Intrinsic Evolvable Hardware Implementation of a Robust Biological Development Model for Digital Systems

Heng Liu, Julian F. Miller, Andy M. Tyrrell
Department of Electronics Department, University of York
Heslington York YO10 5DD, UK
{hl142, jfm, amt}@ohm.york.ac.uk

Abstract

An intrinsic evolvable hardware platform was realized to accelerate the evolutionary search process of a biologically inspired developmental model targeted at off-shelf FPGA implementation. The model has the capability of exhibiting very large transient fault-tolerance. The evolved circuits make up a digital "organism" from identical cells which only differ in internal states. Organisms implementing a 2-bit multiplier were evolved that can"recover" from almost any kinds of transient faults. This paper focuses on the design concerns and details of the evolvable hardware system, including the digital organism/cell and the intrinsic FPGA-based evolvable hardware platform.

1 Introduction

Multi-cellular organisms are the most advanced type of creatures which have evolved over millions of years of evolution. They possess several intrinsic characteristics electronic engineers earnestly long for, in particular, growth and fault-tolerance. These features are achieved by means of identical cells, all of which are developed from one special cell (zygote). The entire process, called development, is controlled by the interaction of cells rather than by a centralized process (such decentralized systems are also of interest to engineers).

1.1 Background of Development Principles

The development of an embryo is determined by genes, which control where, when and how many proteins are synthesized [1]. Complex interactions between various proteins and between proteins and genes within cells and hence interactions between cells are set up by activities of genes. These interactions control how the embryo develops.

Biological development involves several key aspects: cell division, emergence of pattern, change in form, cell differentiation and growth. Cell differentiation emerges as a result of differences in gene activities which lead to the synthesis of different proteins. As development is progressive, the fate of cells becomes determined at different times. Inductive interactions by means of chemicals or proteins between cells can make cells different from each other and the response to these inductive signals depends on the state of this cell. Patterning can involve the interpretation of positional information and lateral inhibition.

1.2 Fault-tolerant Techniques

Fault-tolerance is a technique applied to the implementation of systems to ensure their reliability. With the complexity of systems increasing dramatically, fault tolerant techniques become more and more important.

Built-in redundancies and error handling capabilities are the most widely used conventional fault-tolerant technologies. Redundancies can be employed either spatially or temporally. Spatial (area) redundancy can be employed using Dual Modular Redundancy or Triple Modular Redundancy, both of which are based on the majority vote of individual modules. In temporal (time) redundancy techniques, after an error output is detected, it is recomputed in an attempt to recover from the transient fault. Although time redundancy in general requires fewer resources than area redundancy, it demands error handling capability which will incontrovertibly increase the complexity of the system and its design cost. In addition, it is difficult to design such an error handling circuit to store adequate information for recovery so that it can deal with most transient faults.

Transient faults account most system failures [2], so at present we only concentrate on this kind of fault.

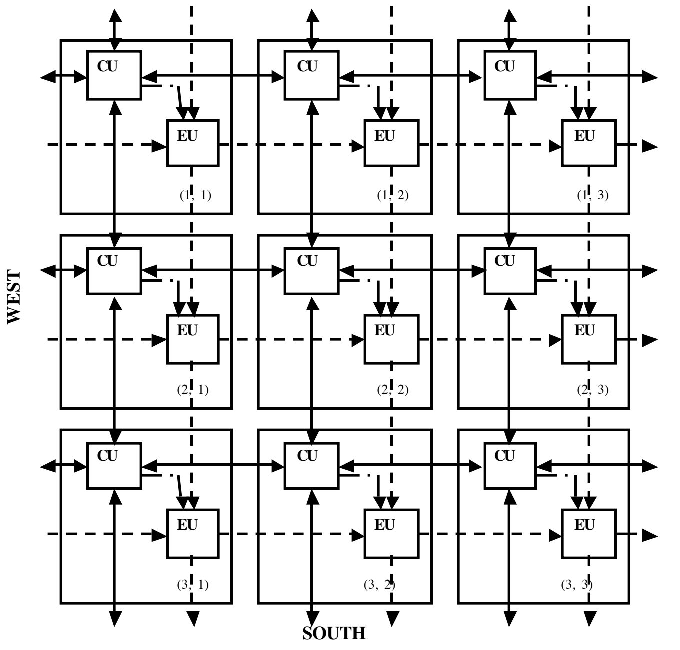
Development has inspired several hardware research projects in the past [3, 4, 5]. However, in this paper a new development-inspired technique will be considered that makes use of chemical signals which grants the system high tolerance to transient faults.

The structure of this paper is as follows: section 2 details the model proposed and its fault tolerant capabilities exhibited in the software simulation; the following two sections will describe the digital organism/cells and intrinsic evolvable platform implementation of the 2-bit multiplier problem respectively. Conclusions will be drawn in the final section.

2 Development Cellular Model for Digital System and its Fault-tolerant Feature

One of the most fundamental features of a development is the universal cell structure: each of the cells in a multi-cellular organism contains the entire genetic material, the genome.

