COMPUTATIONAL EVOLUTION OF FPGA-BASED ADAPTIVE HARDWARE

Jesse Cureton and Dr. Mihail Cutitaru
Department of Electrical and Computer Engineering
Missouri University of Science and Technology
Rolla, MO, USA

ABSTRACT

This paper discusses an attempted implementation of an evolutionary hardware system that can generate mathematically-optimal digital logic circuits based on predetermined evolution criteria. Hardware is implemented using custom evolutionary cells designed in VHDL and run on a Xilinx Artix-7 FPGA development board, with an integrated microprocessor for oversight. Evolutionary computations are performed on a PC connected to the FPGA board via a UART protocol.

1. INTRODUCTION

Field programmable gate arrays (FPGAs) provide an adaptable hardware solution that can be reprogrammed into a new circuit without requiring any physical hardware changes. This allows a designed to modify the functionality of a design, even after it has been produced. With hardware that can be changed, it is possible to perform iterative circuit design, improving the product with each consecutive generation and testing cycle. This is essentially how circuits are designed by humans – iterating and testing through a design and eliminating bugs at each step. When we add these FPGAs to the design process, one can imagine a scenario where if a circuit or can be mathematically described, it can be combined with concepts from computational biology and natural selection simulations to run simulations of many different circuits on the FPGA at a high rate, and computationally evolve the best possible circuit for the task given the available hardware.

In his 1996 paper, “An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics”, Dr. Adrian Thompson [1] did just that, implementing a frequency discriminator based on evolved analog feedback loops in an early Xilinx XC6200 FPGA. Since Dr. Thompson’s work, however, the field has plateaued. This is primarily due to changes in FPGA architecture to address security concerns and remove the ability to change the configuration of the chip on-the-fly.

The initial scope of this project was to design and implement both the FPGA circuitry and the PC-side software to program and test it. On the FPGA side, this entailed the following, each of which will be described in detail in subsequent sections:

- Design of individual evolutionary “cells”

- Design of an overarching “logic block” integrating many cells into an evolvable environment
- Integration of an embedded microprocessor to handle communication with the host PC and logic block
- Design of a UART communication protocol for programming the logic block

On the PC side, the requirements consisted of:

- Implementation of the UART communication protocol for programming the evolutionary circuit
- Evolutionary algorithm implementation that parses output from hardware testing and then generates the next individual for testing
- Visualization software that displays the currently-programmed circuit for analysis

A high-level overview of this system can be seen in **Figure 1** below.

