Principles in the Evolutionary Design of Digital Circuits — Part I

Julian F. Miller
School of Computer Science
The University of Birmingham
Birmingham, B15 2TT
England
j.miller@cs.bham.ac.uk

Dominic Job
School of Computing
Napier University
Edinburgh, EH14 1DJ
Scotland
d.job@dcs.napier.ac.uk

Vesselin K. Vassilev
School of Computing
Napier University
Edinburgh, EH14 1DJ
Scotland
v.vassilev@dcs.napier.ac.uk

October 28, 1999

Abstract

An evolutionary algorithm is used as an engine for discovering new designs of digital circuits, particularly arithmetic functions. These designs are often radically different from those produced by top-down, human, rule-based approaches. It is argued that by studying evolved designs of gradually increasing scale, one might be able to discern new, efficient, and *generalisable* principles of design. The ripple-carry adder principle is one such principle that can be inferred from evolved designs for one and two-bit adders. Novel evolved designs for three-bit binary multipliers are given that are 20% more efficient (in terms of number of two-input gates used) than the most efficient known conventional design.

1 Introduction

Traditionally physical systems (e.g. bridges, computers, mobile phones) have been designed by engineers using complex collections of rules and principles. The design process is top-down in nature and begins with a precise specification. This contrasts very strongly with the mechanisms which have produced the extraordinary diversity and sophistication of living creatures. In this case the "designs" are evolved by a process of natural selection. The design starts as a set of instructions encoded in the DNA whose coding regions are first transcribed into RNA in the cell nucleus and then later translated into proteins in the cell cytoplasm (Coen, 1999). The DNA carries the instructions for building molecules using sequences of amino acids. Eventually after

a number of extraordinarily complex and subtle biochemical reactions an entire living organism is created. The survivability of the organism can be seen as a process of assembling a larger system from a number of component parts and then testing the organism in the environment in which it finds itself. In this paper this is referred to as *assemble-and-test*. Figure 1 illustrates this concept in the general space of designs. The top-down rule-based space of designs is shown in grey as a small sub-region in the much larger space of all possible designs. Occasionally by a process of human inspiration or accidental discovery this space is widened as new concepts and principles are developed. Generally restrictive assumptions have to be made about the range of parts which can be used within this space. This is imposed by the constraints of a tractable system of rules. On the other hand it is argued here that by employing the simple idea of assemble-and-test together with an evolutionary algorithm one can explore the entire design space and use a much larger collection of parts precisely because of the absence of imposed rules of design.

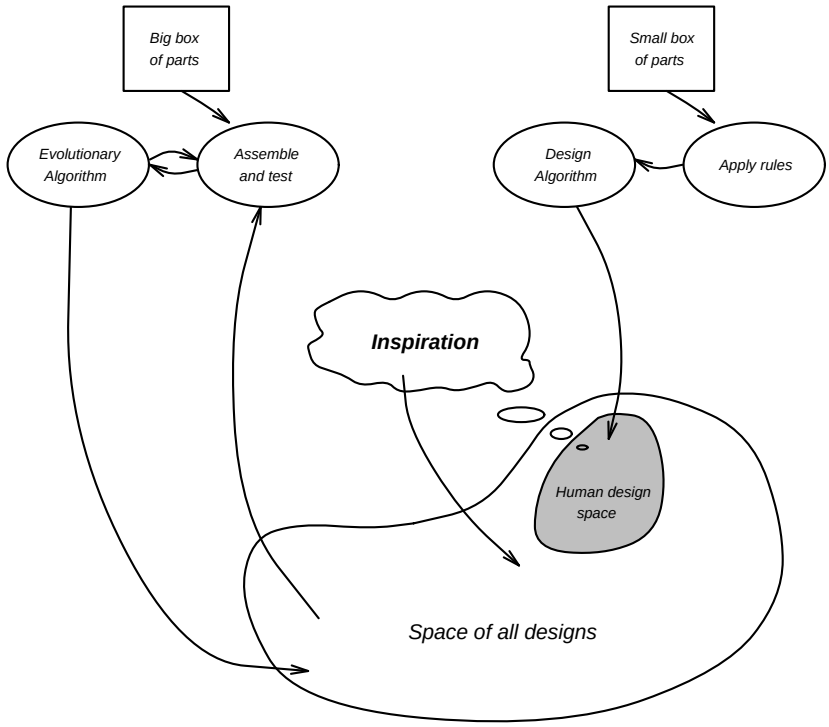


Figure 1: Conventional design versus evolutionary design with *assemble-and-test*.

The concept of assemble-and-test together with an evolutionary algorithm to gradually improve the quality of a design has largely been adopted in the nascent field of *Evolvable Hardware* where the task is to build an electronic circuit (Atmar, 1976; Higuchi *et al.*, 1993a; Higuchi *et al.*, 1993b; Kitano, 1994; Moreno, 1994; Thalmann and Thalmann, 1994; Grimbleby, 1995; Kruiskamp and Leenaerts, 1995; Higuchi and Iwata, 1996; Tomassini and Sanchez, 1996; Blickle, 1997; Sipper, 1997; Mange and Tomassini, 1998; Thompson, 1998). The circuits are encoded in *genotypes* from which the actual circuits

or *phenotypes* are constructed. Research in Evolvable Hardware can be sub-divided into two main categories: *intrinsic evolution* and *extrinsic evolution.* The former refers to an evolutionary process in which each phenotype is built in electronic hardware and tested. The latter uses a model of the hardware and thus evaluates the phenotypes in software. Each of these categories can be further sub-divided into analogue or digital domains. Intrinsic evolution in the analogue domain has recently become possible because of the availability of reconfigurable analogue devices (Motorola, 1997; Grundy, 1994) and already researchers have begun to explore the possibilities for automatic design that they afford (Murakawa *et al.*, 1998; Flockton and Sheehan, 1998; Zebulum *et al.*, 1998; Stoica *et al.*, 1998; Zebulum *et al.*, 1999; Stoica *et al.*, 1999; Flockton and Sheehan, 1999). Thompson (1997) used a reconfigurable digital platform (Xilinx 6216 Field Programmable Gate Array) and showed that it was possible to evolve a circuit which could discriminate between two square wave signals of frequencies 1KHz and 10KHz. It became apparent in this work that the evolutionary process had utilised physical properties of the underlying silicon substrate to produce an efficient circuit (only a 10×10 array of logic cells had been allowed). Indeed Thompson *et al.* (1999) have recently argued that artificial evolution can produce intrinsic designs for electronic circuits which lie outside the scope of conventional methods. Koza *et al.* (1996, 1999) have pioneered the extrinsic evolution of analogue electronic circuits using the SPICE simulator and has automatically generated circuits which are competitive with those of human designers. Zebulum *et al.* (1998) pointed out that analogue circuit simulation software assumes expert users and that one must be very careful to apply additional physical constraints to the particular simulation package being employed. However they showed that provided that this is done one can obtain real behaviour from an evolved circuit which is in close accordance with that obtained in simulation. Intrinsic evolution for purely digital systems has been pioneered by Kajitani *et al.* (1998) but most workers are content with extrinsic evolution (Miller *et al.*, 1997; Iba *et al.*, 1997). One of the advantages of extrinsic digital evolution is that one can obtain symbolic representations of circuits which can be implemented on a variety of digital platforms.