2.1 Cell Structure and Inter-Cell Connections



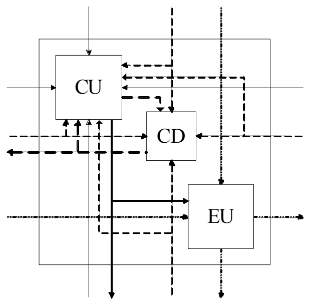
LEGENDS:

CU	Control Unit	EU	Execution Unit
—·▶	EU Function Selection	◀━▶	States & Chemical Signals
━━	Cell border	- -▶	Executing Signals

Fig. 1. Inter-connection of Cells

In the model proposed, shown in Fig. 1, every cell only has direct access to the information of its four adjacent cells: No direct long distance interaction between non-adjoining cells is permitted.

In the digital hardware model (as shown in Fig. 1), the internal structure of digital cells is shown in Fig. 2. A digital cell is composed of three main components: Control Unit (CU), Execution Unit (EU) and Chemical Diffusion module (CD).



LEGENDS:

CU	Control Unit	CD	Chemical Diffusion Module
EU	Execution Unit	——▶	State Signal
- - -▶	Chemical Signal	·-··-·▶	Executing Signal

Fig. 2. Digital Cell Structure

The Control Unit (CU) has a States Register, which stores the internal states of the cell, including the cell state (type) and chemicals. Each CU connects to its 4 immediate neighbors (shown in Fig. 1) and a Next States & Chemical Generator determines its own next state/chemicals according to the current states and chemicals of the neighbors, its own state and its own chemical (illustrated in Fig. 2). The NSCG contains two components: Next States Generator (NSG) and Next Chemical Generator (NCG), both of which are built from combinational circuits.

The EU Function Selection signal (the state of a cell) is 2-bit wide: 0 denotes a dead cell, in this case the EU will simply propagate its west (left) inputs to its south and east neighbors. All other cell types denote otherwise this is a living cells for which the EU will execute and propagate its *calculated* output to the south and east.

The Execution Unit (EU) is the circuit incorporated to do the real calculation of the target application. The inputs to each EU come from its immediate west and north neighbors, and the state of this cell (refer to Fig. 2). Every EU also propagates its output (Executing Signals) to its immediate north and east neighbors. The Execution Unit Core (EUC) is the evolvable core logic circuit, which determines how to process the input signals in the EU.

At present only combinational applications are considered, hence the EU is a combinational circuit. The state and chemical signals are 2-bit and 4-bit wide respectively, while the width of Executing Signal is 3-bit. Both the internal core logical structures of EU (EUC) and CU (NSG and NCG) are determined through evolution. So that the genotype encodes the EU and CU internal structures. The representation of the internal structure of the

EU and CU are based on Cartesian Genetic Programming [7] (CGP): a program is expressed as an indexed graph which is encoded in a linear string of integers. So the genotype just contains a list of node connections and functions.

2.2 Chemical Diffusion

The Chemical Diffusion module (CD) mimics aspects of the real environment in which biological organisms live. In principle, CD should not be a component of a digital cell. However, this hardware design decision makes it more convenient practically, so it is merged into the cell internal structure.

The chemical signal is introduced to transmit information between cells. Another function of the chemical is to serve as a resource which is required for a dead cell to transform to a living one.

Previous experiments [6, 8] suggest that chemicals are indispensable in order to achieve robust solutions: without chemicals, evolved individuals have poor stability and much lower fitness. The chemical diffusion regulation is the key mechanism which makes it such a significant aspect of this model: cells have a means to send long-distance messages.

The chemical diffusion rule employed in this work is similar to that in [6], except that there are only 4 immediate neighbors in this case. So the rule is:

$$(C_{ij})_{t+1} = \frac{1}{2}(C_{ij})_t + \frac{1}{8}\sum_{k,l\in N}(C_{kl})_t \qquad (\mathbf{1})$$

Let N denote all the 4 immediate neighbors of a cell at (i, j) with neighboring position (k, l), the chemical at this position at the new time step is given by (1). The meaning of this equation is that each cell retains half of its previous chemical and distributes the other half equally to its four adjacent cells and receives the diffused chemical from them. It is evident the rule makes sure that chemicals are conserved (apart from the unavoidable loss when the level falls below one) in the diffusion procedure.

The main task of the Chemical Diffusion module (CD) (in Fig. 2) is to calculate the diffused chemical based on the chemicals from the four immediate neighbors and the cell's own chemical value. The CD also propagates the calculated value to the four adjacent cells.

2.3 Digital Organism Growth

Given a genotype, the inner-structure of the cells is determined and a zygote is place at the centre of an 'artificial environment' with x rows and y columns of cells. Initially apart from the zygote cell all cells are dead (in state 0). The position of the zygote was selected to speed up the growth: it takes least time for the digital organism to "cover" the entire area if the zygote is arranged in the centre. The inputs to the cells on the border of this environment are fixed to 0. In our model cells require the presence of chemicals to live. This means that initially some chemicals must be injected at the position of the zygote.

Given a genotype, the growth procedure is described as follows:

1. Initialize chemical and the zygote;
2. Chemical diffusion;
3. All cells update their state simultaneously:
4. If no chemical at a position or all the cell's four neighbors and itself are dead, then this cell's internal program will not be executed;
5. Otherwise, it executes the program that is encoded by the genotype, to generate its next time chemical and state based on current states and chemicals;
6. If next state generated is alive, then overwrite chemical at this position with its own generated chemical;
7. Otherwise, do not touch the chemical at this position;
8. Unless stopping criterion reached go to 2.

