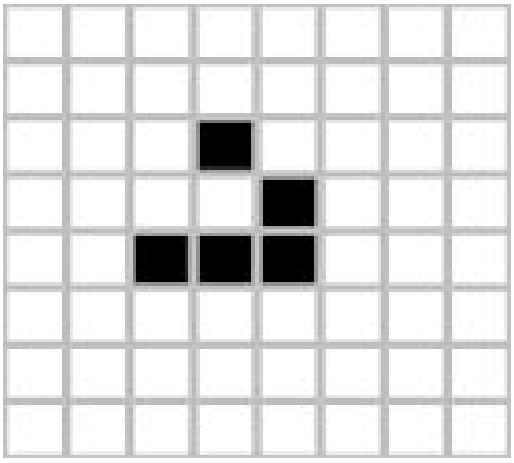
ECE 544 FINAL PROJECT REPORT

JOHN CONWAY'S GAME OF LIFE

(Version 1.0)



BY:

Richa Thakur (rthakur@pdx.edu)

Wesley Chavez(wesley@pdx.edu)

Jeba Farhana(jfarhana@pdx.edu)

March 21,2017

John Conway's Game of Life

Project Description:

John Conway's "Game of Life" is a cellular automaton, developed in 1970, to drastically simplify John von Neumann's mathematical ideas of emergence, self-organization, and replication. "The Game of Life" consists of a two-dimensional grid of size 64 x 64 pixels, with cells that can either be "alive" or "dead". Four rules govern whether a cell is alive, depending on its eight neighbors:

If a cell is "alive", or "populated":

1. If the cell has 0-1 neighbors, it dies, as if by underpopulation.
2. If the cell has 2-3 neighbors, it lives on to the next generation.
3. If the cell has 4 or more neighbors, it dies, as if by overpopulation.

If a cell is "dead" or "unpopulated":

1. If the cell has 3 neighbors, it is populated, as if by reproduction.

Figure 1 shows example computations for four pixels within one frame.

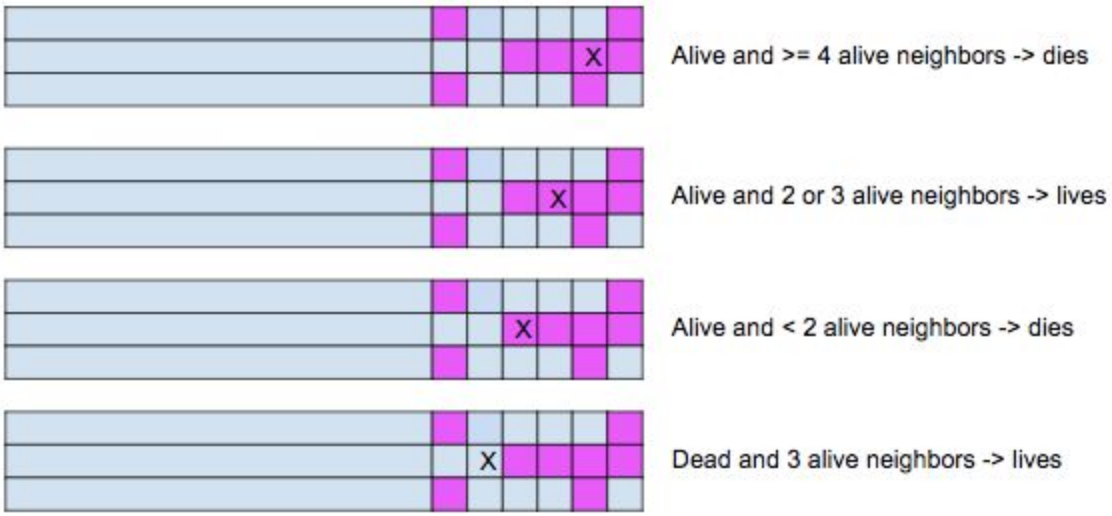


Figure 1: Computation of next state/frame for four pixels

Interesting and complex patterns can evolve from these few rules, depending on the initial pattern in the system.

We have extended the functionality of this **zero-player game**/simulation to a **one-player game**. This includes functionality allowing the user/player to:

1. Choose from a set of hard-coded patterns to initialize the system with, or choose a random initial pattern.
2. Increase or decrease the frame rate/speed of simulation.
3. Pause/Resume the simulation.

