument, return value, and associated data types in the following C prototype.

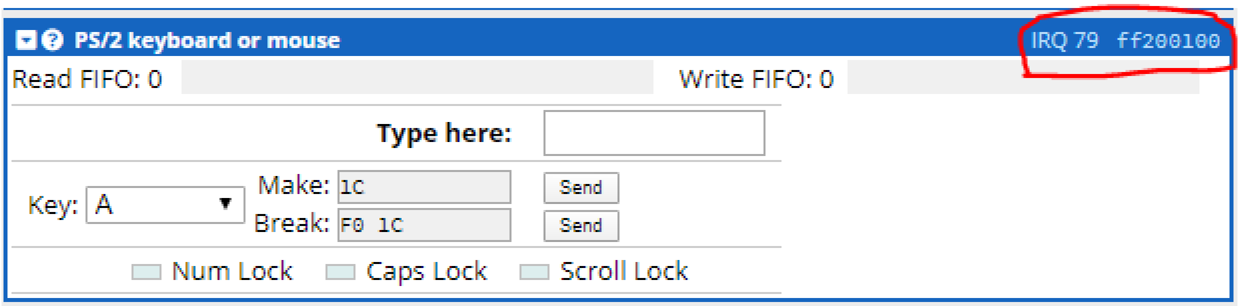
```c
int read_PS2_data_ASM(char *data);
```

- `read_PS2_data_ASM` checks the `RVALID` bit in the PS/2 Data register. If it is valid, then the data should be read, stored at the address `data`, and the subroutine should return 1. If the `RVALID` bit is not set, then the subroutine should return 0.

*Testing the PS/2 driver*

To verify that the PS/2 driver is working correctly, you can type into the simulator's PS/2 keyboard device and verify that the bytes showing up on the screen correspond to the expected codes from the table in this section's introduction.

If you implemented your PS/2 and VGA drivers correctly, then the program will print make and break codes whenever you type in the simulator's keyboard input device. Make sure to use the keyboard device at address `0xff200100`.

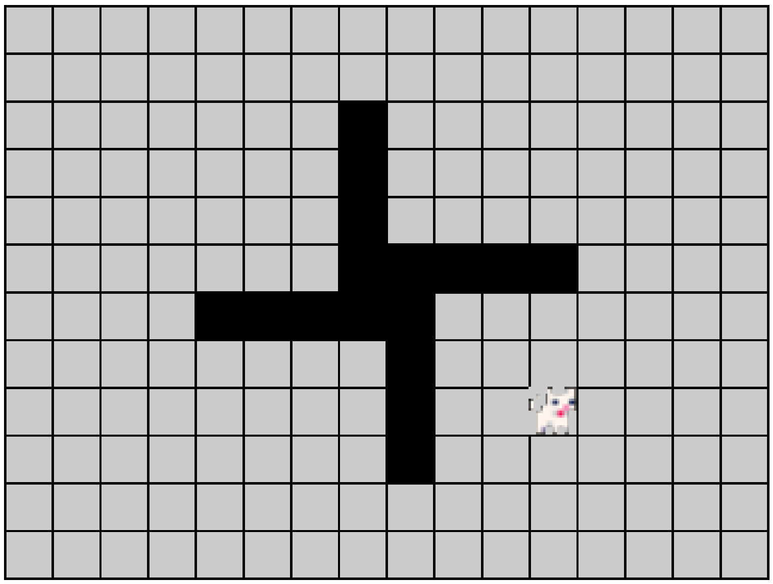



Note: If you did not manage to implement a working VGA driver, then you can still get credit for the PS/2 driver by replacing `write_byte` with the implementation below. It will write PS/2 codes to memory address `0xfff0`. Delete all calls to VGA driver functions and delete the `write_hex_digit` function to ensure that your code still compiles.

```
write_byte:
        push    {r3, r4, lr}
        ldr     r4, =0xfff0
        and     r3, r3, #0xff
        str     r3, [r4]
        pop     {r3, r4, pc}
```

## Part 3. Putting everything together: Game of Life

We will now implement the Game of Life. (Playable here.) We'll use a 16x12 grid, displayed in VGA for the field of play, and use PS/2 input to a) toggle the state of grid locations, and b) update game state.