This model used in this paper was inspired by software simulation of the 'French Flag problem' [8]. In this work, a 63 (9x7)-cell sized French flag was the intended final pattern. 2 bits were used to represent the states of cells. 0 represented a (dead) cell without any colors (gray in pictures); 1, 2, 3 were blue, white and red cells respectively (all stem cells). One 8-bit wide chemical was used. Since the function of the organism was defined as being coloured like a French flag there was no EU. It was shown that some evolved organisms could recover automatically from many faults.

In initial work we developed the software model described in this paper, and found it possible to evolve some extremely robust individuals, one of which was able to recover completely from *any* transient damage.

After this initial experiment, EUs were incorported and we applied it to more practical application. The first such application chosen was 2-bit multiplier, because of its simplicity.

The task was to evolve a cell circuit that would grow to become a 3x3 cells organism that implements a 2-bit multiplier. The inputs to this multiplier were connected to execution signals of cell (1, 1) and (2, 1), while the output execution signals of cell (2, 3) and (3, 3) drove the ouput result of this multiplier.

The next two sections present the implementation concerns and details of the 2-bit multiplier development model.

3 Digital Organism and Cells Implementation

The core of the IEHW is the fitness evaluation module. Digital cells, each of which contains one or more evolvable sub logic circuits, are the building block of the digital organism.

3.1 Evolvable Molecule

The molecules are the most fundamental elements of the evolvable components of the model. Each evolvable sub-circuit is composed of several molecules. Molecules are the fundamental gates that make up the genome of the cell.

In order to save resources, the genotype is stored centrally in registers outside of the digital organism. Each molecule inputs, including the input selection signals and function selection signal (*I1*, *I2*, *I3* and *Func* in Fig. 3.), are connected to its corresponding bits in the genotype. The *Data* input pin connects to all the input data available to this molecule, which may include inputs to the entire circuit or the outputs of the molecules on the left of this one.

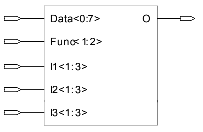


Fig. 3. Molecule Interface

Due to limited resources in the intended hardware platform, the width of the *data* signal is 8, so each input selection signal is 3 bit wide.

It was suggested that 3 input Universal-logic-module – in our case a 2-1 multiplexer (an n-input ULM is logic function which is capable of implementing any function with n-1 independent binary input variables [9]). Although other higher order ULMs could have been chosen we found the MUX fits the average fan-in requirement of human-design circuits most appropriately and if larger fan-in cells are deployed, wiring density/complexity and wiring channel proportion will increase considerably [9]. The MUX is identified in [9] as the most appropriate candidate for general-purpose fundamental logic element. The MUX defines a three variable output function *f*

$$f = y_1 y_2 + \bar{y}_1 y_3$$

This configuration, along with negation of the input variables and constant 1 and 0, can realize all 2 input functions.

Based on this theory, the available functions in a molecule are limited to 4 types of multiplexers (the same as in [6]), which are shown in Table 1: these algebraic expressions are all the 4 possible combination of negation of the input variables, so 2 bits are sufficient for *func* signal.

Name	Algebraic Expression
MUX1(A, B, C)	$AC + B\overline{C}$
MUX2(A, B, C)	$\overline{A}C + B\overline{C}$
MUX3(A, B, C)	$AC + \overline{B}\overline{C}$
MUX4(A, B, C)	$\overline{A}C + \overline{B}\overline{C}$

Table 1. Available functions for Molecules

The available inputs to a molecule in the hardware implementation are constrained. As a result, no constant 1 or 0 is available as input to molecules. However, 1 and 0 can be achieved directly and efficiently by MUX3(*X*, *X*, *X*) and MUX2(*X*, *X*, *X*) respectively, in which *X* refers to one input variable to a molecule.

I1, *I2* and *I3* operate three 8-to-1 multiplexer to select the inputs from the 8-bit width input *Data*. The selected three inputs are then fed to each of the four types of MUX (MUX1 to MUX4). One of the four outputs of the MUXs will be selected by the *Func* input as the final output *O* of this molecule.

This infrastructure of molecule has most similarity with the fundamental multiplexer-based architecture of Xilinx FPGA, but with less flexibility, which is a trade-off to eliminates the possibility of evolving any configuration leading to permanent hardware damages.

3.2 Digital Organism and Cell Interface

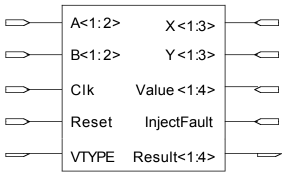


Fig. 4. Digital Organism External Interface

The external interface of the digital organism is shown in Fig. 4. Pin *A*, *B* and *Result* are the inputs and output of the 2-bit multiplier. *Clk* is the global clock signal; if the *Reset* pin is high, all the internal registers will be set to their initial values. All the remaining pins are dedicated to injecting transient fault(s) into the digital organism: when *InjectFault* pin is high, *Value* will be written into the chemical of cell at coordinate (*X*, *Y*) if *VTYPE* is low, otherwise the lowest 2-bit of *Value* is written into the state of the cell. Meanwhile the whole organism stops its growth process.