4. Manually create or destroy life (toggle the state of user-desired pixels).
5. Reset the simulation.

Overall User Interface:

1) The PmodENC is the rotary encoder peripheral which changes the frame rate/speed of the simulation (Rotary knob turned clockwise increases the frame speed and turned anti-clockwise decreases the frame speed).
2) The PmodOLEDrgb is the display peripheral which, based on switches [15:8], displays the initial pattern selected and the name of the pattern.
3) CPU reset button re-initializes the whole system.
4) Switch[0] is used for pausing and resuming the simulation.
5) The directional push buttons let the user create his/her own pattern when the simulation is in the paused state. The center push button will toggle the state of a cell under the cursor (whose location is determined by the U, D, L, R push buttons).
6) The switches [15:8] is assigned for each initial pattern:
   sw[15] = 10 row cell pattern selected
   sw[14] = Glider pattern selected
   sw[13] = Small Exploder pattern selected
   sw[12] = Exploder pattern selected
   sw[11] = Tumbler pattern selected
   sw[10] = Spaceship pattern selected
   sw[9] = Gosper Glider Gun pattern selected
   sw[8] = Random pattern selected

   [**NOTES: 1) These patterns (except the random pattern) are from Reference 1.
   2) If none of the switches are ON, the user can create his/her own pattern on pause condition i.e. by turning the switch[0] ON and using the pushbuttons to create the pattern.]

Hardware:

We implemented this game on the Digilent Nexys4 FPGA, with an embedded MicroBlaze soft-core processor for ease of programming. Noteworthy system components are shown in Figure 2.

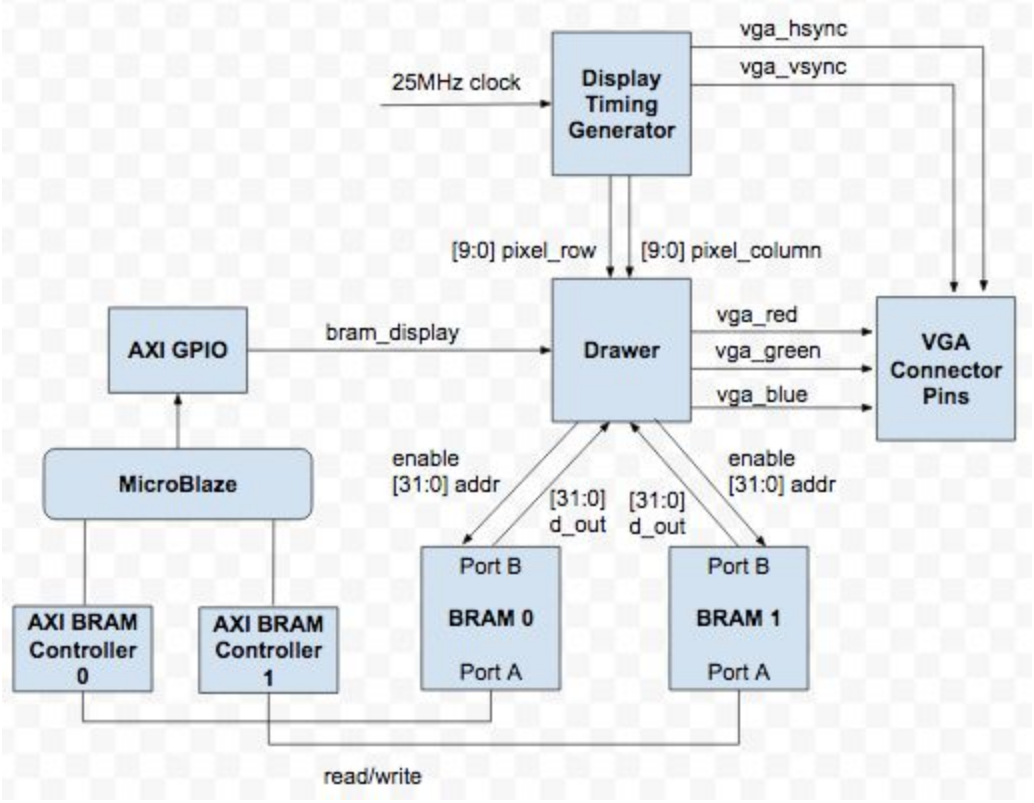


Figure 2: Block Diagram of main hardware components in the system

Block RAM (BRAM 0, BRAM 1):

This is where the simulation patterns are stored. In Figure 3, each bit in BRAM corresponds to a pixel (one of 64 x 64) in the simulation. Since the data is 32-bit, we used lower addresses for the left half of the display (64 x 32), and higher addresses for the right half. For example, a concatenation of the data in address offsets 0 and 260 correspond to the first row of 64 pixels to display. In the Game of Life, a "0" corresponds to a dead pixel, and a "1" corresponds to an alive pixel. These BRAMs are dual-port, with Port A being read and written to by the MicroBlaze (in software), and Port B being read by the Drawer module.

