# The Genetic Algorithm as a *Discovery Engine*: Strange Circuits and New Principles

Julian F. Miller[1], Tatiana Kalganova[2], Natalia Lipnitskaya[3], Dominic Job[4]
[1,2,4] School of Computing, Napier University, 219 Colinton Road, Edinburgh, UK, EH14 1DJ
[3] Dept. of Computing, State University of Informatics and Radioelectronics,
6 P.Brovky, Minsk, Belarus, 220 600
( [1]j.miller, [2]t.kalganova, [4]d.job)@dcs.napier.ac.uk; [3]nat.lip@usa.com

**Abstract**

This paper examines the idea of a genetic or evolutionary algorithm being an inspirational or discovery engine. This is illustrated in the particular context of designing electronic circuits. We argue that by connecting pieces of logic together and testing them to see if they carry out the desired function it may be possible to discover new principles of design, and new algebraic techniques. This is illustrated in the design of binary circuits, particularly arithmetic functions, where we demonstrate that by evolving a hierarchical series of examples, it becomes possible to re-discover the well known ripple-carry principle for building adder circuits of any size. We also examine the much harder case of multiplication. We show also that extending the work into the field of multiple-valued logic, the genetic algorithm is able to produce fully working circuits that lie outside conventional algebra. In addition we look at the issue of principle extraction from evolved data.

## 1 Introduction

There is a great contrast between the way in which physical systems have been designed by blind evolution and the top-down method employed by human designers. In the former case entire systems are constructed and tested in situ without a conscious application of principles. In the latter systems are "evolved" by a process of human ingenuity which employs a collection of rules, concepts and principles. It is indeed curious that organisms such as ourselves which are capable of imagining the world which operates according to definite laws and abstract design process were themselves produced by a mechanism which is entirely blind and has no particular object other than survivability. We argue in this paper that although it is difficult for an evolutionary algorithm to actually suggest new principles directly, new principles may still be inferred by studying an evolved series of examples. We also argue that by employing a blind evolutionary approach, dearly held assumptions and principles may be challenged, and thus, new concepts may emerge.

A well-defined context in which to examine these issues is in the field of electronic circuit design. Here human designers have abstracted the world of binary (an alphabet of *0* and *1*) and multiple valued quantities (an alphabet of *0*, *1*, …, *n*) and specified definite operations such as logical OR, AND, and MAX or MIN. In such a context the *objective* is to construct an electronic or algebraic machine for carrying out a definite function (e.g. addition, multiplication) on a number of input variables, using either as few operations as possible, or a modular construction which can be used to build much larger systems. The usefulness of these electronic machines is readily apparent in modern computers. There is a particular reason why attempting to evolve arithmetic circuits might be a useful and illuminating exercise. Such circuits are *modula*r in construction so that very large systems may be constructed from the smaller building blocks. This is clear when we recall that multiplication is a process of repeated addition, thus we can build multiplication circuits by using AND gates to perform elementary one-bit multiplication and then binary full-adders connected in an arrangement called a cellular array. This process is a classic example of human design. Firstly we construct building blocks which carry out functions which we have abstracted as being fundamental. Secondly we build larger systems by manipulating these building blocks by a process of abstract reasoning. When we allow biologically inspired algorithms such as evolutionary algorithms to design the building blocks and assemble the parts we discover an amazing number of new possibilities. This leads us to the main question studied in this paper and it is so important that we shall refer to it as The Fundamental Question (TFQ): *Can we by evolving a series of sub-systems of increasing size, extract the general principle and hence discover new principles?*

Such a question immediately leads to many other fundamental questions: What is a sub-system or building block? What is a principle? How can we extract a principle? To be honest there are not really very precise answers to these questions even in the context of human design. A building block is a sub-component which has proved useful *by experience*. A principle is really just an observed or deduced rule which is found to helpful in trying to form a synthesis of a wide range of data and may either directly or indirectly lead to a prediction which can be tested. How are principles discovered? This is the mysterious process of discovery which originates in the human mind and is often called intuition or creativity. It is our firm contention that TFQ will be answered in the affirmative it is just a matter of time. We hope to show in this paper that the specialised domain of electronic logic design is a perfect context in which to study this question. We will give a clear example of where we have been able to extract just such a principle (the principle of ripple-carry), however it is one that humans have known about for a few decades! We should clarify the concept of a principle a little more. Some principles are rules-of-thumb others have precise predictive power, often in Science accumulation and study of the former leads inevitably to the latter. Another feature of this paper is that we show some promising results which indicate *how* we might be able to extract principles from evolved circuit data. We do this by constructing a “fingerprint” for a class of circuits. Essentially this consists of looking at the relative frequencies of circuit sub-structures in entire evolved designs. This method and some of our results are discussed in section 5. One can think of this process as one of data-mining (DM) evolved systems and it potentially allows us to “close the loop” of principle extraction. As the extracted principles may feed back into the evolutionary algorithm.

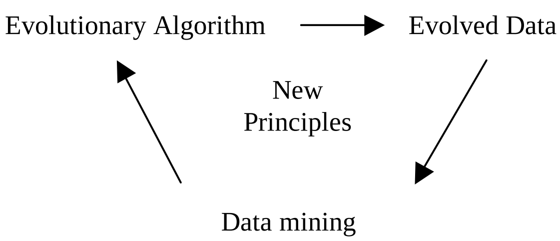



Figure 1: The principle extraction loop

In this paper we do not discuss the details of an evolutionary algorithm, but instead refer the reader to the plentiful texts which cover this field.