Every cell has an identical structure. Fig. 5 demonstrates the external interface of a digital cell. Pin *InjectFault*, *VTYPE* and *Value* are connected to their global counterparts. If this cell is at the coordinate (X, Y) and *InjectFault* is active (high), the *CS* pin of this cell will be driven to high and the cell will overwrite its own chemical or state with *Value*.

A "growth step" lasts two clock cycles: at the falling edge of the first clock cycle a live cell (its state is not 0) will replace the existing chemical with its generated one; at the rising edge of the second clock, the chemicals diffuse according to the rule described in section 2. At the rising edge of the first clock cycle in the next "growth step", the state will be updated.

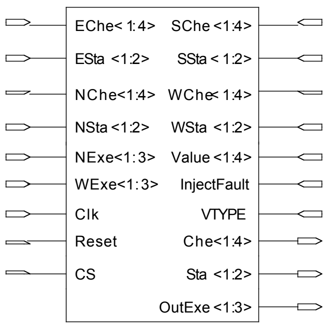


Fig. 5. Digital Cell External Interface

3.3 Evolution Strategy

In the work presented here we used a software simulation to evolve the desired structures. In order to simplify the implementation, two phases are designed to evolve the entire digital organism:

1. Evolve the structure of the EUC and the distribution of states of the 3x3 cells. The genotype in this phase consists of two parts: the CGP part encodes the EUC structure; the other is the states for all the 9 cells. The fitness is the correct bits of the multiplier result output: it has two 2-bit inputs, so there is totally 2^2 x 2^2 = 16 possible combination of inputs. Since the output of a 2-bit multiplier is 4-bit wide, 16 x 4 = 64 is the maximum fitness for the organism in a given step.

2. Structures of NCG and NSG will be evolved to discover a stable solution for the states distribution of cells found in the first phase. This phase is the same as the evolution process described in French Flag problem except for some parameter values [6].

One of the patterns found in the first phase is shown in Table 2. This pattern utilizes most available cells with a diverse and complete distribution of states, so it is chosen as the target configuration of the digital organism along with its corresponding EUC structure obtained via evolution.

0	1	3
2	3	2
3	2	3

Table 2. Chosen Cell State Pattern

In order to search for more resource-saving individuals, once a perfect solution was found, it would receive a fitness bonus based on how many molecules it exactly used: the fewer the better.

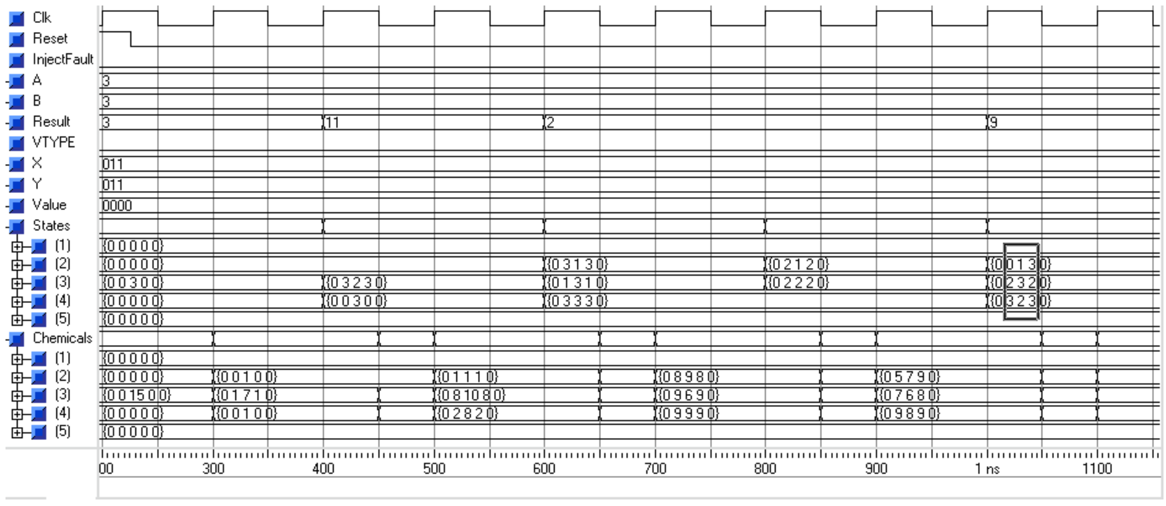


Fig. 6. Developmental Growth Procedure (the white rectangle circle the mature pattern)