*A quick note about grading*

Unlike the previous deliverables, this one is a bit more explicitly broken out in terms of grading. The more detailed rubric is in place to clearly indicate the partial credit available for partial solutions. *x%* below indicates that the code associated with a particular subtask is worth *x%* of the total points available in the demo and report. *It is possible to get 70% of the marks for this lab without attempting to implement any GoL game logic.*

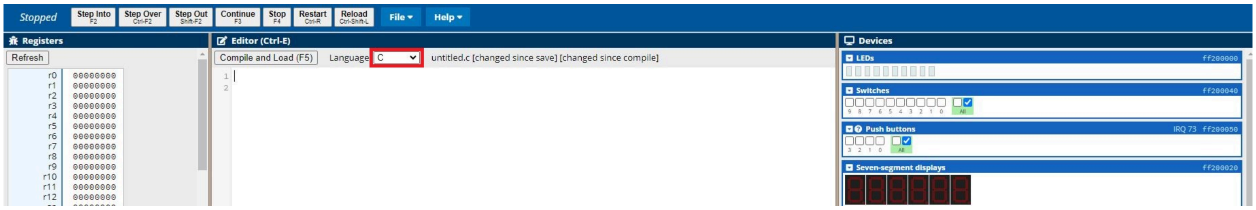
**Please indicate clearly in a dedicated header at the top of your source code what functionality has been implemented and, *as far as you know*, is working correctly.**

*You will write this part of the lab in C*

You may have noticed that the CPUlator interface allows you to write and compile C code. In this part of the lab, you will in-line your assembly-language VGA and PS/2 drivers and write calls to them in C. Alternatively, you are welcome to re-write the drivers in C.

*In-line Assembly*

In-line assembly allows users to combine C language with assembly. To get started, please make sure the language setup is configured as C rather than ARMv7. (ARM-specific documentation)




Also please ensure that you are using the HTTPS protocol when accessing the simulator; otherwise, you will receive a server error when compiling. Here is the HTTPS URL: [https://ecse324.ece.mcgill.ca/simulator/?sys=arm-de1soc](https://ecse324.ece.mcgill.ca/simulator/?sys=arm-de1soc).

Compile and run in CPUlator this an example of using ARM assembly with C:

```c
int foo(int x){
    int y = 0;
    __asm__ __volatile__(
        "add %0, %1, %2"     // %0, %1, and %2 are linked to output and input operands. y = x + 10.
        :"=r"(y)             // Output operands: %0
        :"r"(x), "r"(10)     // Input operands: %1, %2
        :"r1"                // Clobbers
    );
    return y;
}

int main() {
    int x = 0;
    printf("Before inline assembly: x = %d\n", x);
    __asm__ __volatile__(
        "mov r0, %0\n\t"    // \n\t: a newline in assembly; required for each line.
        "bl foo\n\t"        // Call function foo. Note that it automatically uses r0 as the argument register.
        "mov %0, r0"        // r0 is used as the result register.
        :"+r"(x)            // output operands: %0 ('+' instead of '=' here means it can be also used as input.)
    );
    printf("Final result: x = %d\n", x);
    return x;
}
```

Note: The `volatile` keyword is crucial when working with memory-mapped I/O, as it prevents the compiler from optimizing away repeated reads and writes to these essential memory locations.

While using in-line assembly allows complete control over how low-level interactions with the processor, including I/O, are implemented, it is also possible to do so in C. Compile and run this code, which copies switch state to LED state, lighting LEDs in response to switch flips.

```
#define LED_BASE_ADDR       0xff200000
#define SWITCH_BASE_ADDR    0xff200040

int main() {
    // Pointers to memory-mapped I/O for LEDs and switches
    volatile unsigned char *led_base_addr = (volatile unsigned char*)
LED_BASE_ADDR;
    volatile unsigned char *switches_base_addr = (volatile unsigned char*)
SWITCH_BASE_ADDR;

    // Continuously mirror switch states to LEDs
    while (1) {
        *led_base_addr = *switches_base_addr;
    }

    return 0;
}
```

Note: the choice of data type matters! The above code will not update the state of the two most significant LEDs because it is not reading the two most significant switches: it only reads the least significant 8-bit `char`.