# 2 The space of all representations

Every binary and multiple-valued function is specified by a truth table. The truth table specifies what values the outputs of a function are for all values taken by the function inputs. There are certain special collections of operators that act on a binary or multiple-valued function that have the property that *any* function can be represented by expressions involving these operators and the input variables. The collection of these operators and the sets they operate on is often referred to as an *algebra.* In the case of binary functions there are two well-known algebras: Boolean which uses AND, OR, and NOT, and Reed-Muller which uses AND, EX-OR and NOT. Multiple-valued logic also has it’s own algebras and often they are referred to as *functionally complete bases*. When these algebras are used, a given function can only be represented by a particular class of expressions. The basic concept is shown in Fig. 5.

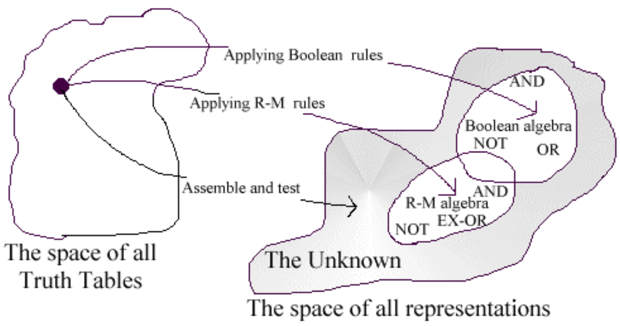



Figure 5: How *assemble-and-test* reaches the unknown regions of the space of all representations

The unknown region in Fig. 5 depicts all the representations of logic functions which are written as an expression which does not use operations take from the set {NOT, AND, OR, EX-OR}. Any expression in this region *once known* could be manipulated to become either an expression in the R-M or Boolean regions. To clarify the concept of a representation we give a simple example. Suppose we have a truth table which we discover by assemble-and-test can be represented quite simply as the output *y* given below:

$$y = \overline{f} x_4 \tag{1}$$

where $f$ is the output of a MUX gate with inputs $x_1$ and $x_2$ and control input $x_3$ . This is represented in circuit form below:

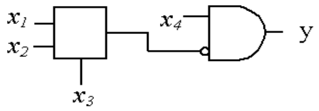



Figure 6: Example circuit corresponding to equation (1)

If this function were to be represented in the Boolean region of Fig. 5, it would have the expression given by equation (2), where we have represented the NOT operation by an over bar, the OR operation by +, and the AND operation is assumed between literals $x_i$.

$$y = \bar{x}_1 \bar{x}_2 + \bar{x}_1 \bar{x}_3 + \bar{x}_2 x_3 + x_4 \quad (2)$$

Implementing this expression as an actual circuit would require 6 logic gates (3 ANDs and 3 ORs) not including the NOT gates required.

It would be extremely difficult to develop an algorithm which did not use the assemble-and-test concept but would be able to synthesise circuits of the form given by equation (1). One of the fundamental contentions of this paper is that the assemble-and-test method is the *only* way that the space of all representations can be explored.

Although we have discussed the design of logic circuits the idea expressed in Fig. 5 could be regarded as an analogy for the design process in general. The space of all representations would then become the space of all designs.

## 3 Evolutionary Algorithms which assemble electronic circuits from a collection of available components

The idea of building electronic circuits by using an evolutionary algorithm to connect logic gates together from a set of possible types has only recently begun to emerge in recent years. Using the *assemble-and-test* methodology to try to get whole circuits to perform specific tasks rather than employing complex human design principles is one of the important themes which has arisen in the nascent field of *Evolvable Hardware* (Sipper et al. 1997). Many different approaches to this have been developed. Iba et al (1996) showed how boolean combinational logic functions (not involving time) could be built using a genetic algorithm to evolve the connections between AND, OR and NOT gates. Thompson (1996) showed how circuits could be synthesised on special devices called Field Programmable Gate Arrays (FPGAs) to carry out frequency discrimination tasks and robot control. The basic technique was to evolve binary configuration sequences to program an FPGA to carry out the desired task. There was no human input about *how* this might be done, just a measurement of the degree to which a given circuit achieved the desired response. Remarkably it was shown that tiny circuits could be evolved to efficiently carry out the task, but which operated in ways which are still not understood, and still under intense investigation. It was clear that the evolved circuits were making use of all properties which were potentially useful, including the physical properties of the silicon medium. In our own work in this field (Miller et al. 97, Miller et al. 98a, 98b, 98c) we have designed a powerful technique for evolving the choice of functions and the connections of an array of digital logic functions. The general concept of this can be envisaged with a simple example:

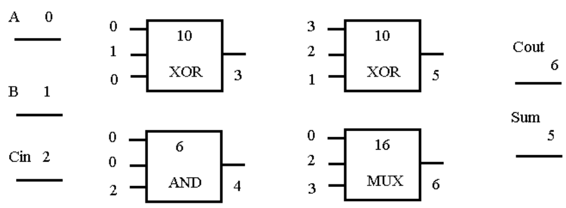



Figure 2: Gate array representation of evolved 1-bit adder

In Fig. 2 a gate array representation of an evolved one bit adder is given. The inputs A, B, and Cin are the binary inputs. The outputs Sum and Cout are the binary outputs. Sum represents the sum bit of the addition of A+B+Cin, and Cout the carry bit. The chromosome representation of this is shown below:

0 1 0 **10** 0 0 2 **6** 3 2 1 **10** 0 2 3 **16** 6 5

The figures in bold represent the functions of the corresponding logic cells. The allowed cell functions can be chosen to be any subset of those shown in Table 1, where ab implies a AND b, $\bar{a}$ indicates NOT a, ⊕ represents the exclusive-OR operation and + the OR operation.