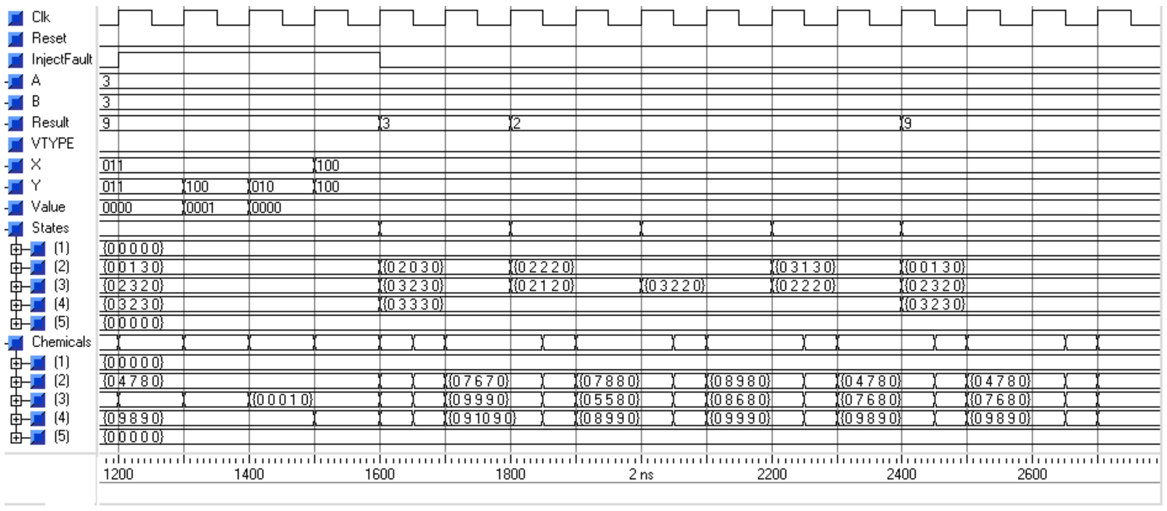


Fig. 7. Injection of the first set of faults and the recovery procedure

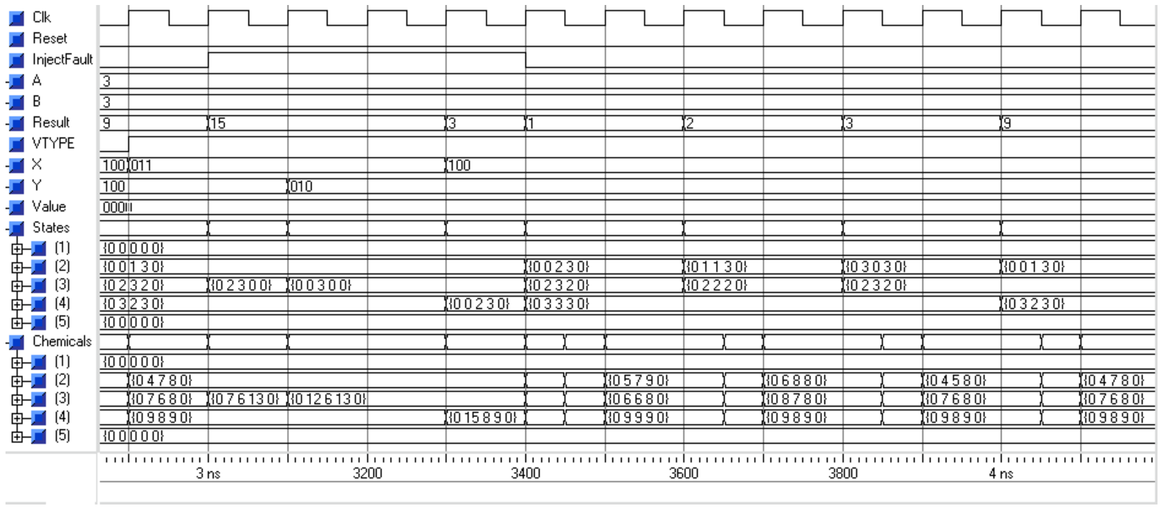


Fig. 8. Injection of the second set of faults and the recovery procedure

3.4 Hardware Simulation and Verification

The structures of the evolvable sub-circuits were evolved in software and a robust solution was applied to the hardware. The FPGA implementation was synthesized by ISE 6.1i from Xilinx, downloaded into the hardware. The detail of the waveform is demonstrated in Fig. 6. It can be seen that the organism matures at 1ns, when the state pattern is identical to that shown in Table 2. The following experiment was carried out: enough time was allocated to let the organism grow and mature (see Fig. 6). Subsequently, two sets of transient faults were injected: the first set was composed of 4 transient errors in the chemicals of cell (2, 1), (2, 2), (2, 3) and (3, 3); the other set of faults were injected into the states of cell (2, 1), (2, 3) and (3, 1). Every fault was chosen to make the corresponding value 0. The time between the injections of the two sets of transient faults was more than enough for the organism to recover completely and stabilize itself again in terms of chemicals and states of the cells (see Fig. 7 and Fig. 8).

The recovery from of the first set of transient chemical faults is illustrated in Fig. 7. At the beginning, the chemical of some of the cells were modified, and then the organism resumed growing. It recovered flawlessly at 2.4ns and the result output regained the correct value.

Fig. 8 demonstrates the recovery procedure from the second set of transient state faults. The states of the 3 selected "victim" cells were killed (state 0) at the beginning of this period. The organism again recovered completely to the correct pattern at 4ns.

The FPGA on the RC1000 board [10] connects to the host PC with a data width: 8-bit read and 8-bit write. A further FPGA module "IOControler" was implemented to latch all the required inputs and feed them to the digital organism. Another function of IOControler is to cache the result output of the digital organism.

4 Intrinsic EHW Implementation

After implementing the digital organism, several other functional modules which are indispensable in the Intrinsic Evolvable Hardware platform (IEHW) were identified according to "divide and conquer" principle. The top level modules are demonstrated in Fig. 10.