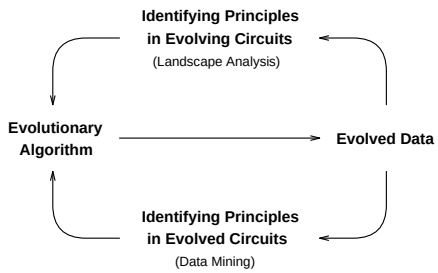


Figure 2: The principle extraction loop.

This paper is only concerned with the evolution of digital *combinational* (non-sequential) circuits. The paper concerns itself with just two kinds of digital functions: even-parity functions and arithmetic functions. These are sufficient to explore the efficiency of the techniques and the novelty of new designs. Even-parity functions are well known to be difficult to find by random search when the primitive operators are chosen

from the set {AND, NAND, OR, NOR} (Koza, 1992). Consequently they have been used extensively to test the effectiveness of various algorithms (Poli *et al.*, 1999). Arithmetic functions (addition and multiplication) are extremely good candidates to be used in the study of digital evolution for two main reasons: (a) they are modular in construction, and (b) there are well established conventional methods of building them. They are also fundamental building blocks of many digital devices. They afford a study of **T**he **F**undamental **Q**uestion (TFQ) addressed in this paper:

> *"Can we by evolving a series of sub-systems of increasing size, extract the general principle and hence discover new principles?"*

This leads on to the further question: In which type of design problems is it most likely that new principles might be discovered? Clearly since the search space of all possible designs is enormously enlarged compared with traditional rule-based methods one requires an extremely fast fitness function. It should be anticipated that tens of millions, even billions, of genotypes will have to be examined. Digital circuit design is an ideal candidate for novel principle extraction. The fitness function merely uses the bitwise operations that CPUs were designed for. For example, on a 450MHz PC one can evaluate 50,000 designs for a three-bit multiplier per second (in a 1×30 geometry).

In this paper it is argued that TFQ can be answered in the affirmative so that in the particular instance of designing arithmetic circuits new principles of scalable design can be discovered. A simple example of how such discoverable and scalable principles can be extracted is given in section 4 where it is seen that the principle of the *ripple-carry adder* follows as a consequence of examining the best evolved designs for the one and two-bit adders with carry. The processes by which new principles might be discovered is shown in Figure 2 which illustrates the cycle of evolutionary discovery and forms a diagrammatic representation of the concept of this work that is covered in this paper, Part I, and its sequel Part II (Miller *et al.*, 2000). In section 3 the way in which a digital circuit is encoded into a genotype and the characteristics of the evolutionary algorithm are given. This produces, given a sufficient number of generations, designs for fully functional circuits. This evolved data is discussed in section 4. In the sequel paper, the techniques of landscape analysis developed in (Vassilev, 1997; Vassilev *et al.*, 1999) are discussed, and using this, the principles by which an effective evolutionary search may be conducted are studied. The process of discerning design rules and principles from the evolved data can be seen as a form of data mining (Job *et al.*, 1999). This can make recommendations about useful components and sub-structures which feed back into the evolutionary algorithm and so improve the evolvability of the circuits in question and enhance our ability to understand the new designs. This is also examined in the succeeding part. The paper begins with a discussion of conventional methods of digital design (section 2) and closes with conclusions and suggestions for future work (section 5), some of which are discussed in Part II.

2 Conventional Logic Synthesis

2.1 Definitions and Preliminaries

The symbols for logical operators used in this paper are given below:

– indicates the NOT operator

$\cdot$ indicates the `AND` operator

$+$ indicates the inclusive-`OR` operator

$\oplus$ indicates the exclusive-`OR` operator (`XOR`)

The term *literal* refers to a Boolean variable or its complement. When two literals are next to one another the `AND` operator is assumed (e.g. $ab \equiv a \cdot b \equiv a$ `AND` b). A *product* term refers to number of literals connected by the `AND` operator. A *minterm* is a product involving all input variables where the output of a function is one. It is denoted by a decimal equivalent with inverted variables being represented by 0 (e.g. for a three variable function $a \cdot \overline{b} \cdot c$ is represented by 5). A *prime implicant* is a product term that cannot be combined with any other product term to generate a term with fewer literals than the original term. A prime implicant is called *essential* when it implies at least one minterm that is not implied by any other prime implicant of the function. A *don't care* is a minterm for which the function output is unspecified (either one or zero). A function with minterms m_i and unspecified terms d_i is written $\sum_i m_i + \{d_i\}$.

De Morgan's theorems are useful for converting `NAND` gates in expressions to `OR` gates, and also `NOR` gates to `AND`. The theorems are given below:

Theorem 2.1 *For Boolean variables a and b*

$$\overline{a \cdot b} = \overline{a} + \overline{b} \tag{1}$$

$$\overline{a + b} = \overline{a} \cdot \overline{b} \tag{2}$$

$\diamond$

2.2 Specification of Logic Functions

It is beyond the scope of this paper to give a complete description of Boolean algebra (Devadas *et al.*, 1994; Lala, 1996). Combinational logic functions are commonly specified by `pla` files (`pla` stands for programmable logic array – pronounced $pee - ell - ay$). A `pla` file has the format shown in Table 1.

$.i$	4
$.o$	2
$.p$	4
0-0-	1-
0-1-	-1
1-0-	10
1111	01
$.e$	

Table 1: An example `pla` file for a four variable function.