Table 1: Allowed cell functions

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | a | b | $\bar{a}$ | $\bar{b}$ | ab | $a\bar{b}$ | $\bar{a}b$ | $\bar{a}\bar{b}$ |

| **10** | **11** | **12** | **13** | **14** |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| $a \oplus b$ | $a \oplus \overline{b}$ | $a + b$ | $a + \overline{b}$ | $\overline{a} + b$ |
| **15** | **16** | **17** | **18** | **19** |
| $\overline{a} + \overline{b}$ | $a\,\overline{c} + bc$ | $a\,\overline{c} + \overline{b}c$ | $\overline{a}\,\overline{c} + bc$ | $\overline{a}\,\overline{c} + \overline{b}c$ |

Functions 0-15 are the basic binary functions of 0, 1 and two inputs. Functions 16-19 are all binary multiplexers with various inputs inverted. The multiplexer (MUX) implements a simple IF-THEN statement (i.e. IF c=0 THEN a ELSE b).

An evolutionary algorithm is used to evolve circuits by beginning with a population of randomly initiated chromosomes. In some cases this was a Genetic Algorithm (GA) with uniform crossover (50% genetic exchange). The chromosomes are constrained so that columns of cells can only connect to cells to their left. This is necessary to ensure that the resulting circuits are not time dependent (feed-forward, combinational). Sometimes the evolutionary algorithm employed was a simple form of $(1+\lambda)$ evolutionary strategy (ES). In this case a population of $(1+\lambda)$ random chromosomes were randomly generated and the fittest chromosome selected. The new population is then filled with mutated versions of this. Random mutation was defined as a percentage of genes in the population which were mutated. The mutation operator respected the feed-forward nature of the circuits and also the different alphabets associated with connections and functions.

### 3.1 Binary circuit symbols

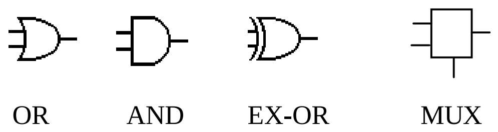



Figure 3: Binary circuit symbols

Note that on some diagrams a small circle may be seen on an input or output wire, this indicates that the input or output is inverted (by applying a NOT operation). This notation is also used in the multiple-valued case.

### 3.2 Multiple-valued circuits

The methods used in the section above have also been extended to multiple-valued circuits. Multiple-valued logic is an extension of the more familiar Boolean logic to an alphabet of positive integers *0, 1, …, n,* where *n* is referred to as the *radix*. In this logic the familiar operators such as logical a AND b, a OR b, are replaced by the minimum of {a, b} and maximum respectively. Many other operators can be defined which have no counterpart in binary. The details of this approach have been reported in (Kalganova et al. 98a, 98b). The following logic gates have been used in circuit evolution: NOT, MIN, MAX, MODSUM, TSUM, TPRODUCT, and any of these with the output inverted (excluding NOT).

In Fig. 4 various diagrammatic representations of multi-valued functions are given together with their symbolic expressions. Implementations of these gates can be found in (Jain et al. 93). We also use the 3-valued T-gate as one of the basic component for the design (Kameyama et al. 86). The T-gate is the multiple-valued counterpart of the binary multiplexer.

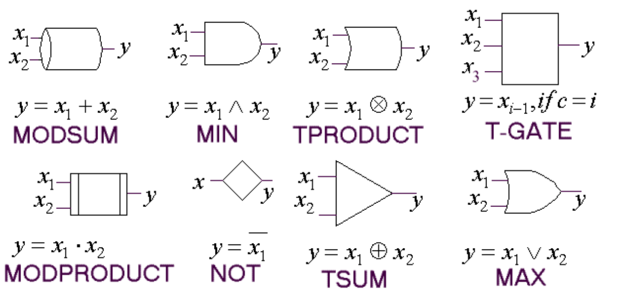



Figure 4: Symbols and analytic representation of two-input multiple-valued logic gates

## 4 Results

### 4.1 One-bit adder

The conventional one-bit adder is depicted below

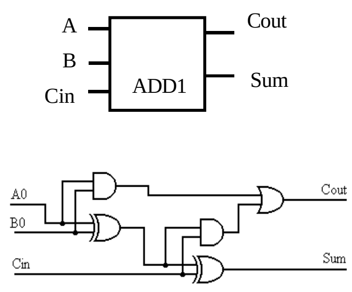



Figure 7: Block diagram of one-bit adder with carry and conventional circuit diagram

The conventional (most efficient) circuit diagram shown in Fig. 7 requires 5 logic gates of three types (AND, OR, EX-OR). Note that the sum output is implemented with

EX-OR gates. This is a characteristic of the human design.

When an evolutionary algorithm is used some strange but quite elegant solutions are possible. Fig. 8 depicts an efficient 1-bit adder which was obtained. In reality MUX gates can be are built out of smaller components, 2 ANDs and 1 OR. However some modern devices use the MUX gate as an ‘atomic’ device in that all other gates are synthesised using this. The MUX gate is unusual in this respect in that by merely setting one of the three inputs to 0 or 1 it can realise any binary function of two variables (actually there are precisely six functions which have this special property, often they are called universal logic modules or ULMs).