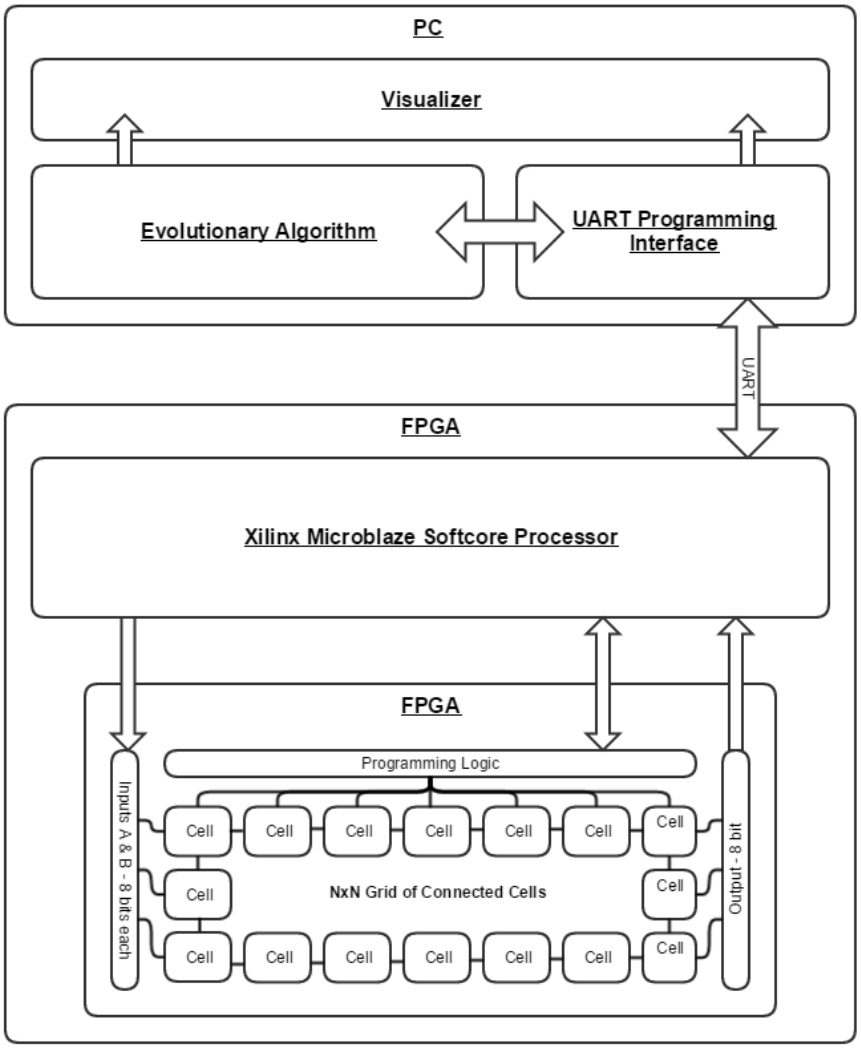

Figure 1 - High Level Architecture Diagram

The rest of this paper will focus on architecture decisions, information learned, and the minutiae of the segments implemented, along with the implementation plan for those that weren't successfully completed.

2. FPGA IMPLEMENTATION

2.1 EHW CELL IMPLEMENTATION

The real core of the project's ability to be modified on the fly and evolve lies in the implementation of the evolutionary hardware cell. As noted in Section 1, one of the primary causes of the stagnation in this field has been the removal of the ability to reprogram most consumer FPGAs while they are running. To overcome this, we implement what amounts to an abstraction layer on the basic FPGA fabric, adding our own configurable logic gates that can be configured on the fly. Figure 2 below shows a block-diagram description of each EHW cell.
Each cell implements bidirectional IO at each of the north, east, west, and south (NEWS) edges. Each NEWS output has an output multiplexer allowing the output to be the signal received at any other NEWS input, or the output of the cell's logic function, *x*. The logic function of each cell has two inputs, both selectable between any of the NEWS inputs, and the output of the logic function can be selected as any a logical NAND, AND, OR, or NOT operation between the two inputs.

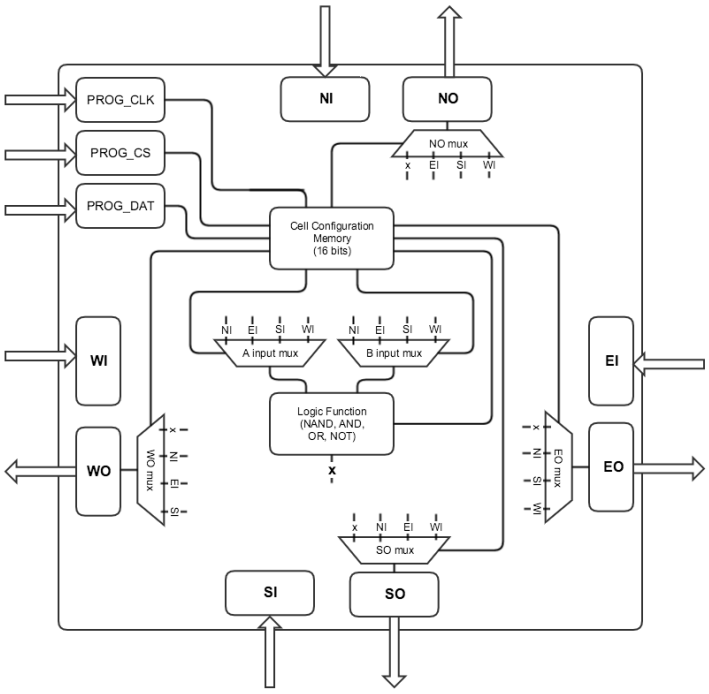

Figure 2 - EHW Cell Block Diagram

All configurable functions inside a cell are controlled by the 16-bit cell configuration memory, which is implemented internally as 16 series D-flip flops forming a serial shift register, with parallel outputs of each bit running to the configuration inputs of all muxes and the logic function. At initialization, the configuration memory must be loaded serially via the programming interface

on the PROG_* pins. This programming interface will be detailed in the following logic block detail section.