*Getting started: drawing lines (10%)*

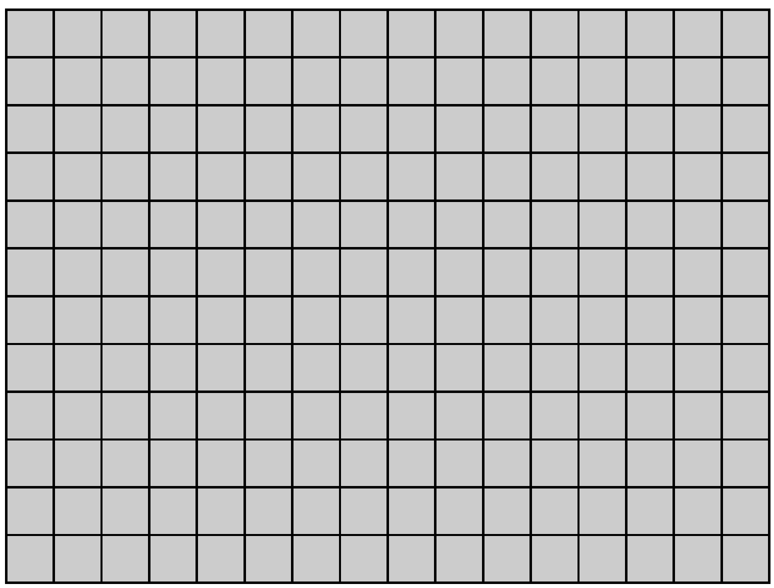
Implementing the game will be easier with subroutines to assist us; it's up to you to define the function prototypes how you see fit.

We'll start by drawing the grid, line by line. Write an assembly subroutine that calls `VGA_draw_point_ASM` repeated to fill pixels in a line from ($x_1$, $y_1$) to ($x_2$, $y_2$), where either $x_1$ = $x_2$ (a vertical line) or $y_1$ = $y_2$ (a horizontal line):

- `VGA_draw_line` draws a line from pixel ($x_1$, $y_1$) to ($x_2$, $y_2$) in color *c*.

Begin writing your Game of Life program by writing and calling a subroutine that draws a 16x12 grid on the VGA display:

- `GoL_draw_grid`, draws a 16x12 grid in color *c*.



***Part 3 Performance Analysis A***

Interacting with the VGA interface requires complex code and many memory accesses. Profile your implementation so far: when calling `GoL_draw_grid`, how many instructions execute, and what fraction of these instructions are data memory accesses? Which functions require the most total time? What would happen, in terms of performance, if the time required to access memory were substantially longer than the emulator assumes?

Note: if you have not completed all of the functionality required, do the most complete analysis you can with the features you have implemented.

Submit a report to Crowdmark, no longer than one page, detailing your measurement approach and results.