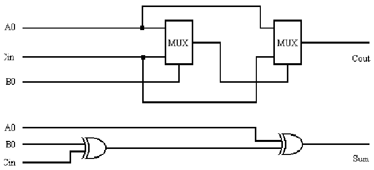



Figure 8: Evolved 1-bit adder with carry (A)

An interesting feature of this circuit is that the sum and carry parts of the circuit are completely de-coupled and there is a nice symmetry to it, which suggests that just as EX-OR gates naturally carry out elementary addition, so the MUX gate naturally synthesises the carry process of addition. This might be thought of as potential new principle which has not been observed by human designers. It should be noted that the circuit was actually evolved by separating the carry function from the sum and evolving them separately. When both are evolved together under a more constrained geometry (2 x 2 cells) it becomes more difficult for the EA to correctly synthesise the circuit and it requires about 20 runs of 2000 generations (population size 50, 100% breeding, 5% mutation) to produce a solution. This is shown in Fig. 9.

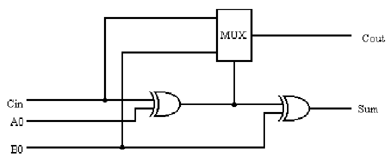



Figure 9: Evolved 1-bit adder with carry (B)

This is a remarkably efficient circuit (it’s genome was explained in section 3). Actually this circuit *is* known but was only recently discovered[1].

[1] Personal communication by John Gray, formerly of Xilinx, Edinburgh.

## 4.2 Two-bit adder

One of the principles used by human to construct larger adders is known as the *ripple-carry* principle. The block diagram for a two bit adder this is shown below:

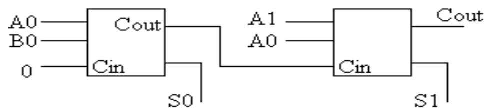
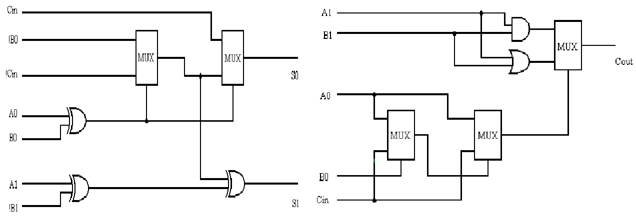



Figure 10: The two-bit ripple-carry adder

Each of the blocks in Fig. 10 are identical to that depicted in Fig. 7. The two-bit adder can perform the calculation $a+b$ where $a$ and $b$ are any positive integers between 0 and 3. We were able to evolve the two-bit adder without the insertion of any hints about how to do it. In the first design shown in Fig. 11 we again separated the sum part of the circuit from the carry part.




Sum circuit Carry circuit

Figure 11: Evolved two-bit adder design (A)

We did this so that we could see if the new evolved circuit has anything in common with the evolved one-bit adder (Fig. 8). It is clear that they as the carry circuit of the one-bit adder is actually used as an important part of the carry circuit for the two-bit adder. There is a lot of subtlety to this! As before with the one-bit adder we subsequently evolved the complete adder without the carry being separated. Again the task was more difficult. A 3 x 3 geometry was chosen and the genetic algorithm had the following parameters:- population size 50, 50,000 generations, 20 runs, breeding 100%, mutation 5%, elitism, levels-back = 2. In the 20 runs 5 solutions were obtained which 100% functional. The number of required cells were 6 (1), 7 (1), 8 (3). The best circuit is shown below:

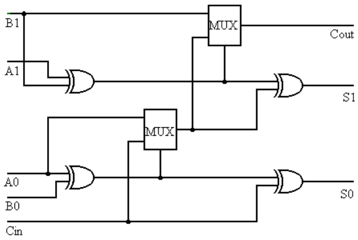



Figure 12: Evolved two-bit adder (B)

Comparing this circuit with the best evolved one-bit adder circuit (Fig. 9) it can be seen that the two bit adder circuit is produced by connecting the two smaller adders in a configuration identical to that shown in Fig. 10. Clearly we can then deduce the modular design principle of the ripple-carry adder and hence build adder circuits of *any* size. This demonstrates that principles can be extracted from a series of evolved examples. However in this case human designers already know the principle!

## 4.2 Two-bit multiplier

The process of designing a circuit to perform multiplication is modelled on the familiar long multiplication process shown below:

```
                A2    A1
                B2    B1
      ------------------------------
              A2B1  A1B1
        A2B2  A1B2
--------------------------------------
  P4    P3    P2    P1
```

Figure 13: Multiplication of two-bit binary numbers

The multiplication of two one-bit binary numbers is accomplished with and AND gate. Thus to build a circuit to multiply two integers a, b (between 0 and 3) one requires the circuit shown in Fig. 14.

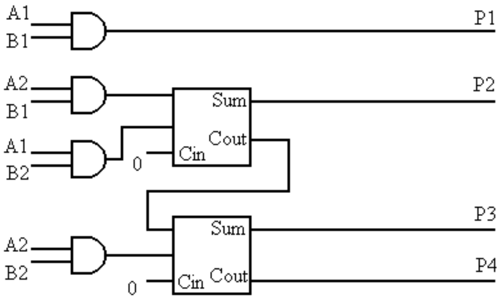



Figure 14: Two-bit cellular multiplier

A gate-level picture of this given in Fig. 15. The cross on the topmost AND gate indicates that in practise this gate is not required as the most significant bit of the output product (P4) can be obtained without it.