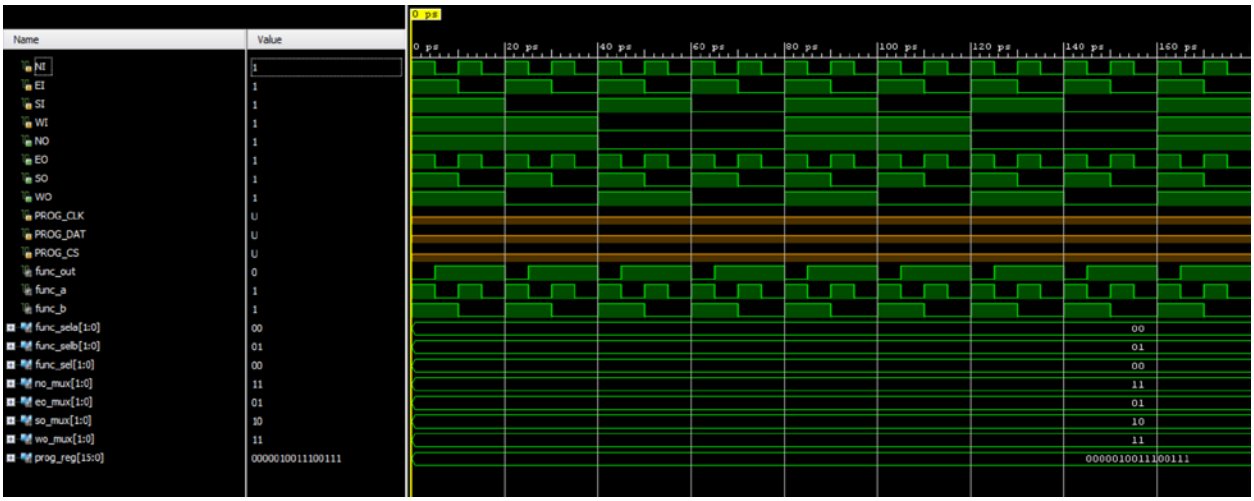

Figure 3 - EHW Cell Logic Simulation

If we examine the circuit simulation in Figure 3 above, we can prove the functionality of each cell. Each of the NEWS inputs is fed a simple square wave with a different period on each input. The **o_mux[1:0]* bits configure each of the output muxes to output the signal 90 degrees counter clockwise from it – i.e. the WO output contains the signal present at SI, since south is 90 degrees counterclockwise to west.

The *func_sel*[1:0]* bits select which signal is present at each logic function input *a* and *b*, in this case input *a* is NI, and input *b* is EI. The *func_sel[1:0]* bits then configure the logic function output to be a logical NAND of *a* and *b*. This a *NAND b* output is seen correctly on *func_out*.

2.2 LOGIC BLOCK IMPLEMENTATION

One layer up from the EHW cells lives the logic block (LB) implementation. The LB creates a 16x16 grid of cells, connects all of their NEWS edges to one another, formalizes two 8-bit input values at the top left of the grid and one 8-bit output value at the top right, and handles the programming logic for configuring as many cells as possible at once from the microprocessor that lives alongside the LB, and will be detailed later.

Figure 4 shows the effective connections of the entire logic block. Each NEWS pair is connected to its immediate neighbor, the 16 cells in the leftmost column have their west input connected to a bit in two 8-bit inputs from the microcontroller, the upper 8 cells in the rightmost column have their east outputs connected to a bit in an 8-bit output value to be read in by the microcontroller, all cells in each column are connected to a common PROG_CS line for column select during programming, and all cells in each row are connected to a PROG_DAT line for programming data. Each cell is then also connected to one single PROG_CLK line for the entire array.