This is interpreted in the following way. The four input logic function with two outputs has four product terms. Suppose that the inputs are labelled from a to d going from left to right and the outputs are labelled y and z. The first product term is $\overline{a} \cdot \overline{c}$ and y is 1 in this case and one doesn't care about the value of z. Thus one can interpret

the dashes in the input field as missing variables (b and d in this case) and dashes in the output field imply that the output variable is a don't care. Note that the `AND` operator is assumed between literals in the input field and the `OR` operator is assumed between product terms. Thus the outputs y and z can be written

$$y = \overline{a} \cdot \overline{c} + a \cdot \overline{c} + \{\overline{a} \cdot c\} \tag{3}$$

$$z = \overline{a} \cdot c + a \cdot b \cdot c \cdot d + \{\overline{a} \cdot \overline{c}\}. \tag{4}$$

The `pla` file is an abbreviated truth table (where all inputs are specified) and doesn't list products for which all the outputs are zero.

2.3 Canonical and Two-level Boolean Functions

Logic functions can be represented in a variety of different ways. First one can use a *two-level* representation in which literals are combined with a single operator (in equations 3 and 4 above this is the `AND` operator) and then these terms are combined with a second operator (the `OR` operator above). When a logic function is expressed in terms of product terms which involve all input variables and all true output products are present the expression is referred to as a canonical Boolean expression. Usually the goal of logic synthesis is to represent a logic function in the simplest way by reducing the number of product terms and literals.

2.4 Karnaugh Maps, The Quine-McCluskey Algorithm and `ESPRESSO`

The Karnaugh map of a Boolean function of four variables

$$f = \sum(1, 4, 5, 10, 12, 13, 14, 15) \tag{5}$$

is depicted in Figure 3 (symbol **a** is most significant). The Karnaugh map is a way

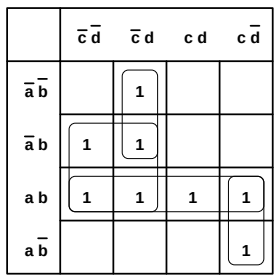


Figure 3: The Karnaugh map of function $f = \sum(1, 4, 5, 10, 12, 13, 14, 15)$.

of representing a Boolean function so that logically adjacent terms (with Hamming distance 1) are physically adjacent (the map is cyclic so that the horizontal and vertical boundaries are taken to be logically adjacent). Minterms are entered on the map as ones. The loops around the groups of ones in the map represent a possible simplification by applying the rules of Boolean algebra. The final result of employing a Karnaugh map

is to reduce a function to a sum of its essential prime implicants. It can be shown that f may be simplified to

$$f(a,b,c,d) = \overline{a} \cdot \overline{c} \cdot d + b \cdot \overline{c} + a \cdot b + a \cdot c \cdot \overline{d}. \quad (6)$$

Note that this gives an expression which is the sum of products. If zeros are marked on the Karnaugh map instead of ones and the same operations applied one obtains a product-of-sums representation of a logic function. This is because one obtains a sum-of-products for the complement of the original function which on inversion and applying De Morgan's theorems become a product-of-sums.

The Quine-McCluskey algorithm (Quine, 1952; McCluskey, 1956) is an exhaustive search method which essentially formalises the operations used for simplifying a function in a Karnaugh map. It is only practical for functions with small numbers of input variables. An effective and widely used heuristic method called `ESPRESSO` (Brayton *et al.*, 1984) is used to minimise two-level `AND-OR` representations of Boolean functions. There are some functions that have a sum-of-product representation that grow exponentially with the number of input variables. The Achilles Heel function (Brayton *et al.*, 1984), the parity functions and the n-bit multiplier are examples of this.

2.5 `NAND-NAND` and `NOR-NOR` Representations

Any Boolean logic function can be built entirely using `NAND` gates or `NOR` gates. The method of obtaining a minimised two-level representation of a logic function with `NAND` gates is the following:

1. Take the complement of the minimised Boolean sum-of-products.
2. Take the complement of the complemented expression. Eliminate the `OR` operator from the resulting expression by applying De Morgan's theorem.

The steps required to obtain a representation of a logic function using `NOR` gates is as follows:

1. Derive the product-of-sums expression of the function.
2. Take the complement of the product of sums expression and eliminate `AND` operators by applying De Morgan's theorems.

2.6 Multilevel Boolean Functions

A multilevel representation of a logic function allows the use of factoring and decomposition into sub-functions. For example

$$f = a{\cdot}b{\cdot}e{\cdot}\overline{g}+a{\cdot}b{\cdot}f{\cdot}g+a{\cdot}b{\cdot}\overline{e}{\cdot}g+a{\cdot}c{\cdot}e{\cdot}\overline{g}+a{\cdot}c{\cdot}f{\cdot}g+a{\cdot}c{\cdot}\overline{e}{\cdot}g+d{\cdot}e{\cdot}\overline{g}+d{\cdot}f{\cdot}g+d{\cdot}\overline{e}{\cdot}g \quad (7)$$

can be written

$$f = (a \cdot (b + c) + d) \cdot (e \cdot \overline{g} + g \cdot (f + \overline{e})). \quad (8)$$

The starting point for multilevel minimisation is the minimum two-level canonical form. Sophisticated heuristic minimisation algorithms have been written which try to reduce the literal counts in Boolean multilevel expressions (equation 7 has 33 literals, while equation 8 has 9) (Brayton *et al.*, 1987; Brayton *et al.*, 1990; Kunz and Menon, 1994).

2.7 Decision Diagrams

Classical representations of Boolean functions like truth tables, canonical sum-of-products, Karnaugh maps are impractical as their size is exponentially dependent on the number of inputs. Even representations with prime essential implicants are problematic as simple operations like complementation may produce representations of exponential size. In addition these representations have different equivalent forms making it difficult to check the equivalence of two functions. *Binary decision diagrams* (BDD) were proposed by Lee (1959) and developed by Akers (1978) but they are not necessarily canonical in form. However, Bryant (1986) showed that *reduced-ordered* binary decision diagrams do have a canonical form. A BDD is a rooted acyclic, directed graph with vertex set V containing two types of vertices. A *nonterminal* vertex v has the following attributes: a decision $i = index(v) \in \{1, ..., n\}$ and two children $low(v), high(v) \in V$. A *terminal* vertex v has as attribute $value(v) \in \{0, 1\}$. A BDD with root vertex v denotes a function f_v defined recursively as:

$$f_v(x_1, ..., x_n) = \begin{cases} 0, & \text{if } v \text{ is terminal and } value(v) \neq 1 \\ 1, & \text{if } v \text{ is terminal and } value(v) = 1 \\ \overline{x}_i f_{low(v)}(x_1, ..., x_n) + x_i f_{high(v)}(x_1, ..., x_n), & \text{otherwise,} \end{cases} \tag{9}$$

where x_i is called the decision variable for vertex v. A BDD is ordered if for any nonterminal vertex v for which $low(v)$ and $high(v)$ are nonterminal $index(v) < index(low(v))$ and $index(v) < index(high(v))$. The OBDD for the even 4-parity function is shown in Figure 4. The even n-parity function requires $2n - 1$ vertices in an OBDD rep-