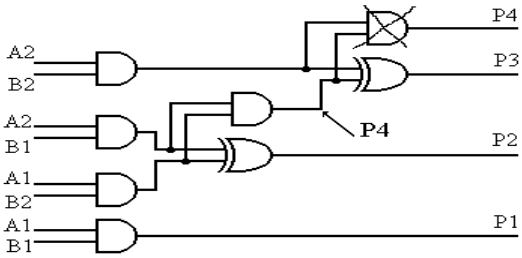



Figure 15: Gate diagram for most efficient two-bit multiplier

To obtain a plentiful supply of fully functional evolved two-bit multiplier circuits a (1+3)-ES algorithm was used with uniform mutation. The mutation probability was 0.02. One thousand runs of 50,000 generations were carried out with a 3 x 4 geometry and allowed gates types 6-16 (see Table 1). The levels-back parameter was set to 4. Of the 1000 runs 992 circuits were produced which were 100% correct. Of these 139 required only 7 gates so were as efficient as the conventional circuit shown in Fig. 15. Three of the evolved circuits are shown below:

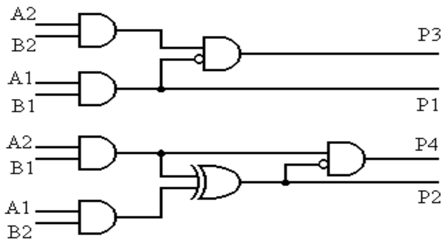



Figure 16: An evolved two-bit multiplier (A)

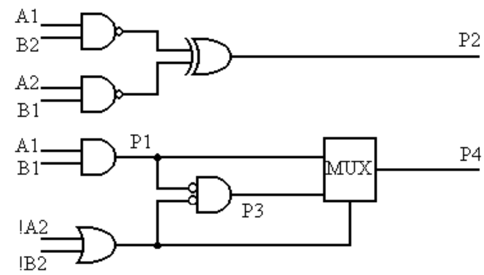



Figure 17: An evolved two-bit multiplier (B)

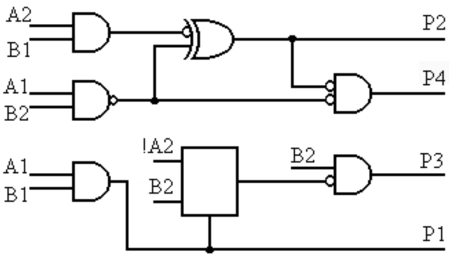



Figure 18: An evolved two-bit multiplier (C)

Examining these circuits instantly reveals their strangeness. In circuit A there are two independent sub-circuits, one involves P1 and P3 and the other, P2 and P4. This is very counter intuitive as in the conventional model of multiplication (Fig. 15) none of the outputs are re-used (except possibly P4 which turns out to be an input to the EXOR gate with P3 as the output). P3 is produced by three gates in circuit A, whereas it needs four in the conventional circuit. The circuit for P2 in all three evolved circuits is effectively the same. In the conventional circuit P1 has nothing to with P3, yet in two of these circuits (A and C) P1 is used to produce P3. Despite the fact that one can apply the symbolic rules of Boolean algebra to show that all these circuits are transformations of one another (including the conventional) the calculations are not obvious at all. Another strange feature of the circuits particularly circuit A is that only *one* EX-OR gate is required yet there are *two* additions. It would appear that our human concept of addition as only being modelled by the EX-OR operation is not correct.

Since our desire here is to answer TFQ in the affirmative we looked at the problem of evolving the three-bit multiplier. Could we discern any principles at work in the evolved two-bit multipliers, which also operated in the evolved three-bit multipliers which would enable us to build efficient multipliers of any size?

## 4.2 Three-bit multiplier

When the conventional 3-bit cellular multiplier is represented at gate-level and all redundant gates are removed the circuit shown below is obtained:

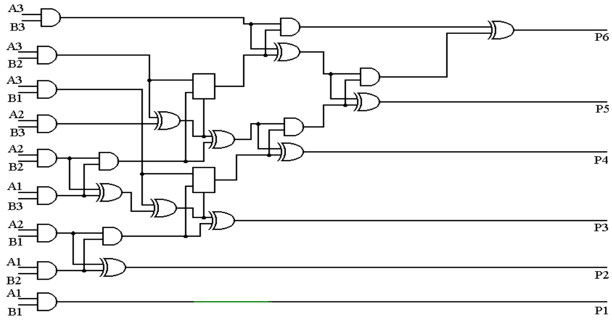



Figure 19: Most efficient conventional gate-level three-bit multiplier (30 2-input gates, 26 with MUXs)

The circuit shown in Fig. 19 can calculate the product of two integers a and b in the range 0-7. In our experiments we chose to use just the gates ab, a$\underline{b}$, and a^b (see Table 1) because we felt that the evolved two-bit multiplier shown in Fig. 16 was quite elegant and involved only three gates and also because of the 'fingerprinting' methods explained in section 5. We felt that if we examined evolved three-bit multipliers with just these gates we might stand a better chance of deducing some general principles. To obtain a reasonable probability of obtaining 100% correct solutions it was found that one had to evolve for of the order of 3,000,000 generations. We used a geometry of 6 rows and 7 columns with the maximum possible levels-back (7). The mutation probability was chosen to be 0.02.