Left half of display		Right half of display	
Address Offset	Bits	Bits	Address Offset
0	31.....0	31.....0	260
4	31.....0	31.....0	264
8	31.....0	31.....0	268
...			...
256	31.....0	31.....0	512

Figure 3: Pattern Storage in a BRAM

The reason we use two BRAMs is to designate one as the current frame (that the Drawer reads from and displays), and the other as the next frame (that the MicroBlaze computes and writes to) in the simulation. Once it is time to display the next frame in the simulation, the roles of the

BRAMs switch, and the “next frame” BRAM is now designated as the current frame and displayed, while the other BRAM gets written to by the MicroBlaze.

Display Timing Generator:
This module sends the appropriate signals (in accordance with VGA timing specifications) to the VGA connector pins, vga_vsync and vga_hsync. It also outputs two 10-bit signals, pixel_row and pixel_column, so that the Drawer knows which pixel is currently being displayed.

Drawer:
This module reads two BRAMs in an alternating fashion (not at the same time), as described earlier, and depending on the row and column that the VGA is currently displaying, outputs a 64x64 cell grid, where each (4x4 pixel) cell is one bit in BRAM. If the bit is a 0, the cell is gray, and if 1, the cell is pink. A black square borders the pixel grid. Since each BRAM address holds 32 bits, and there are 64x64 cells, addresses are designated as in Figure 3. The input from GPIO, bram_display, chooses which BRAM to read from.

MicroBlaze:
The Microblaze takes care of reading and writing to/from the appropriate memory locations using the respective AXI BRAM controllers and all the functionalities implemented for the working of the simulation through the use of desired IPs included in our embedded system.

Software:

It is a Xilinx-based design that simulates the mathematical theory of John Conway and the application includes several of the key functions provided by the Xilkernel. The application (gameoflife.c, board support package, and xilkernel OS) sets up two independent threads and a master thread. Their functionalities are described as below:

a) master_thread: This thread is responsible for setting up the microblaze, initializing the peripherals, creating the other threads (child threads), and initializes the semaphore that is used to lock the thread whenever the switch inputs are changing. It also registers the interrupts and then enters a main loop.

b) Pattern_Creation_Thread: This thread contains only a few lines, but is where higher-level functions that make up the beef of the application are called:

```
while(1)
{
        sem_wait(&cond_sema);
        pmodENC_read_count(&pmodENC_inst, &RotaryCnt);
        if (RotaryCnt < 50)
        {
                high_fit_count = (u32)(5000 - (RotaryCnt*100));
        }
```

```
        sw = NX4IO_getSwitches();
        if ((sw & 1) == 1)
        {
                pauseSim();
        }
        calcNextFrame();
        if (fit_flag == 1)
        {
                toggleBramDisplay();
                fit_flag = 0;
        }
        sem_post(&cond_sema);
}
```

For comments, look to gameoflife.c. This thread reads the rotary count and sets the frame rate accordingly, reads the switches and pauses the simulation if switch 0 is ON, calculates the next frame, and displays this next frame in toggleBramDisplay() if fit_flag says it's time.

c) Pattern Selection thread: This thread selects the base pattern based on the status of the switches. It acquires lock on the critical section only when the status of the switches are changed. Once the lock is acquired, it checks generates the base pattern and also displays the selection in the OLEDrgb. Once the base pattern is selected and generated, this thread releases the lock so that pattern creation thread can create the nextframe.

The other helper functions included in the application are:

a) pauseSim(): This function pauses the simulation if switch 0 is ON. While switch 0 is ON, it lets the user to create and destroy life or make pattern of their own to start the simulation and see the future generation of the created pattern using the directional push buttons. To do this, first we need to check which BRAM is designated as the current frame, with the volatile bram_display. It goes through the while loop, and if a left or right, button is pressed, it checks to see if x_pos has moved across the boundary of low and high addresses (x_pos = 32) in order to account for border conditions. The tricky part of this is making sure that we write a 1 to the location of the cursor, and remember the previous pattern (bram_read_val), in case the center button is not pressed at that position. If the center button is pressed, we have a new bram_read_val. If up or down are pressed, we need to write bram_read_val to the current memory location, read a new location with a new bram_read_val, and write a 1 to the current position of the cursor. Button left and right changes the X coordinates and button up and down changes the Y coordinates. Button center toggles the state of the selected cell on the grid between alive or dead.