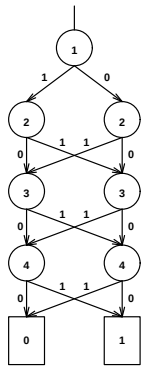


Figure 4: The ordered binary decision diagram for the even four-parity function.

resentation but 2^{n-1} in the minimum sum-of-products representation. A reduced ordered binary decision diagram (ROBDD) is defined as a BDD in which there is no vertex with $low(v) = high(v)$ nor does it contain distinct vertices v and w such that the subgraphs rooted by v and w are isomorphic (equivalent). One of the problems with BDDs is that their size is very strongly dependent on the variable ordering and there have been many heuristics devised to find a good ordering (Brace *et al.*, 1990; Friedman and Supowit, 1990; Fujita and Matsunaga, 1993). Indeed evolutionary algorithms have also been applied to finding a good ordering (Drechsler *et al.*, 1996). There have been many other types of decision diagrams proposed which can provide smaller

more efficient representations of Boolean functions. The *ordered Kronecker functional decision diagrams* (OKFDD) are canonical and can provide a more efficient representation in which XOR and OR gates are used (Drechsler *et al.*, 1994a). There are many other decision diagram methods (Jain *et al.*, 1992; Shen *et al.*, 1995) (these are not canonical). There are certain functions whose canonical OBDDs have provably exponential numbers of vertices as functions of the number of input variables. The n-bit multiplier is an example of this (Bryant, 1991) and also the Devadas function (Devadas, 1993).

2.8 Reed-Muller and Exclusive-OR Logic

It is well known that many Boolean functions which can be easily implemented using exclusive-OR gates are very inefficiently represented in canonical Boolean logic. The most extreme case of this being the n-parity functions which can be realised with $n-1$ XOR gates only but which require $2^{n-1}-1$ OR gates and a large number of AND gates. When a Boolean logic function is expressed using XOR gates and uncomplemented variables it is called a *Reed-Muller (RM) canonical form* (Green, 1986). If any particular variable is allowed to be complemented or uncomplemented throughout the expansion then the representation is known as a *fixed polarity RM form.* Finding a good polarity is a difficult problem and evolutionary algorithms have been used (Miller *et al.*, 1994; Drechsler *et al.*, 1994b). Work has been done on trying to minimise the less restricted XOR sum-of-products representations (Sasao, 1993; Thomson and Miller, 1996).

2.9 Exploring the Space of All Representations

In section 1 it was seen how the use of an evolutionary algorithm combined with assemble-and-test could be used to explore over a much larger area of design space than that possible using a top-down rule based design algorithm. Figure 5 shows a particular case of this for the problem of finding efficient representations of Boolean functions and it illustrates one of the fundamental concepts of this paper.

In conventional logic design one begins with a precise specification in the form of a truth table, `pla` file, binary decision diagram, symbolic expression etc. The expression is manipulated by applying canonical Boolean rules or Reed-Muller algebraic rules. One never escapes from the space of logically correct representations. The methods though powerful in that they can handle large numbers of input variables are not adaptable to new logical building blocks and require a great deal of analytical work to produce small optimisations in the representation. Assembling a function from a number of component parts begins in the space of all representations and maps it into the space of all the truth tables with m input variables ($m \leq n$). The evolutionary algorithm then gradually pulls the specification of the circuit towards the target truth table (shown as a small dark ellipse). Thus the algorithm works in a much larger space of functions many of which do not represent the desired function. It is one of the contentions of this paper that this is the only way one can discover radically new designs.

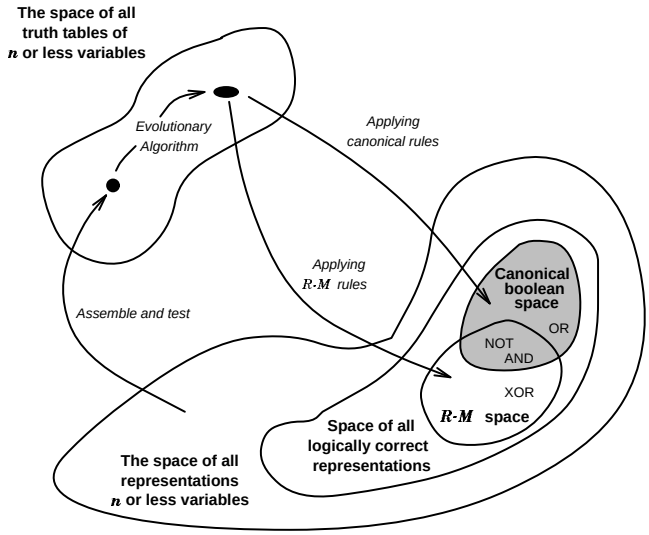


Figure 5: How "*assemble-and-test*" reaches the unknown regions of the space of all representations.

3 Digital Circuit Evolution

3.1 Encoding a Digital Circuit as an Indexed Graph

The encoding of a digital combinational circuit into a genotype which is presented in this paper is a development and simplification of earlier models (Miller *et al.*, 1997; Miller and Thomson, 1998b; Miller and Thomson, 1998a). It treats a digital logic circuit as a particular case of a more general graph based computational model called *Cartesian Genetic Programming* (CGP) (Miller, 1999). CGP has some similarities with other graph based forms of genetic programming, in particular, Parallel Distributed GP (PDGP) proposed by Poli (1997) and Parallel Algorithm Discovery and Orchestration (PADO) (Teller and Veloso, 1995) and represents a dataflow graph (see page 258 (Banzhaf *et al.*, 1998)).