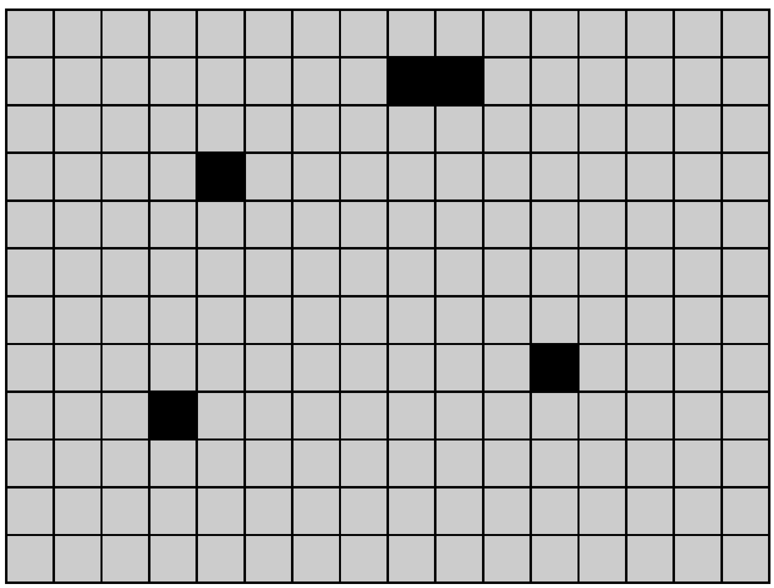
*Next steps: drawing rectangles (10%)*

Now let's draw some rectangles. You have a choice at this point: you can write a new standalone function that calls `VGA_draw_point_ASM`, or one that calls `VGA_draw_line`, drawing rectangles line by line:

- `VGA_draw_rect`, draws a rectangle from pixel $(x_1, y_1)$ to $(x_2, y_2)$ in color *c*.

Put your new subroutine to work by writing a new subroutine that fills in a specified grid location $(x, y)$, $0 \leq x < 16$, $0 \leq y < 12$, with color *c*:

- `GoL_fill_gridxy`, fills the area of grid location $(x, y)$ with color *c*.



*Game logic: initialization (10%)*

The next step is to start writing game logic, beginning with the process of initializing the game environment, the board. The layout of the board will be saved in memory in a 2D structure with dimensions 16x12, just like the grid we've drawn above. The board can be initialized to different things to test game logic; it should be initialized to something interesting in your submission to show off that this code works, that the game logic works, etc. The first example board given does something interesting; there are [many other examples](#) of interesting configurations.

Allocate the board somewhere in memory. We'll use a '1' to indicate an active grid location (it's alive!); a '0' will indicate an inactive location. Note that the board below is defined assuming one word per grid location; some game logic may be easier, or memory use may be more efficient, if shorts or bytes are used instead. The choice is yours.

In ARMv7 assembly, we may define GoLBoard as follows:

```
GoLBoard:
    //  x 0 1 2 3 4 5 6 7 8 9 a b c d e f    y
    .word 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 // 0
    .word 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 // 1
    .word 0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0 // 2
    .word 0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0 // 3
    .word 0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0 // 4
    .word 0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0 // 5
    .word 0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0 // 6
    .word 0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0 // 7
    .word 0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0 // 8
    .word 0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0 // 9
    .word 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 // a
    .word 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 // b
```

In C, we may define it as `int GoLBoard[12][16]`.

Write a function that calls `GoL_fill_gridxy` to fill in the grid locations with 1s in the GoLBoard array:

- `GoL_draw_board` fills grid locations (*x*, *y*), $0 \leq x < 16$, $0 \leq y < 12$ with color *c* if `GoLBoard[y][x] == 1`.

*Game logic: changing the playing field (10%)*

A user may wish to modify the playing field either at the beginning or some later step in the game. Initialize your board to display a cursor of your choice in (0,0). Use polling or interrupts to check for a keypress on WASD. The cursor may be displayed at all times, or displayed only in response to a keypress that moves it. It should not be possible to move the cursor off the field.

- w: move the cursor up (toward lower *y*)
- a: move the cursor left (toward lower *x*)
- s: move the cursor down (toward higher *y*)
- d: move the cursor right (toward higher *x*)

Your cursor can be anything you like that distinguishes it from the (in)active grid locations on the board: a tile of a different color , the dog that lives next door , etc.

Using polling or interrupts, also check for a keypress on the spacebar.

- Spacebar: toggle the state of the grid location where the cursor is located

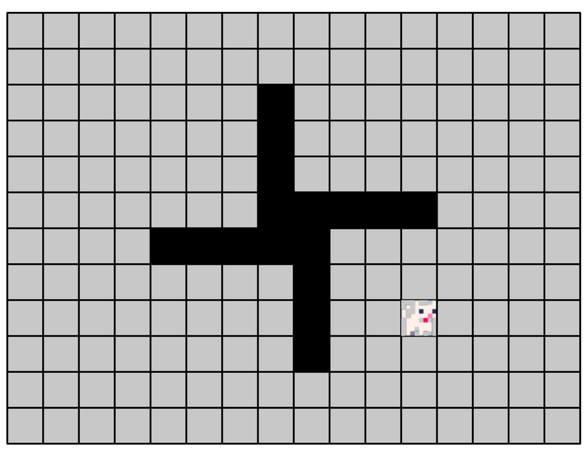
It may be helpful for the cursor to look different if it is on a tile that is active  vs. inactive .
*Game logic: state update (20%)*

Now that all the building blocks are in place, we can implement the core of the game: the rules for updating playing field state.