b) clearSim(): This function clears the whole grid/board and all the addresses from both BRAMs. It is called at startup time, and when a new pattern is to be displayed.

c) calcNextFrame(): This function writes to next frame that is to other BRAM based on the current frame (read from current frame). It reads from one BRAM (current frame being displayed, based on the volatile bram_display) and writes to other BRAM i.e. next frame that we are computing. There are 4 for loops in this function. The first nested loop, for (i = 0; i < 2; i++), runs twice and computes the next frame low and high addresses (left and right side of the display). The second nested loop, for (j = 1; j < 63; j++), computes the next frame for a row. Rows a, b, and c are read, and row b is computed based on its immediate neighbors in rows a, b, and c. After row b is computed for the next frame, it is written into the BRAM designated as next frame. The third nested loop, for (k = 1; k < 31; k++), computes the next frame for a given pixel in that row by following the rules of the Game of Life (determining current state (0 or 1, dead or alive) and number of neighbors). After a row is computed, we only need to read the next row, not three new memory locations by updating the rows with:

```
a = b;
b = c;
c  = XBram_ReadReg(read_base_address,i*256+(j+2)*4);
```

The fourth for loop, for (j = 1; j < 63; j++), is a little tricky, and computes the pixel states on the boundary of the low and high addresses (the middle of the display). That means instead of reading rows a, b, and c, we need to read rows a, b, c, d, e, and f, as follows:

```
A  D
B  E
C  F
```

We read these six rows in order to compute the next state for the two pixels on the border in rows b and e. Since these are computed after the non-boundary pixels, we need to read the rows in the BRAM designated as next frame that we just computed (in the nested loops) to update the border pixels and re-write with the updated bit (at the edge of the address associated with rows b and e) into the next frame BRAM. These rows previously computed are b_write and e_write. We move down the rows similarly to the non-boundary pixels, except we have 8 registers to update (a, b, c, d, e, f, b_write, e_write).

d) FIT handler(): This is an 5KHz fixed interval timer interrupt handler which runs every time and the moment that fit_count reaches high_fit_count, the fit_flag is set to 1 and fit_count resets. The FIT raises the flag (fit_flag = 1) it is the time to swap the BRAMs roles and display the next frame. Basically, the rotary encoder is clipped to a value of 50, and the volatile high_fit_count is determined in Pattern_Creation_Thread() from:

```
if (RotaryCnt < 50)
{
        high_fit_count = (u32)(5000 - (RotaryCnt*100));
}
```

This ensures that the higher the rotary count, the lower the high_fit_count, and the quicker we display a new frame.

e) toggleBramDisplay(): The current frame is informed to both hardware and software through the gpio_out port whose bit 0 defines whether it is stored in BRAM 0 or BRAM 1. This value gets written to the GPIO, and volatile bram_display toggles, letting the rest of the application know which frame is the current, and which is the next. This function only gets called if the fit_flag is raised (when we want to display the next frame).

Work Distribution:

Wesley Chavez

1) Hardware creation (mainly drawer.v and BRAM setup)
2) Next frame calculation and BRAM switching (calcNextFrame(), toggleBRAMDisplay(), FIT_Handler())
3) Ability to create and destroy life (pauseSim())

Richa Thakur

1) Xilkernel threading (master_thread(), Pattern_Selection_Thread(), Pattern_Creation_Thread())
2) Pattern initialization (within Pattern_Selection_Thread())
3) PmodOLED display (within Pattern_Selection_Thread())

Jeba Farhana

1) UI logic
2) Audio testing (Not proposed or integrated, but implemented with sine wave data)
3) Random pattern initialization (within Pattern_Selection_Thread())

Challenges Faced:

1) Reading/Writing from/to BRAMs in both hardware and the software.
2) The correct calculation of the next frame initially had bugs.
3) Moving the grid or board size from 32X32 to 64X64 increased the complexity in terms of calculating the boundary conditions for the next frame simulation and also for creating/destroying life.
4) Integrating audio in the project. (not integrated in the project since audio produced was plain sine wave).
5) Implementing the application using Xilkernel was difficult as during multi-threading we were facing racing conditions which was slowing down the maximum frame speed of our simulation.

References:

1) https://bitstorm.org/gameoflife/
2) https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
3) Project release documentation by Prof Dan Hammerstorm and Chetan.
4) ECE 544 Lecture notes by Prof Dan Hammerstorm.
5) ECE 540 Lecture notes by Prof Roy Kravitz.
6) dtg.v by John Lynch & Roy Kravitz