In this paper a digital electronic circuit is seen as a particular instance of a *program* in which functional units or cells are connected together to perform some computational task on binary data. In CGP a program is seen as a rectangular array of nodes. The nodes represent any operation on the data seen at its inputs. Each node may implement any convenient programming construct (e.g. `if`, `switch`, `OR`, $*$ etc.). All the inputs whether primary data, node inputs, node outputs, and program outputs are sequentially indexed by integers. The functions of the nodes are also separately sequentially indexed. The genotype is a linear string of these integers and is characterised by three parameters: the *number of columns*, the *number of rows*, and *levels-back*. The first two are merely the dimensions of the rectangular array and the last is a parameter which controls the internal connectivity. It determines how many columns of cells to the left of a particular cell may have their outputs connected to the inputs of that cell. The parameter is also applied to the program outputs. The cells and outputs are *maximally connectable*

when the *number of rows* is one and *levels-back* is equal to the *number of columns*. If however *number of rows* is one and *levels-back* is one then each cell must be connected to its immediate neighbour on the left. Cells within any particular column cannot be connected together. In this paper a particular form of CGP is adopted in which all cells are assumed to have three inputs and one output and all cell connections are feed-forward. In general CGP the cells may have multiple inputs and outputs and the numbers of these would be encoded into the genotype for the cell. Also in general primary outputs could be allowed to be treated as clocked inputs thus allowing the CGP programs to possess internal states. The genotype and the mapping process of genotype to phenotype are illustrated in Figure 6.

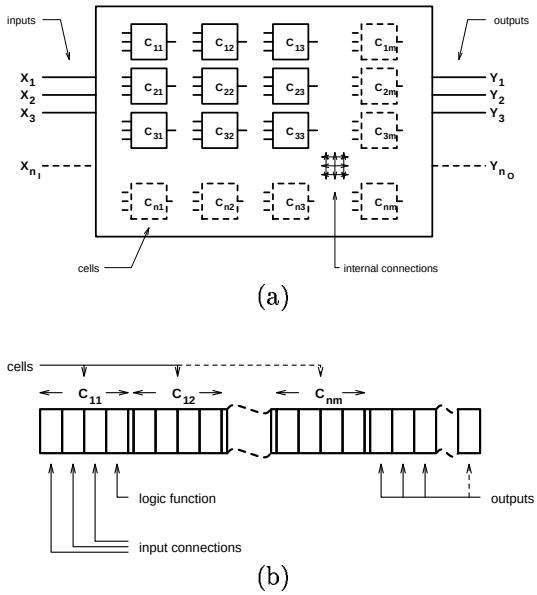


Figure 6: The genotype-phenotype mapping: (a) a $n \times m$ geometry of logic cells with n_I inputs and n_O outputs, and (b) the genotype structure of the array.

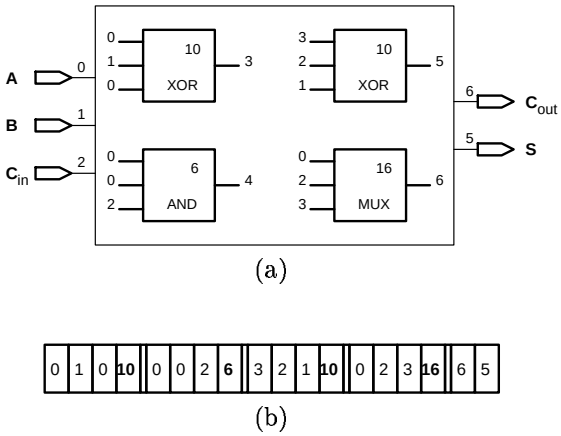
The n_I primary circuit inputs $X_1, X_2, \ldots, X_{n_I}$ are allowed to be connected to the input of any cell or any of the n_O primary circuit outputs $Y_1, Y_2, \ldots, Y_{n_O}$. The cells c_{ij} may implement any of the binary functions listed in Table 2. Functions 16 to 19 are all binary multiplexers with various inputs inverted. The multiplexer (`MUX`) implements a simple `IF-THEN` statement (i.e. `IF` $c = 0$ `THEN` a `ELSE` b). It is important to note that one can consider multiplexers to be *atomic* both formally and from an implementational point of view. It is atomic in that it is a universal logic module (Chen and Hurst, 1982) so that it can be used to represent any logic function. Also some modern FPGAs now use a multiplexer based architecture so that all two input gates are synthesised with multiplexers.

The genotype which is a list of connections and cell functions is shown in Figure 6b. In general one can think of the connections as *addresses* in data, thus provided that the function set is appropriate for a particular data type, the genotype is data independent. Note that in Table 2 only functions 16 to 19 use all three inputs and some functions

Letter	Function	Letter	Function
0	0	10	$a \oplus b$
1	1	11	$a \oplus \overline{b}$
2	a	12	$a + b$
3	b	13	$a + \overline{b}$
4	$\overline{a}$	14	$\overline{a} + b$
5	$\overline{b}$	15	$\overline{a} + \overline{b}$
6	$a \cdot b$	16	$a \cdot \overline{c} + b \cdot c$
7	$a \cdot \overline{b}$	17	$a \cdot \overline{c} + \overline{b} \cdot c$
8	$\overline{a} \cdot b$	18	$\overline{a} \cdot \overline{c} + b \cdot c$
9	$\overline{a} \cdot \overline{b}$	19	$\overline{a} \cdot \overline{c} + \overline{b} \cdot c$

Table 2: Allowed cell functions.

are actually constants with an output independent of the inputs (letters 0 and 1). Thus the genotype can contain completely redundant genes. This type of redundancy is referred to as *input redundancy*. Cells may also not have their outputs connected in the operating circuit between the primary inputs and outputs, these collections of genes (3 connections, 1 function) are also redundant. This is called *cell redundancy*. There is yet another form of redundancy called *functional redundancy* which is more typical of genetic programming. This is where a number of cells implement a function which requires less cells. To clarify the interpretation of the genotype structure depicted in Figure 6 consider the example of a one-bit adder with carry shown in Figure 7.



(a)

(b)

Figure 7: The genotype-phenotype mapping of evolved one-bit adder with carry: (a) gate array representation, and (b) genotype representation.

Figure 7 shows the genotype and phenotype for a small gate array consisting of four logic cells. The logic cells in this case have functions `XOR`, `AND`, or `MUX`. The circuit in question actually arose in an experiment reported in section 4 (Miller *et al.*, 1997) and is novel in its own right. **A**, **B**, and $\mathbf{C_{in}}$ represent the primary inputs. $\mathbf{C_{out}}$ and **S** (Sum) are the output bits of the adder. For example the upper right cell (output 5) below has

input connections 3, 2, 1. This means that the first input is connected to the output of the cell with output label 3 (upper left), the second input is connected to the primary input $\mathbf{C_{in}}$, and the third input is connected to primary input **B**. The function of each cell is expressed as the fourth gene associated with each cell (shown in bold typeface). The primary outputs of the gate array are also expressed as connections. For example, $\mathbf{C_{out}}$ is connected to the output of the cell with output label 6. It is important to emphasise that cell outputs may be re-used and when a program is used to evolve the genotypes the amount of re-use of sub-calculations is determined entirely automatically .