We ran the ES 550 times and obtained 178 100% functional circuits. Table 2 shows the number of circuits, which required 30 gates or less. There were 58 circuits of this type. The most efficient conventional circuit shown in Fig. 19 requires 30 two-input gates. Thus these 58 circuits are all either equally or more efficient and 29 of these are more efficient. Note that in this experiment we did not use multiplexers. It is therefore possible that we could obtain even more efficient circuits.

Table 2. Experimental results for three-bit multiplier

| # runs | # 100% | # gates used | | | |
|---|---|---|---|---|---|
| total | cases | 30 | 29 | 28 | 27 |
| 550 | 178 | 29 | 17 | 11 | 1 |

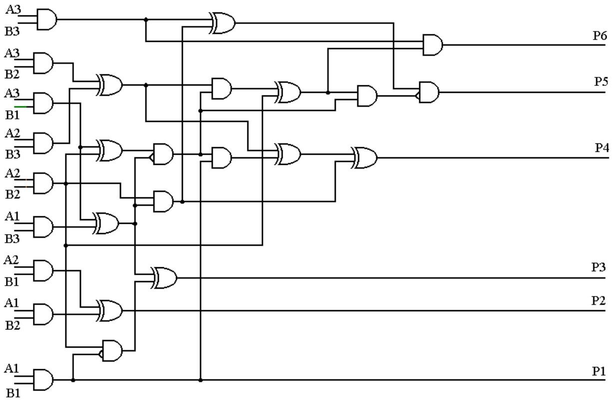



Figure 20: Evolved three-bit multiplier (most efficient – 26 gates)

The most efficient evolved three-bit multiplier shown in Fig. 20 requires only 26 gates rather than 27. This is because when we examined the evolved circuit we discovered that one of the gates used was logically redundant and could be removed. When we compare the evolved three-bit multiplier with the conventional we notice that there are considerable differences and it is not possible looking at the evolved circuit to see whether there are repeating modules. However we can see that outputs $P_1$ and $P_2$ are implemented in the same way in both circuits. In the evolved circuit $P_1$ is used twice whereas in the conventional circuit it is never used again. $P_2$ is not re-used in either circuit. $P_3$ requires 7 gates in the evolved circuit in comparison with 9 gates for conventional circuit. One of the interesting features of this design is that in the evolved circuit $P_3$ depends on $P_1$. This does not happen in the conventional circuit. The same is true of output $P_4$, it too depends on $P_1$ in the evolved circuit.

The differences between two circuits were very marked when we looked how the outputs $P_k$ depended on the elementary products ($A_iB_j$). Firstly we noticed that in conventional design $A_1B_1$ is not used by any other outputs. However in the evolved circuit this elementary product is used in the implementation of $P_3$ and $P_4$. The elementary products ($A_2B_1$) and ($A_1B_2$) which are used in the implementation of $P_2$ are not involved in any other outputs in the evolved circuit. However in conventional implementation these products are used for every output except $P_1$.

It is clear that the way the multiplication process is modeled in the evolved circuit is very different from the human one.

## 4.3 Multiple-valued one-digit adder with carry

Multiple-valued logic contains a lot of different algebras, which we can use to represent a given multiple-valued logic function. Each of these logic algebras contains a specific set of multiple-valued logic operators and is called in the literature a *functionally complete basis*. This means that it can implement any multiple-valued logic function of *n* variables. As we saw in section 2 there is a space of expressions, which represents a given function. This function can be represented by any of the functional complete bases. There are well-known human developed tools to map this function using the specific functional complete basis into a given expression. However we encounter problems when we want to combine some of these functional complete bases to represent a function in an economical way. Traditionally this would mean that we would have to develop a specific logic algebra, this takes a lot of time and effort and there is no guarantee of success at the end of the process. In addition it is only after this procedure that we are able to map the logic function into a given basis. This is one of the disadvantages of the human design procedure. In order to overcome this difficulty we, as in the binary case, adopted the method of assemble and test. This allows us to use *any* of the logic sets and in principle obtain *any* possible logic expression. Note that the set used should contain one of the functional complete basis known in order to guarantee success. Thus we potentially can discover new and highly efficient alternative representations which are counter to human intuition.

Here we will look at the some evolved designs for one-digit 3-valued adder with output carry. Three-valued logic functions can take the values in the set {0, 1, 2}. Thus, for example, in terms of base 3 arithmetic 1+2=10, that is to say, the sum is 3 carry 0. In Figs. 21-22 we give some examples of evolved circuits in the non-standard set of multiple-valued gates. Note that within any of the sets discussed we always used a functional complete basis as a subset of the gates chosen.

Fig. 21 shows some evolved designs for the one-digit 3-valued adder with carry. All these circuits have the same structure. The difference between them lies with gates 3 and 4. Analysis of the resulting logical expressions gives rise to equations which are extremely difficult to prove in a purely algebraic manner.

