# *Conway’s Game Of Life On FPGA*

## Mojo V2 FPGA Development Board

Abstract

The main aim of this project was to implement Conway’s game of life (a cellular automata) on a grid of 16x16 led matrices. This was later upgraded to work on a VGA display.

We have used Mojo V2 FPGA Development board; our design uses its parallelization and is hence fast.

We tried to make it interactive but it was not implemented due to technical difficulties. The game of life is much more interesting if it’s interactive.

Introduction

The Game of Life is not just a typical computer game. It is a 'cellular automaton'. Cellular automata are discrete, abstract computational systems that have proved useful both as general models of complexity and as more specific representations of non-linear dynamics in a variety of scientific fields.

The Game of Life is a simulation of a collection of cells which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game.

Conway’s game of life is a zero player game and is governed by simple rules. The real challenge to us was implementation using FPGA.

FPGA (Field Programmable Gate Arrays), unlike microcontrollers, do nothing by themselves and the entire hardware has to be coded. This also the power of FPGA

Motivation behind the Project

FPGAs are capable of ‘parallel processing’ which means that they can simultaneously perform several operations which otherwise have to be performed sequentially. We have exploited this property of FPGA to manipulate the next stage every cell in our game according to its neighbourhood cells simultaneously. As clear from the description, the execution this way is considerably faster than one-by-one method. This factor will be noticeably significant once the setup is scaled up. This makes it ideal to study cellular automata as the processes necessarily don’t occur sequentially.

Background Theory

The game of life by John Conway runs on simple rules as follows:

1. A live cell with fewer than two live neighbours dies, as if caused by under-population.
2. A live cell with two or three live neighbours lives on to the next generation.
3. A live cell with more than three live neighbours dies, as if by overcrowding.
4. A dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

A prerequisite for working on FPGA was to learn a HDL-Hardware Descriptive Language. For FPGA one can code in Verilog or VHDL. We learnt the former. Each set of commands in Verilog is executed simultaneously as mentioned earlier for parallel processing.

Implementation Details

**Hardware:**

The FPGA development board we used is Mojo V2. The Mojo features include

* Spartan 6 XC6SLX9 FPGA
* 84 digital IO pins
* 8 analog inputs
* 8 general purpose LEDs
* 1 reset button
* 1 LED to show when the FPGA is correctly configured
* On board voltage regulation that can handle 4.8V - 12V
* A microcontroller (ATmega16U4) used for configuring the FPGA, USB communications, and reading the analog pins
* On board flash memory to store the FPGA configuration file

Mojo V2 FPGA Development Board

Source: embeddedmicro.com

For the first stage of implementation i.e. simulation of the game of life on a 16X16 LED matrix we used four 8X8 LED matrices shorted appropriately to effectively work as a 16X16. Each LED matrix we used was a 26 pin matrix 10 pins of which are not required for lighting it up. Following is the pin description of the LED matrix used:



In order to light up an LED, the column-pin(C) corresponding to it has to be given high and the row-pin has to be grounded. For e.g.: for LED(3, 4) we have to set R =0, C4=1and other row-pins have to be 1 and column-pins 0.

These pins have to be connected to the I/O pins of the Mojo board. There are some pins which are by default connected to the elements like the Reset button and internal LEDs in the mojo board itself and this has to be taken care of.

For the second stage of the project i.e. simulation of game of life using VGA display: A VGA monitor requires 5 digital signals to display a picture:

* R, G and B (red, green and blue signals).
* HS and VS (horizontal and vertical synchronization)

 All we needed to do is get a monitor for VGA display and a female VGA connector and connect the 15 pins in the following manner:



Source: fpga4fun.com/PongGame

* Pins 13 and 14 of the VGA connector (HS and VS) are digital signals so can be driven directly from two FPGA pins.
* Pins 1, 2 and 3 (R, G and B) are 75Ω analog signals with nominal values of 0.7V. With 3.3V FPGA outputs, use three 270Ω series resistors. The resistors form voltage dividers with the 75Ω resistors in the monitor inputs so that 3.3V become 3.3\*75/(270+75)=0.72V, which is close to 0.7V
* Pins 5, 6, 7, 8 and 10 are ground pins.

**Software Platform**:

We have used ISE(**I**ntegrated **S**oftware **E**nvironment) design suite 14.7 by Xilinx**Xilinx ISE**(**I**ntegrated **S**oftware **E**nvironment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. One can simulate the hardware using iSim. However the coding in iSim does not work on FPGA. Testbenches /Test-fixtures are meant for simulation part only.

We descried small blocks of hardware named modules. Inputs and outputs of each module are needed to be specified in the beginning and way it processes the inputs and outputs are needed to be specified after that. For more on syntax and rules see section on code. A module once created can be instantiated in any other module keeping in mind the hardware feasibility.

**Code for Version 1:**

Initially the team wrote small modules to light internal/ external LEDs with internal/external switches and lighting a single LED in the LED matrix.

**Following is the entire code used for this version of the game of life:**

In the main module, in which all modules are instantiated and from where input/outputs from external hardware of the set-up are taken, we have three modules instantiated, namely ‘pin’, ‘copy’ and ‘light’ in the following manner:

pin p(

 .clk(c1),

 .arr(arr),

 .next(next),

 .rst(rst)

);

copy c(

 .rst(rst),

 .clk(clk),

 .arr(arr),

 .next(next)

);

light pov(

 .clk(clk),

 .test(next),

 .R(R),

 .C(C)

);

Here R and C are the arrays of I/O pins in mojo board to which row and column pins of the LED matrix (16X16) are connected. R and C are defined as outputs from the mojo-top. We had to configure these pins in the mojo.ucf file; mojo.ucf is where we configure the I/O pins of the mojo board with some pins configured by default to internal elements of the board.