3.2 Calculating the Fitness of a Genotype

All functions are specified by a truth table. The fitness of a genotype is the number of correct output bits. Thus for the one-bit adder with carry seen in Figure 7 there are 8 input cases and 2 outputs, this gives 16 output bits. A fully correct circuit would have fitness 16. In practise the fitness of a circuit is calculated using 32-bit arithmetic. Thus the binary data is handled as 32-bit unsigned integers and all the operations defined in Table 2 are 32-bit operations. A truth table with 5 input variables is then represented as a single line (Poli, 1999). For example the truth table of the two-bit adder with carry is represented as:

Inputs: 4294901760 4278255360 4042322160 3435973836 2863311530
Outputs: 4277723264 3783728760 2573637990.

3.3 The Evolutionary Algorithm

The evolutionary algorithm used to produce all of the evolved circuit designs in this paper is a simple form of $(1 + \lambda)$-ES evolutionary strategy (Schwefel, 1981; Bäck *et al.*, 1991), where λ is usually about 4. Experiments were reported in (Miller, 1999) which indicated the efficiency of this approach. The algorithm is as follows:

Step 1 Randomly initialise a population of genotypes subject to constraints imposed by feed-forward nature of circuits and levels-back parameter.

Step 2 Evaluate fitness of genotypes.

Step 3 Copy fittest genotype into new population.

Step 4 Fill remaining places in population by mutated versions of fittest genotype.

Step 5 Replace old population by new and return to step 2 unless stopping criterion reached.

The mutation was defined as a percentage of the genes in a single genotype which were to be randomly mutated (subject to constraints). It was necessary to adjust the mutation rate if the genotype length was too small to prevent zero mutation. Generally speaking a mutation rate which resulted in 2 or 3 genes being changed in each genotype was found to be suitable.

A suitable population size was found by experiment using a two-bit multiplier circuit (see section 4). For each population size in the range below, 100 runs of the evolutionary algorithm were carried out. The experimental parameters were as follows:

- number of rows - 1
- number of columns - 10
- levels-back - 10
- mutation rate - 8% (3 genes)
- number of generations - up to 150,000
- gates used - $\{6, 7, 10\}$ (Table 2).

The minimum number of evaluations to obtain a 0.99 probability of success (fitness equal to 64) was calculated (Koza, 1992) and the results obtained are shown in Table 3.

Population size	Computational effort
2	148808
3	115224
4	81608
5	126015
6	100824
7	100821
8	96032
9	108036
10	108090
12	115248
14	117698
16	120080
18	145854
20	120100
25	180075
30	162180
40	216360
50	225250

Table 3: Computational effort of evolving two-bit multiplier for different population sizes.

3.4 Practical Aspects of Circuit Implementation

One of the objectives of this paper is to try to evolve as novel and efficient digital logic circuits as possible. The table of logic functions Table 2 which has been used is modelled on the resources that are available on modern FPGA platforms. The experiments described have assumed that there are no practical constraints imposed by wiring. In practise the routing of connections between components is a significant factor in the successful implementation of a circuit. Other representations of digital circuits in which the routing is explicitly taken into account have been devised (Miller and Thomson 1998b, 1998a). To improve the potential routability of circuits evolved using the

techniques described here one can adjust the levels-back parameter so that it takes much lower values. The complete investigation of the influence of this on circuit routability is a subject for further work. It was shown elsewhere (Miller and Thomson 1998b, 1998a) that the dominant factor in the evolvability of the circuits is the amount of functional resources made available, however increasing this tends to produce less efficient circuits. Conventional logic synthesis techniques minimise the symbolic representation of a circuit and then carry out *technology mapping*. This is a process of trying to rewrite the symbolic logic into a form that can be implemented with whatever gates are available on the chosen platform. To do this efficiently is a non-trivial exercise. Such a process is unnecessary when one is evolving a circuit using the gates available on the device.

3.5 Binary Circuit Symbols

Figure 8 shows the symbols used to represent logic gates in circuit diagrams. Note

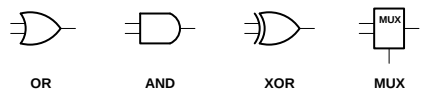


Figure 8: Binary circuit symbols.

that small circles may appear on some of the inputs and outputs of these devices, this indicates inversion.

4 Evolved Data

4.1 Interesting Problems

It is clear that the number of input combinations in a truth table grows exponentially with the number of inputs. Thus it is not practical to evolve very large truth tables ($\gg$ 25 input variables). Conventional logic synthesis techniques (see section 2) can handle hundreds of input variables. Thus the question arises what is the use of evolving truth tables by *assemble-and-test*? The answer is that one should try to evolve *interesting functions*. These are useful functions which are one of a series of increasing scale but similar function. A classic example of this are arithmetic functions, namely, binary adders and multipliers. If one can evolve a particularly efficient adder or multiplier one can use this as a building block for adders of any size. However there is yet another more interesting reason to try to evolve arithmetic functions. One can try to evolve a series of examples with increasing numbers of inputs and attempt to deduce the general design principle. If this is possible then one can potentially obtain new designs for arithmetic functions of any number of input variables. It is precisely these principles that are employed in the design of large arithmetic circuits. These are interesting problems to contrast conventional with evolved designs as one can examine the modularity of the evolved circuits. A number of key questions emerge:

1. Can more efficient designs for arithmetic functions be found by evolution?
2. Can general principles be extracted?

3. How modular are the evolved circuits?

In this section circuits are evolved for one and two-bit adders with carry, and two and three-bit multipliers. The even four-parity function is also studied as parity functions have received much attention from the genetic programming community and it is an interesting function to study as its fitness landscape changes dramatically with the choice of gates used to build it (Miller *et al.*, 2000).