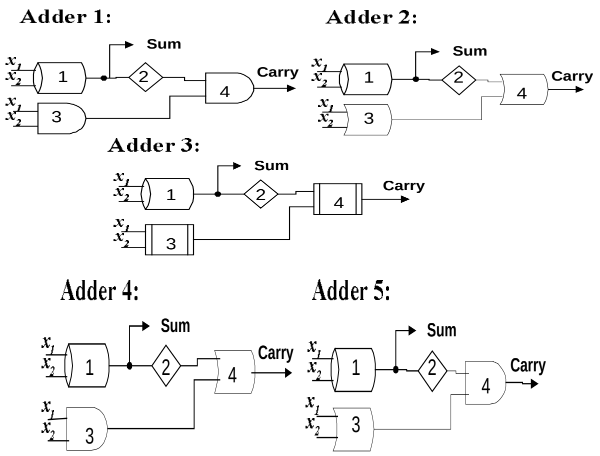



Figure 21: Evolved one-bit adder circuits which are identical except for gates 3 and 4.

However because the assemble-and-test method doesn't explicitly carry out formal algebraic operations we find such unusual structures relatively easily.

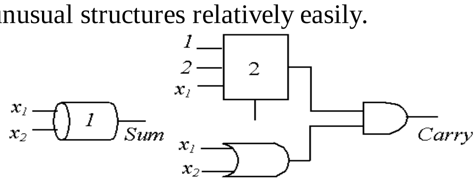



Figure 22: Evolved one-bit adder with T-gate

The circuit shown above involves four different types of gates. We were unable to find any multiple-valued logic design methods, which allowed us to represent the function in this way. The algebra of multiple-valued logic is still incomplete however using the assemble-and-test method allows us to escape from the restrictions inherent in a particular algebra.

# 5 Fingerprinting and Principle Extraction

There are many types of principles which potentially might be extracted from a database of genotypes of evolved circuits. In Fig. 1 we saw that after the stage of collecting evolved data one was faced with a problem of data mining. The extraction of knowledge from a large collection of evolved data is a little like trying to identify which genes (in terms of base sequences of DNA) are responsible for particular inherited characteristics. It is not an easy task. One approach to this is to try to categorise the various types of sub-circuit which are present in the evolved genotypes. Since there are many sub-circuits which are permutations of one another one must find a way of normalising the data so that the permutations are readily identifiable. Additionally one must look for sub-circuits of a particular form and size. This is necessary to avoid the combinatorial explosion which would occur if one wished to enumerate all possible sub-circuits. As a first attempt in this direction we decided to analyse the evolved genotypes (after permutational normalisation) in terms of 2-into-1 sub-circuits. The concept of a 2-into-1 sub-circuit is shown in Fig. X.

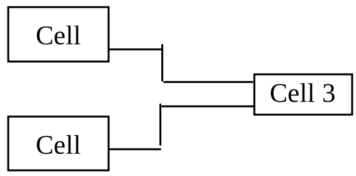



Figure 23: The 2-into-1 sub-circuit principle

In the diagram each cell may be any logic gate including a multiplexer. Each multiplexer would therefore have six 2-into-1 sub-circuits associated with it (as it has three inputs. Also if cell 3 was a a$\underline{b}$ gate (type 7) there would be two different sub-circuits of form XY7 and YX7 for each pair of X Y values (X,Y are allowed gate types in the genotypes). To avoid this explosion of possible sub-circuits we ignored which particular inputs of cell 3, cells 1 and 2 were connected to. We then indexed all possible XYZ triples and collated the frequencies of occurrence of these in the evolved material. We analysed the evolved genotypes associated with the 100% solutions for the two-bit multiplier. There were 61 distinct 2-into-1 principles which occurred more than ten times. Fig. 24 shows a histogram of these.

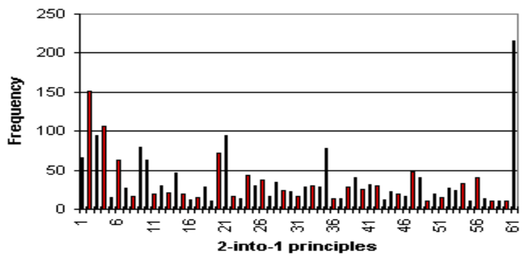



Figure 24: Distribution of 2-into-1 principles occurring in evolved two-bit multiplier genotypes (with frequency greater than 10)

The seven highest peaks correspond to the sub-circuits shown in Table 2. The 6-6-10 sub-circuit corresponds to two ANDs into EX-OR. In human design terms one would see this as adding two multiplications so we would not be very surprised to see this being an important principle in the multiplication process. The 6-6-6 sub-circuit could be seen as connected with the 'carrying forward' operation. The third most frequently occuring is 6-15-7.

Table 2: Seven most frequently occurring sub-circuits

| Cell 1 | Cell 2 | Cell 3 | Frequency |
|---|---|---|---|
| 6 | 6 | 10 | 215 |
| 6 | 6 | 6 | 151 |
| 6 | 15 | 7 | 107 |
| 6 | 6 | 7 | 94 |
| 6 | 6 | 8 | 94 |
| 15 | 6 | 8 | 80 |

Actually gate 15 is logically identical to a NAND gate, thus 6-15-7 is a close relative of 6-6-7. Also 6-6-8 is also a close relative as gate 8 is the same as gate 7 with the inputs reversed. Clearly the 6-6-7 sub-circuit is very important. Yet this structure is not involved in the conventional multiplier at all (Fig. 15). It can be seen in the evolved example shown in Fig. 16. One can think of the histogram shown in Fig. 24 as being a circuit ‘fingerprint’. Admittedly it is just one of a number of possible sub-circuit histogram plots but it is probably one of the most fundamental. Inspired by the obvious usefulness of gates 6, 7, and 10 (AND, AND with inverted input, and EX-OR) we evolved many 100% solutions for the three-bit multiplier where we had allowed only these three gate types. The evolved three-bit multiplier shown in Fig. 20 is an example of this. It is clear one should distinguish between inputs to obtain a better picture. This was carried out for the analysis of three-bit multiplier solutions which were 100% correct using gates 6, 7, and 10 only.  The fingerprint is shown below:

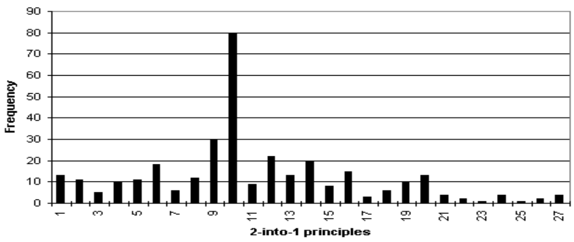



Figure 25: Distribution of 2-into-1 principles occurring in evolved three-bit multiplier genotypes

The highest peak corresponds to 6-6-10 with 80 occurrences. The next highest peak corresponds to 6-6-7 with 30 occurrences.

There is still much further work to be done here. A better way of trying to characterise 2-into-1 principles would be to look at logical behaviours as this would remove the occurrence of sub-circuits which have different gene triples but the *same* behaviour. Also one needs to take into account that useful modular sub-blocks may not fit neatly into the 2-into-1 category. This can be illustrated by the known usefulness of the half-adder in addition and multiplier circuits. The half-adder is shown below:

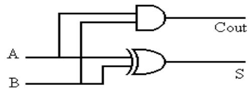



Figure 26: The half-adder sub-circuit

## 6 Conclusions

In this paper we have put forward the view that evolutionary algorithms together with the assemble-and-test methodology can be regarded as a discovery engine or creative machine for new designs. We studied this idea in the context of digital logic. We suggested that new principles may be able to be discovered by examining a series of evolved designs, in our case, for arithmetic logic circuits. We examined the concept of the space of all circuit representations but feel that similar ideas may well carry over to the general field of design. The human designed algebras which form subsets of the space of all representations both for binary and multiple-valued systems are analogous to small ‘pools’ of human principles and that by employing the blind evolutionary technique we may discover new principles. We also looked at the difficult problem of principle extraction from evolved data. We feel confident that the process of learning new principles from a blind evolutionary process is inevitable, it is just a matter of time.

## References

H. Iba, M. Iwata, T. Higuchi, Machine Learning Approach to Gate-Level Evolvable Hardware, in Higuchi T. et al. (Eds.), Proceedings of The 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES96), Lecture Notes in Computer Science 1259: pp. 327-343, Springer-Verlag, Heidelberg, 1997.

A. K. Jain, R.J. Bolton, M. H. Abd-El-Barr. CMOS Mutliple-Valued Logic Design - Part 2: Function Realization. IEEE Trans. on Circuits and Systems - I. Fundamental theory and applications. 40(8): pp. 515-522, 1993.

T. Kalganova, J. F. Miller, T. C. Fogarty. Some Aspects of an Evolvable Hardware Approach for Multiple-Valued Combinational Circuit Design. Proc. of the 2nd Int. Conf. on Evolvable Systems (ICES'98).

1478: pp. 78-89 Lausanne, Switzerland, M. Sipper et al. (Eds): Springer-Verlag, 1998a.

T. Kalganova, J. F. Miller, N. Lipnitskaya, Multiple-Valued Combinational Circuits Synthesized using Evolvable Hardware Approach. Proc. of the 7th Workshop on Post-Binary Ultra Large Scale Integration Systems (ULSI'98) in association with ISMVL'98, Fukuoka, Japan. IEEE Press, 1998b.

M. Kameyama, T. Higuchi, Synthesis of optimal T-gate Networks in Multiple-valued Logic. Proc. of the 16-th Int. Symposium on Multiple-Valued Logic, pp. 128-136. IEEE Press, 1986.

J. F. Miller, P. Thomson, Evolving Digital Electronic Circuits for Real-Valued Function Generation using a Genetic Algorithm. J. Koza et al., (Eds). Genetic Programming: Proceedings of the Third Annual Conference. Morgan Kaufmann. San Francisco, CA: pp. 863-868, 1998a.

J. F. Miller, P. Thomson, “Aspects of Digital Evolution: Evolvability and Architecture”, in A. Eiben et al. (Eds.), Proceedings of the 5th Int. Conf. on Parallel Problem Solving from Nature (PPSNV), Lecture Notes in Computer Science, Vol. 1498, Springer-Verlag, Heidelberg, pp. 927-936, 1998b.

J. F. Miller, P. Thomson, Aspects of Digital Evolution: Geometry and Learning, in M. Sipper et al. (Eds.), Proceedings of 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES98), Lecture Notes in Computer Science, 1478, Springer-Verlag, Heidelberg, pp. 25-35, 1998c.

J. F. Miller, P. Thomson, T. C. Fogarty, Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study, in Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: D. Quagliarella et al. (Eds.), pp. 105-131, Wiley, 1997.

M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Perez-Uribe, A. Stauffer, “A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems”, IEEE Trans. on Evolutionary Computation, 1(1): pp. 83-97, 1997.

A. Thompson, An evolved circuit, intrinsic in silicon, entwined with physics, in Higuchi T. et al. (Eds.), Proceedings of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES96), Lecture Notes in Computer Science, 1259: pp. 390 – 405, Springer-Verlag, Heidelberg, 1997