The default clock of the board (clk) is a 50 MHz clock which is too fast to have a noticeable updating of present state of the cell to the next. C1 is a 1 Hz clock which has been slowed down using the following module:

module count (clk, b);

 input clk; // synthesis attribute PERIOD clk "50 MHz"

 reg [25:0] count = 0;

 output reg b = 0; // one pulse per second

 always @ (posedge clk) begin

 b <= (count == 50000000 - 2);

 count <= b ? 0 : count + 1;

 end

endmodule

For reference to any LED in the LED matrix we used a 256 bit array. The LED (i, j) is referred by arr[i\*8+j]. The other 256 bit array, next is the updated state, i.e. the next state of our cellular grid. The module ‘copy’ assigns the values in next to corresponding cells in arr.

The module ‘light’ is for POV. Any state of the game of life is being displayed using Persistence of Vision. This is the module which sends output to R and C pins. This module takes as input a 256 bit array and depending on the state of each LED lights up each of them one by one at a frequency of 50 MHz.

The module ‘pin’ takes as input the array arr, i.e. the present state of the LED matrix and its function is to give the next state of the matrix in form of output next following the rules of the game of life. In other words, according to the present state of the neighbouring cells of the cell under consideration, it assigns the next state of the cell in the array next. For checking the rules, we simply add the states (1-live, 0-dead) of the neighbours and compare. For e.g. if arr[x] is live and the neighbours add upto 5, we assign next[x] to be a dead cell.

The matrix is refreshed to its initial seeding each time the reset button on the mojo board is pressed. As mentioned earlier, the updating of the state occurs at a frequency of 1 Hz.

All these modules run parallel to give fast execution. If summarized, the ‘pin’ module initializes and updates the states. The ‘light’ module implements POV for displaying the current state. The module ‘copy’ copies the array next onto array arr.

**Code for Version 2:**

This version of Conway’s game of life gets displayed on a VGA screen. This is not as elegantly simple as version-1.

The modules for the slowed down 1Hz clock (module: timer) and for copying (module: copy) remain the same. For horizontal and vertical synchronization pulse generator we are using two modules, namely counter and VGA. CounterX counts 768 values (from 0 to 767) and CounterY counts 512 values (0 to 511). The two modules are:

Module counter:

module counter(input clk, output CounterX, output CounterY

 );

reg [9:0] CounterX;

reg [9:0] CounterY;

wire CounterXmaxed = (CounterX==767);

always @(posedge clk)

if(CounterXmaxed)

 CounterX <= 0;

else

 CounterX <= CounterX + 1;

always @(posedge clk)

if(CounterXmaxed)

 CounterY <= CounterY + 1;

endmodule

Module vga:

module vga(input clk, input [9:0] CounterX, input [9:0] CounterY, output vga\_HS, output vga\_VS

 );

reg vga\_HS, vga\_VS;

always @(posedge clk)

begin

 vga\_HS <= (CounterX[9:4]==0); // active for 96 clocks

 vga\_VS <= (CounterY==0); // active for 2 clocks

end

endmodule

To define cells for the game we have hardcoded the blocks in this way:

block[0] = ((CounterX>=200&&CounterX<220)&&(CounterY>=200&&CounterY<220))

The entire pin module has been shifted to mojo-top, so the updating of the grid occurs at the same block of hardware as the display output.

In order to make a color be displayed in the block we defined, we assign the block signal to the color signal output in this way: R = block[x];

**Block Diagram:**

Out to VGA Monitor

**Female VGA Connector**

 **16X16 LED Matrix**

 **Mojo V2**

Limitations and Compromises

There are some limitations to the latest version of the project:

* The code for initial seeding needs to be changed whenever a different initial seed is needed.
* The code is not robust at present. For scaling up the display, the code has to be changed significantly, especially when it comes to VGA display which mainly involves hard coding.
* We haven’t been able to implement user interfacing to the game till now. Once the game is set on, user can not change the state of a desired cell.

Future Prospects and Scope

* User interfacing can be added to the game. This would mean the user can set the initial seeding and further user can access any cell and change its state in run-time of the game.
* There can be more than two states associated with each cell corresponding to growth and aging using a color gradient instead of just one color.
* Another idea is to simulate spread of diseases and mutation at cellular level.
* There are several more applications of FPGAs. One of the ideas we were considering is making a genetic algorithm solver using FPGAs.
* FPGAs can be used in bots making their execution faster due to parallel processing.
* To emphasize on the scope of FPGA: it can be used to design any circuit be it a simple AND gate or a microprocessor.

References

* <http://embeddedmicro.com/tutorials/mojo/>
* <https://www.youtube.com/watch?v=pkJAWpkaiHg>
* <http://www.asic-world.com/>
* <http://www.fpga4fun.com/PongGame.html>

Team: **E\_Trix**

Team Members:

**Avani Samdariya,**

**Krati Agrawal,**

**Samyak Jain, and**

**Sanjari Srivastava**

Acknowledgements:

**Arjun Bhasin**

**Avi Singh (Mentor)**

**Kevin Jose**

**Piyush Awasthi**