4.2 One-bit Adder with Carry

The conventional one-bit adder with carry is shown in block form in Figure 9a. It adds the three binary inputs to produce a sum bit (denoted **S**) and a carry bit. The most

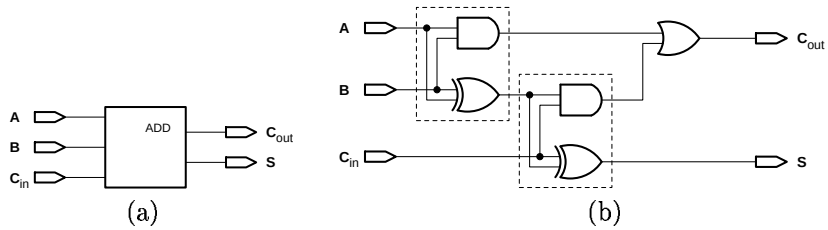


Figure 9: One-bit adder with carry: (a) block diagram, and (b) conventional circuit diagram.

efficient gate-level representation of this is shown in Figure 9b. It uses two half adders (indicated on the figure as dotted rectangles containing `AND` and `XOR` gates) and an `OR` gate. Note that the elementary addition is performed with the `XOR` gate.

The one-bit adder with carry can be easily evolved using a $(1+4)$-ES algorithm with one row and three columns of cells and allowing `XOR` and `MUX` gates only (10 and 16 in Table 2). In 100 runs of 10,000 generations (maximum) with mutation of 2 genes per genotype, 84 runs successfully evolved the function and 16 runs evolved a circuit with two output bits incorrect. The average number of generations at which the last improvement occurred was 130 generations. The minimum computational effort occurred at 5 runs of 201 generations giving a total of 5,025 evaluations for a 0.99 probability of success. The circuit which emerged is seen in Figure 10. This circuit is still generally

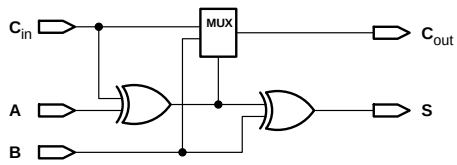


Figure 10: Evolved one-bit adder with carry.

unfamiliar in the logic design community, however certain FPGA manufacturers use this as the basic unit in their adder macros [1]. Actually evolving using 1×4 geometry

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

also produces an interesting circuit. The sum-bit is implemented as two XOR gates and the carry-bit as two MUX gates (Miller *et al.*, 1997).

4.3 Two-bit Adder with Carry

The conventional two-bit adder with carry is seen in block form in Figure 11. It carries out the function of adding together all five input bits and giving the three-bit binary number representing the sum as the output. The lowest two bits ($\mathbf{S_0}$ and $\mathbf{S_1}$) are referred to as sum-bits while the highest order bit ($\mathbf{C_{out}}$) is called the carry-bit. The adder is the simplest example of a ripple-carry adder. It is clear that any size adder can be built out of one-bit adders cascaded in this fashion. This is an example of a

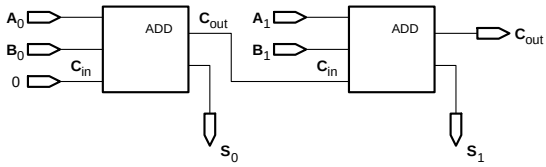


Figure 11: Two-bit ripple-carry adder.

generalisable principle.

The two-bit adder with carry can be evolved with a $(1 + 4)$-ES using 1 row and 6 columns of logic cells. 100 runs of 50,000 generations were carried out using XOR and MUX gates (10 and 16 in Table 2) with a mutation of 2 genes per genotype. 62 solutions were found which correctly implemented all 96 output bits. 12 circuits had 4 bits incorrect, 10 had 8 bits incorrect and 16 were evolved with 12 bits incorrect. The average number of generations taken to reach the last improvement was 12,581. The problem requires 22 runs of 3,501 generations to produce a 0.99 probability of success giving a computational effort of 385,110. An example of a 6 gate evolved two-bit adder is seen in Figure 12. All solutions found required 6 gates. Careful examination of this circuit and comparison

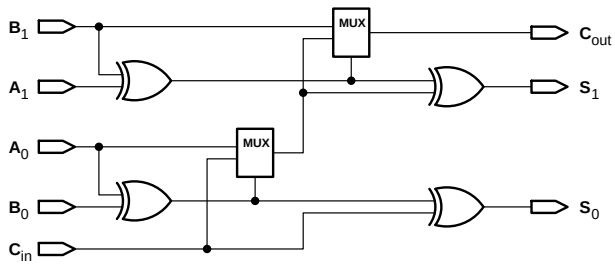


Figure 12: Evolved two-bit adder with carry.

with the evolved one-bit adder with carry (Figure 10) reveals that it is none other than the conventional two-bit adder with carry seen in Figure 11. Thus by studying two evolved solutions, the one-bit adder with carry and the two-bit adder with carry it has been possible to re-discover the well known principle of the ripple-carry adder. Thus in principle we could construct an adder of any size.

4.4 Two-bit Multiplier

The two-bit multiplier implements the binary multiplication of two two-bit numbers to produce a possible four-bit number. This is shown in Figure 13. This can be imple-

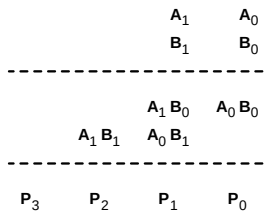


Figure 13: Multiplication of two binary numbers.

mented in block form by the cellular multiplier shown in Figure 14. The AND gates carry

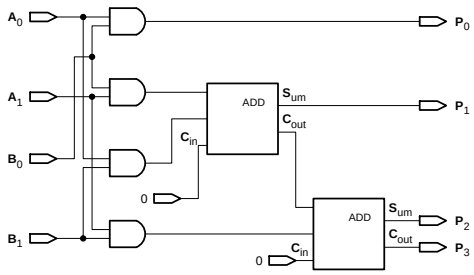


Figure 14: Two-bit cellular multiplier.

out elementary one-bit multiplication and two one-bit adders with carry are required to calculate the three most significant product bits. Since the one-bit adders have a carry-in of zero the modules can be reduced and the gate-level circuit shown in Figure 15a is obtained. In fact one of the AND gates connecting to output $\mathbf{P_3}$ can be eliminated

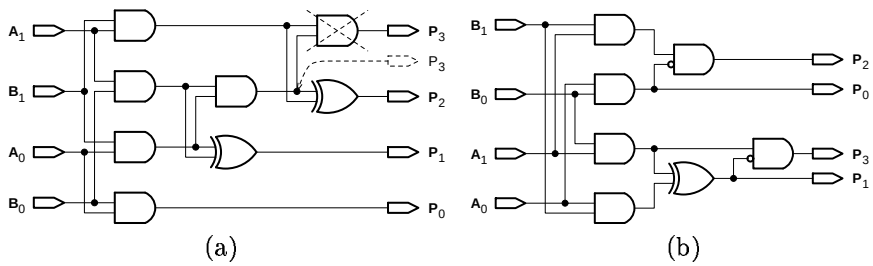


Figure 15: Most efficient (a) conventional and (b) evolved two-bit multipliers.

and thus the final most efficient conventional circuit is obtained. It requires 7 two-input gates.