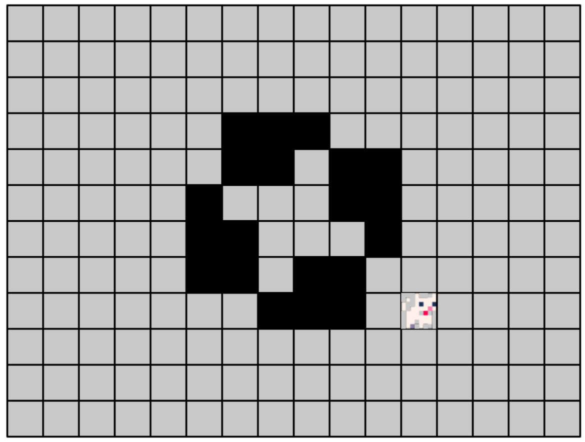
Using polling or interrupts, additionally check for a keypress on n. When n is pressed, iterate across all grid locations in the playing field, and [update their state] based on the state of their 8 neighboring cells (one in each cardinal direction, and another in each diagonal direction).

- Any active cell with 0 or 1 active neighbors becomes inactive.
- Any active cell with 2 or 3 active neighbors remains active.
- Any active cell with 4 or more active neighbors becomes inactive.
- Any inactive cell with exactly 3 active neighbors becomes active.

E.g.: this configuration, after a single press of n and update pass,



becomes this configuration.



A wide variety of approaches are possible for performing the state update loop. E.g., you could

- Define a set of functions that check for each of the (4) update conditions, and iterate across all grid locations in memory, calling all four functions for each. Or,
- Create a second data structure that mirrors the GoL board that keeps track of the number of neighbors each cell has, and in a single pass update the GoL board (and neighbor count state).

***Part 3 Performance Analysis B***

What is the computational cost of a single game state update step, and how is this cost divided across interaction with I/O, game logic, etc? Use breakpoints to measure instructions executed, data memory accesses performed, etc, for the different parts of your game logic loop (e.g, from when a keypress is detected, through to when the grid locations have been redrawn, for instance). Submit a report to Crowdmark, no longer than one page, detailing your measurement approach and results.

**Deliverables**

Your code will be graded manually to test the functionality of your I/O libraries and application.

**Grading**

*Source Code*
- 20% Part 1: VGA driver
- 20% Part 2: PS/2 driver
- 60% Part 3: Game of Life application

Note that multiple tests will be used to evaluate each submitted deliverable; the value of each test case is weighted equally.

*Report*

- 50% Part 3a: Performance analysis of game board display initialization
- 50% Part 3b: Performance analysis of game state update and display sequence

Each section will be graded for: (a) clarity, (b) organization, and (c) technical content:

- 1pt *clarity*: grammar, syntax, word choice
- 1pt *organization*: clear narrative flow from problem description, approach, testing, challenges, etc.
- 3pt *technical content*: appropriate use of terms, description of proposed approach, description of testing and results, etc.

**Submission**

Please submit your source code to MyCourses in a single .zip archive, using the following file naming conventions:

- Archive: `StudentID_FullName_Lab4_src.zip`
- Code: `part1.s`, `part2.s`, `part3.c`

Please submit your reports to Crowdmark as PDFs, no more than one page each (1” margins, 10 pt font), using the following file naming convention:

- Part 3a: `StudentID_FullName_Lab4_report_p3a.pdf`
- Part 3b: `StudentID_FullName_Lab4_report_p3b.pdf`