The IEHW implementation includes 3 main outputs: *MAX_Fitness*, *GenerationCount* and *Genotype*. The first and the second will be updated every generation to reflect latest values, while the last output always propagates the best individual that is evolved so far. Only two inputs are required for this IEHW to function as expected: the global clock signal and reset signal. Other inputs are optional parameters, such as the seed for *RNG* and stop fitness.

Before describing the modules in details, the evolution algorithm employed in this work will be discussed first.

4.1 Evolution Algorithm

Most existing widely employed evolution algorithms are designed primary for software implementation, so they are not particularly hardware friendly and efficient.

One of the significant hindrances of transforming the most popular EAs to hardware is the "sorting" issue. All of these popular EAs require some kinds of sorting to work properly. Taking (MU + Lambda) - Evolution Strategy [16] as an example[1], it selects the best individuals from both offspring and parents to generate the population of the next generation. This mechanism infers that the fitness for all offspring and parents has to be sorted to determine which the fittest individuals are. However, hardware implementation of sorting is not only inconvenient, but also extremely expensive in terms of both resources consumption and design complexity.

In order to evade the problems of the popular EAs and increase the efficiency of evolution, other alternative ones were investigated. D. Levi's HereBoy [14] algorithm was considered a good candidate, since it was designed specifically for the hardware (FPGAs).

HereBoy is a combination of the characteristics from both Genetic Algorithms and Simulated Annealing. The binary chromosome (a string of 1s and 0s), one of the most important features of Genetic Algorithm, is utilized as the data structure in HereBoy, for in principle this kind of chromosome can be mapped to any problem domain. The population only contains one individual, and mutation is the only variation operator employed, which is also the cases in Simulated Annealing [14]

In each step of iteration, the chromosome is mutated by flipping bits and then evaluated. If the mutation leads to a better individual than its parent, the offspring will be retained; otherwise the algorithm discards this mutation based on a possibility test. This means that sometimes a worse chromosome will be maintained. This mechanism allows the system to search for better solutions. [14]

It may be argued that HereBoy Algorithm is the same as the random mutation GA or the 1+1-ES. The most significant variation of HereBoy is that it honours a better individual most of the time, but does not favour a better one all the time: inferior offspring has a limited opportunity to overwrite its superior parent individual.

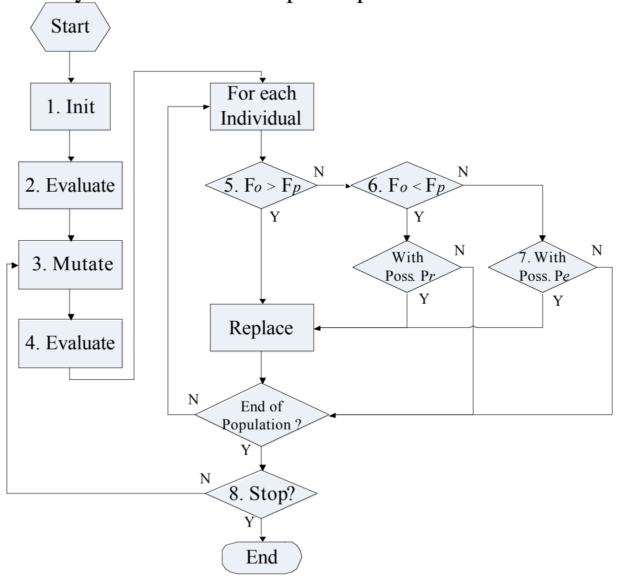


Fig. 9. Flow Diagram of the Evolution Algorithm Proposed

It is obvious that no sorting is required to perform a HereBoy type algorithm. Inspired by this algorithm, a new EA as follows was conceived:

1. Randomly initiate *n* individuals;
2. Evaluate all of them;
3. Mutate every individual once to generate *n* offspring; (Mutation rate p_m is fitness related, described below)
4. Evaluate all offspring;
5. If an offspring (F*o*) is better than its parent (F*p*), replace its parent with it.

[1] Only those ES in which MU is greater than 1 are considered here.

6. Else if an offspring is worse than its parent, the offspring has a probability p_r to substitute its parent;
7. Else (when an offspring's fitness is the same as its parent), the offspring has a constant probability p_e (which is determined as an input parameter beforehand) to substitute its parent;
8. Unless stopping criterion reached return to 3.

The flow chart of this algorithm is illustrated in Fig. 9. The main difference from the original HereBoy is that a population with more than one individual is possible in the proposed algorithm: HereBoy is a special case of this algorithm which has only one individual in its population.

Adaptive mutation rate has been shown to be efficient for hardware evolution [15]: a mobile robot can adapt to unpredictable environments with the help of an evolution algorithm which employs a mutation rate defined according to the normalized fitness. It was also suggestion in [14] that the evolution would benefit from adaptive mutation rate. Based on these findings, an adaptive mutation rate p_m was employed in this work. It is defined as:

$$p_m = \begin{cases} p_{\min} & (p_c < p_{\min}) \\ p_c & (p_{\min} < p_c < p_{\max}) \\ p_{\max} & (p_c > p_{\max}) \end{cases} \qquad (2)$$

p_c is calculated based on the individual's current fitness f and the maximum fitness f_{max}, given as:

$$p_c = p'_m(1 - \frac{f}{f_{\max}}) \qquad (3)$$

p_{min} , p_{max} and p'_m are parameters defined before evolution. In general, p_{min} multiplied by the number of total molecules should be greater than or equal to 1 and p_{max} should be equal to or less than 0.5.

p_m starts at high (normally p_{max}) when the evolution begins, and declines to p_{min} as the process converges on the final solution. This scheme allows the algorithm to focus on searching for generally good solutions at beginning and then fine tunes them to evolve the best one.