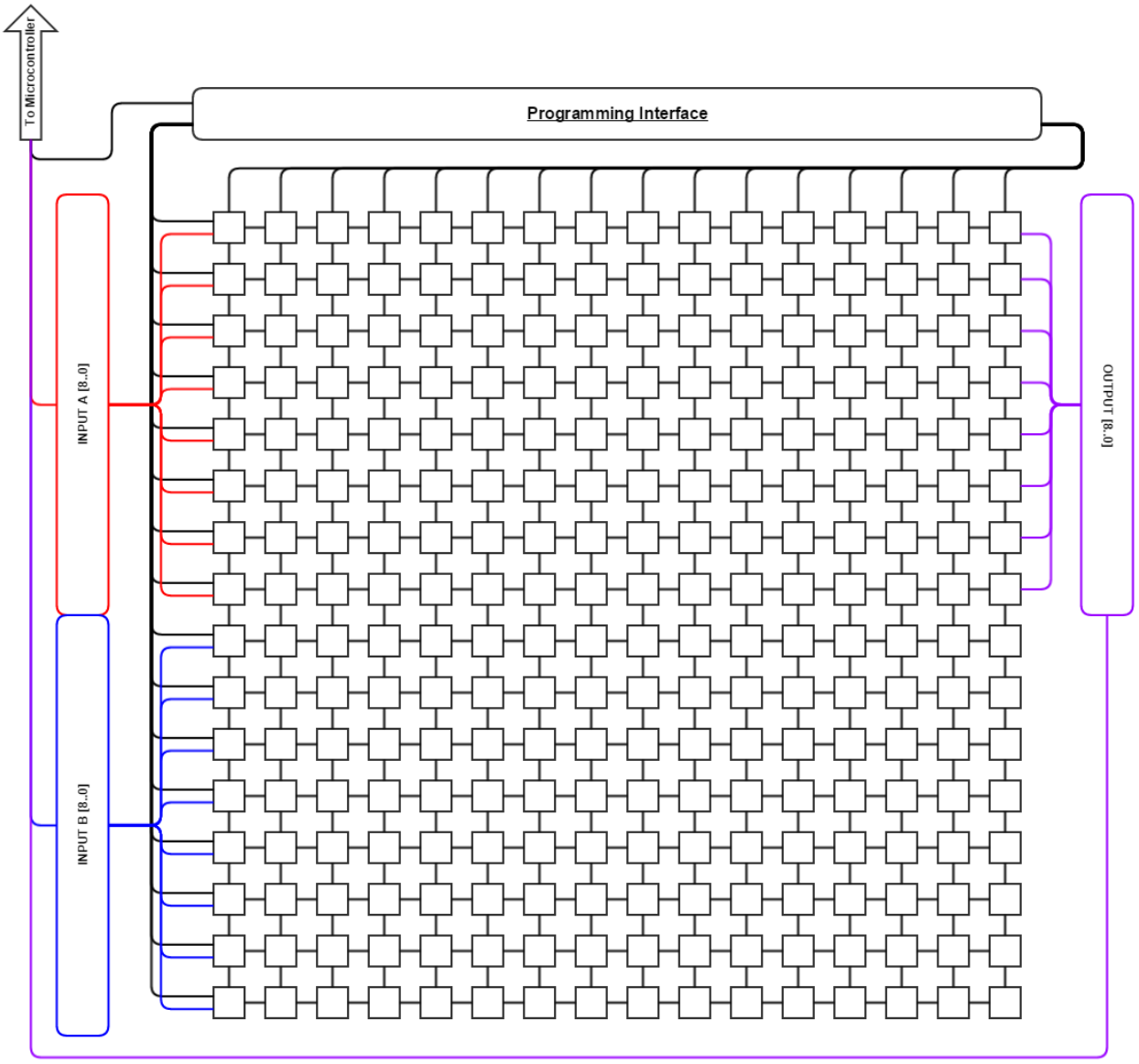

Figure 4 - Logic Block Functional Diagram

This means that in practice, one configures the entire logic block by programming one column at a time. The 16 individual PROG_DAT lines send 16 bits of data to one cell in the column whose column select pin is currently high. All cells in the block receive the clock signal concurrently, but only those whose column select line is high act upon the clock and data being sent.

One interesting note is that due to the fixed placement of the global input and output bytes for the array, the evolutionary solutions must also account for these fixed IO positions. This is similar to what would be seen in a real-world implementation of an evolvable circuit where the algorithm must properly account for fixed hardware limitations.

2.3 SIMULATED MICROCONTROLLER CONFIGURATION

The final portion of the FPGA hardware is a microcontroller to interface with the PC-side over UART, send this data to the programming interface, and parse and return data from the LB to the PC. In order to have the entire FPGA system run on one piece of hardware, the choice of a simulated microprocessor, or "softcore" was the only real option available.

Many different softcore architectures exist, everything from Intel 8051-based, to high end ARM processors, to custom FPGA manufacturer designs. The requirements for this project meant that whatever processor was used had to be able to drive 56 outputs, and read 8 inputs, along with have an integrated UART peripheral, timer support for generating accurate clocks, and if possible, interrupts.

8051 designs lack these high-level features, and ARM cores are more difficult to implement and have higher overhead on the software development side. This left the option of Xilinx's custom Microblaze architecture – which implements all of the features listed above, with some room to grow.

3. PC SOFTWARE

Unfortunately, the project was not able to progress past the work completed in Section 2.3 due to timeline constraints. What follows in this section details the planned architecture for the evolutionary algorithms and the desktop software that would interface with the FPGA hardware.