The probability p_r of a worse offspring replacing its parent is governed by a similar rule:

$$p_r = p'_r(1 - \frac{f}{f_{\max}}) \qquad (4)$$

p'_r is another input parameter, called the *principle maximum mutation rate*. The other part of the product which generates the p_r is a fraction which will reduce from 1 to 0 as the run converges. So p_r decreases as the fitness of individual approaches the maximum.

p_r is introduced to aid the avoidance of local fitness maxima. With this mechanism, the algorithm can search for better opportunities in its surrounding area when it is trapped in a local maxima.

4.2 Top-level Hardware Modules in the IEHW

As shown in Fig. 10, there are 5 functional independent top-level modules which implement the IEHW as a whole.

All the genotypes of each individual are stored in the *Population* module. This is implemented in the FPGA as distributed RAM, for only one individual is manipulated at any given time.

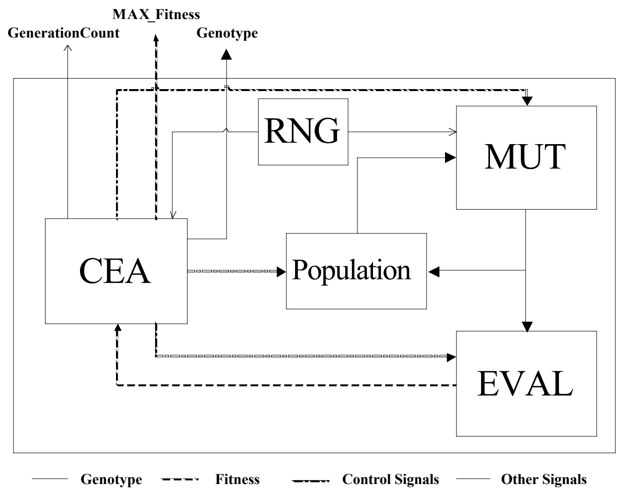


Fig. 10. Top-level Overview of the Intrinsic Evolvable Hardware Platform

The Controller of EA (*CEA*) is the central management component which supervises the entire evolution process and all the other modules. The fitness for all the individuals in the population is also stored in this module. The EA is essentially the same as described in section 4.1, with one exception: no adaptive mutation rate is employed, although adaptive replacement rate of a bad-offspring overwriting its better parent is used. The CEA module is realized as a finite state machine (FSM). It is obvious in Fig. 10 that the *CEA* has nothing to do with any genotype signals, so it is a representation-independent functional module.

RNG is a 64 bit Linear Feed-back Shift Register (LFSR) (more details are available in the references [11] and [12]), which is employed as a pseudo-Random Number Generator [13]. If supplied, the seed of RNG will also be saved in this module.

The main function of Mutation Module (*MUT*) is to mutate a given genotype and latch the mutated genotype to be used by the *EVAL*. This module reads in the mutation rate and mutates molecules one by one until the specified mutation number is met. This module is also implemented as an FSM.

The core component of this IEHW is the Evaluation Module (*EVAL*), where the Digital Organism resides. Its main function is to evaluate the fitness of each individual. This module feeds every possible input to the 2-bit multiplier implemented by the evolved digital organism and sums up the total correct bits. Finally, the result of the subtraction of the total correct bits from the maximum possible correct bits (which is 64 in this case) is the fitness of this individual. The *EVAL* module is made up from two FSMs: one is used to manage the digital organism and the other is in charge of feeding inputs, calculating correct output bits, the summary and the final subtraction to generate the fitness output of this module.

Rather than a central bus, all the modules have their dedicated input/output signals as shown in Fig. 10. This feature improves the efficiency of the evolution process.

4.3 Execution Phase of the IEHW

After reset signal is pulsed (low) for one clock cycle, all the modules, including all FSM and internal registers, are all cleared to their initial states.

In this state, the IEHW will receive and latch any input parameters if provided, otherwise the default parameters are used.

When the start signal is activated by the host PC, the *CEA* module will take all the responsibilities of the IEHW.

Firstly, the population are initiated one by one, evaluated and saved into *Population*: *CEA* signals the *MUT* to mutate at the highest possible rate so all the molecules in the genotype are randomly generated, then *EVAL* evaluates it and propagates the fitness to *CEA*, finally the *CEA* saves the fitness and signal the *Population* module to store the new generated individual. These individuals make up the 0 (first) generation.

Secondly, after the initial population are ready, the main loop of evolution process begins: in each generation, the *CEA* selects each of the individuals in the *Population* and feeds it to the MUT. The mutated genotype is then evaluated by the *EVAl* module, and the fitness is again propagated back to *CEA*. If the mutated one (offspring) is better than the original one (parent), or with a probability p_r a worse offspring substitutes its parent, which means the *CEA* asks the *Population* to store the mutated genotype; otherwise the content of *Population* module is untouched. After all the individuals have undertaken this procedure, a new generation is created. The evolution will continue to process the next generation unless the stop criterion, the specified fitness has been reached, is fulfilled.

When the main loop of the evolution process terminates, the best individual evolved is presented through the *Genotype* pin, while its fitness and the generation where this evolution stops are propagated out via *MAX_Fitness* and *GenerationCount* respectively.

5 Conclusions and Future Work

The biological development model proposed in this paper is capable of exhibiting the intrinsic highly fault-tolerance feature similar to its living organism counterpart when it is applied to real world application: the best solution discovered was able to tolerate virtually any transient damages.

Admittedly it is true that this model consumes more resources for the 2-bit multiplier than would be required in conventional majority voting systems. We feel that the approach used is worthy of further study. The decentralized nature of the circuits is an interesting aspect. The 2-bit multiplier is just a proof of concept on a simple problem. When this model is applied to more complex application, development has the potential to be more efficient than conventional approaches, just as the case in nature: human beings are too complicated to be described in detail, while DNA wisely encodes the develop procedure of a person. In the meantime, built-in fault-tolerant features can be expected in a generated system just as the situation described in this paper. We hope that, because of its development nature, this model can be applied to more sophisticated systems without fundamental modification.

With the help of the IEHW platform described in this paper, the evolution process is accelerated dramatically compared to software or VHDL simulation.

In future, the module will be extended to explore more possibilities, such as making full use of chemical signals when dealing with state signals and unconstrained growth world: At present, only the 2-lowest bits of chemicals are used in NSG, ignoring the top 2 bits and new generated chemicals will overwrite previous ones. Effort will be put into the manipulation mechanism of chemicals, such as introducing another kind of chemical behaving as energy and some anisotropic chemicals. A circuit capable of a particular function that is also able to grow and mature in a confined area by its own presents an interesting topic for future investigation.

In addition, the IEHW implementation will receive more improvement, including incorporation of adaptive mutation. With the IEHW, we can also carry out more researches about the impacts of different parameters to the evolution outcome.

References

[1] Lewis Wolpert, "Principles of Development" 2nd, Chapter 1, *Oxford University Press*, 2002.

[2] H. Ball and F. Hardy, "Effects and detection of intermittent failures in digital systems", 1969 FJCC, *AFIPS Conf. Proc.*, vol. 35, pp.329-335.

[3] Canham, R. and Tyrrell, A.M., "An Embryonic Array with Improved Efficiency and Fault Tolerance", *5th NASA Conference on Evolvable Hardware*, Chicago, pp 265-272, July 2003.

[4] Gordon, T.G.W., Bentley, P.J., "Towards development in evolvable hardware", *NASA/DoD Conference on Evolvable Hardware, 2002. Proceedings.*, pp. 241-250, 15-18 July 2002.

[5] Metta, G., Sandini, G., Konczak, J., "A developmental approach to sensori-motor coordination in artificial systems", *IEEE International Conference on Systems, Man, and Cybernetics 1998*, 10/11/1998 - 10/14/1998, 11-14 Oct. 1998, Location: San Diego, CA USA, pp. 3388 – 3393, vol.4.

[6] Julian F. Miller, "Evolving developmental programs for adaptation, morphogenesis, and self-repair", *Seventh European Conference on Artificial Life, Lecture Notes in Artificial Life*, Vol. 2801, pp. 256-265, 2003.

[7] J. Miller and P. Thomson, "Cartesian genetic programming", *Lecture Notes in Computer Science*, Vol. 1802, pp. 121-132, Poli, R., Banzhaf, W., Langdon, W.B., Miller, J. F., Nordin, P., Fogarty, T.C., (Eds.).

[8] Julian F. Miller, "Evolving a self-repairing, self-regulating, French flag organism", *GECCO 2004: Genetic and Evolutionary Computation Conference*, Seattle, WA, USA, Proceedings, pp. 129-139, 2004

[9] X. Chen and S. L. Hurst, "A comparison of universla-logic-module realizations and their application in the synthesis of combinatorial and sequential logic networks", *IEEE Transactions on Computers*, vol. C-31 pp. 140-147, 1982

[10] http://www.celoxica.com/

[11] P.H.R. Scholefield, "Shift Registers Generating Maximum-Length Sequences", *Electronic Technology*, pp.389-394, 10-1960.

[12] S.W. Golomb, "Shift Register Sequences", *Holden-Day*, San Francisco, 1967.

[13] Peter Alfke, "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators", *Xilinx Application note, XAPP 052 (Version 1.1)*, July 7 1996.

[14] D. Levi, "HereBoy: A Fast Evolutionary Algorithm", *Proceedings of the 2nd NASA/DoD Evolvable Hardware Workshop*, IEEE Computer Society, Los Alamitos, Ca, July 2000.

[15] R. Krohling, Y. Zhou and A. Tyrrell, "Evolving FPGA-based robot controller using an evolutionary algorithm", *1st International Conference on Artificial Immune Systems*, Canterbury, Sep 2003.

[16] Schwefel, H.-P. "Numerical Optimization of Computer Models", *Chichester: Wiley*, 1981.