3.1 EVOLUTIONARY ALGORITHMS

There are three primary evolution strategies used in evolvable hardware implementations [2]: canonic genetic algorithms (elitist evolution), $\mu+\lambda$ evolution, and Cartesian genetic programming.

As noted in Section 1 most early implementations used elitist evolution, as in the Thompson study. These elitist strategies copied over one or more of the fittest individuals from the population at each evolutionary step, and then used them as parents for the entirety of the next generation of individuals [1]. This eliminates crucial genetic diversity in a population, making the evolutionary process much more dependent on random mutation than if all individuals have a chance of producing offspring.

$\mu+\lambda$ evolution takes some steps to counter this. It copies over the most fit individuals, uses them as parents to generate a new subset of individuals, and also copies over portions of the previous population. This maintains genetic diversity, and helps prevent optimizing out needed genetic material from an otherwise nonoptimal solution.

Cartesian genetic programming (CGP) represents a significant departure from these elitist-style algorithms, and is instead based heavily in graph theory. By representing complex graph relationships as a string of integers, the function of nodes in a graph, connections between nodes, and all other graph parameters can be modified from an easily-evolvable string that represents the

genotype of the individual solution [3]. An astute reader will note that the logic block detailed in Section 2.2 can be easily represented as a graph, and the 16-bit programming string at each cell – which handles signal routing between cells, cell function, and cell configuration – could be concatenated to form a genotype for the entire logic block, fitting into the CGP model perfectly.

3.2 PROPOSED EVOLUTION IMPLEMENTATION

The proposed implementation is a slightly modified CGP algorithm. The logic block and its configuration bits will be represented as a graph in the desktop software. The desktop software will perform the evolution computations for each generation, and then generate a configuration bitstream that is 4096 bits long. This will be sent over the UART to the softcore, which will coordinate programming the cells with these values.

Once cells are programmed, the softcore will put known 8-bit values at the input of the circuit, and record the output. All three of these values will then be sent over the UART back to the desktop implementation, which will perform a fitness calculation that was not determined, and then record this fitness value for each individual in the generated population.

The genotype of the logic block is made up of 16*16*16 bits, or 16 bits of memory for each of the 256 cells. During recombination for two parents, each of the 256 cells can be seen as one "gene," with its 16 bits of configuration information being divided up into its respective functions controlling the internal multiplexers. The internal multiplexers represent the lowest level of granularity at the recombination level – that is, when two parent individuals produce an offspring, for each internal configuration mux there will be a 50/50 chance that the offspring's configuration mux values come from either parent. There will be no implementation of dominant or recessive alleles. Following recombination to produce a new offspring, each individual be subject to a chance of random mutation at each bit in the configuration string, helping foster genetic diversity.

Parent selection will be probabilistic based on fitness scores. To avoid eliminating potentially-important genetic information contained in an otherwise non-optimal individual, all individuals will have a chance of producing offspring. After fitness calculation for each individual, their fitness scores will be normalized to create a percent-chance of producing offspring. All individuals will then be added to a "mating pool" where for n offspring desired in the resulting population, $2*n$ parents will be selected from the mating pool at random, based off of their reproduction chances.

The desktop side software will be implemented in Processing, which is a Java based language built around graphical visualization and data processing. It also features easy-to-implement UART/Serial port access, critical for programming the FPGA board. The software will consist of the graph visualizer for the logic block, the evolutionary computation logic, and the programming logic.

4. CONCLUSIONS

This paper has detailed a modern implementation of an evolvable hardware circuit on a new Xilinx Artix-7 FPGA development board. It has also detailed a planned implementation of an evolutionary algorithm and the software to control the hardware implemented onboard.

Despite hardware implementation progressing as planned, the project hit a stopping point before the desktop software could be written. As such, it was not able to come to completion or produce any valid results as far as evolutionary circuits are concerned.

The project, however, accomplish its primary goal at the onset. A successful design and implementation of the circuits for the reprogrammable logic was created and tested on modern FPGA hardware that did not natively support many of the features the early designs were built on.

References

[1] A. Thompson, "An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics," in *ICES '96 Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware*, London, 1996.

[2] F. Cancare, S. Bhandari, D. B. Bartolini, M. Carminati and M. D. Santambrogio, "A bird's eye view of FPGA-based Evolvable Hardware," in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, San Diego, CA, 2011.

[3] J. Miller, "Cartesian Genetic Programming," [Online]. Available: cartesiangp.co.uk.