When the two-bit multiplier is evolved using gates 6 to 15 (Table 2) with 100 runs of 100,000 generations and a geometry 1×7, 66 solutions which correctly implement all 64 output bits are evolved (mutation equal to 2 genes per genotype). 8 solutions had one bit incorrect, 23 two bits, and 3 circuits had 3 bits incorrect. The average number of generations to the last improvement were 17,269. The minimum computational effort was 585,045 (9 runs of 13,001 generations). All of the correct circuits required 7 gates, thus implying that 7 is a theoretical minimum. Some very interesting and surprising circuits were obtained. One circuit of particular interest is shown in Figure 15b. The circuit uses only a single `XOR` gate yet still carries out two elementary additions. It re-uses sub-calculations in a very surprising way. To create the second highest product ($\mathbf{P_2}$) it re-uses the lowest product ($\mathbf{P_0}$) and to create the highest product bit ($\mathbf{P_3}$) it re-uses the second lowest product ($\mathbf{P_1}$). The whole circuit sub-divides into two unconnected parts. The circuit is very elegant but also very counterintuitive. Comparing it with the conventional two-bit multiplier caused some consternation! It is clear that it is modelling multiplication in a very unusual way.

The choice of gates which are used to evolve circuits can have a dramatic effect on the ease of evolution. The two-bit multiplier was also evolved under the same conditions as above but with gates 6, 7, and 10. The following results were obtained. 69 circuits were correct, 5 had one bit incorrect, and 26 had two-bits wrong. The average number of generations to the last improvement was 6,771 (a lot lower than before) and the minimum computational effort was 280,035 (comprising 7 runs of 8,001 generations). Once again the circuit was evolved with gates 6, 7, 10, and 16. This proved to be a very effective choice as 94 correct circuits were obtained (under the same conditions as above) with 6 circuits having one bit wrong. The average number of generations to the last improvement was 8,863 and the minimum computational effort was 210,015 evaluations (3 runs of 14,001 generations). This implies that useful evolution was continuing longer than the previous case. Finally, an experiment was performed using only gates 16 and 17. This produced 73 correct circuits, 23 with one bit incorrect and 4 with two bits incorrect. The average number of generations to the last improvement was 11,646 and the minimum computational effort was much higher (though less than that in the first scenario) at 400,040 (8 runs of 10,001 generations). Again no correct circuits were found with less than seven gates.

4.5 Three-bit multiplier

The conventional three-bit multiplier is again modelled using the familiar process of long multiplication and is built as a cellular array of adders with the 9 elementary products being implemented with `AND` gates. The most efficient implementation of this at gate-level is shown in Figure 16. This circuit requires 30 two-input gates and 26 gates with `MUX`. Once again the familiar half and full adders are employed and there is little re-use of sub-components.

Inspired by the elegant two-bit multiplier solution found earlier (Figure 15b) the three-bit multiplier was evolved with gates 6, 7, and 10. 100 runs of 4 million generations using 1×25 geometry with levels-back equal to 25. Three solutions were found that correctly implemented all 384 output bits. The best result is shown in Figure 17. This requires 24 two-input gates (20% better than the conventional). Once again it is very strange in appearance. The lowest product is re-used twice and the second highest product is re-used once. The circuit only uses the unconventional `AND` gate with one

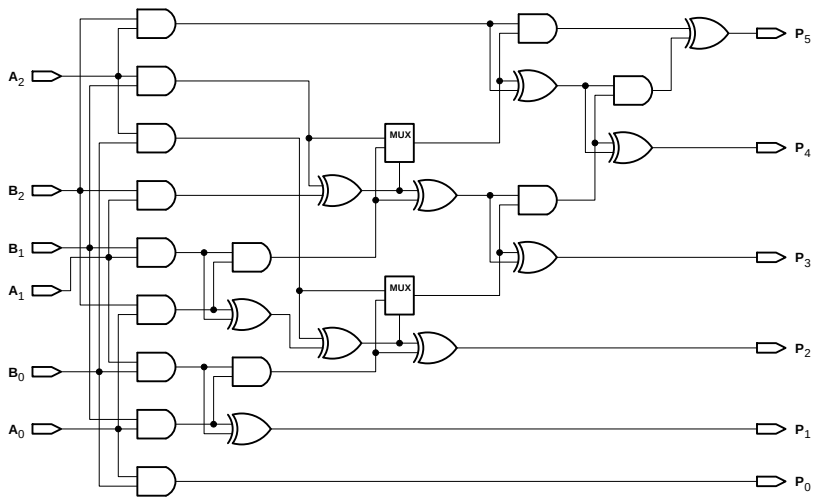


Figure 16: Most efficient conventional three-bit multiplier (30 two-input gates, 26 gates with MUX).

input inverted (gate 7) three times but it appears to make a big difference. The circuit is considerably more complex than the two-bit multiplier and it is not possible at this time to see any kind of generalisation.

When 100 runs of 10 million generations of a $(1 + 3)$-ES was used with a geometry of 1×21 and levels-back equal to 21, 3 perfect solutions were found (mutation was again 2 genes per genotype). The gates allowed were 6, 10, 16, 17. The average number of generations to the last improvement was 3,189,016. One of the circuits obtained is shown in Figure 18. This uses only 21 gates. This is again 20% more efficient in gate usage than the best conventional alternative (see Figure 16) but is 30% better than the conventional if MUX gates are counted as elementary. The circuit is very difficult to understand and it is not clear whether it consists of identifiable sub-modules which are useful in building larger systems. It departs radically from conventional principles in that it does not directly synthesise the nine elementary products of the inputs.

4.6 Even Four-parity

It is well known that even-parity functions are extremely difficult to evolve when using the logic gates AND, NAND, OR, NOR. This is because even-parity functions are most easily implemented using XNOR gates and it is difficult to synthesise XNOR function using this set. The most efficient implementation of even four-parity requires 3 XNOR gates. This is shown in Figure 19a. When one tries to automatically synthesise this circuit using only XNOR gates, it appears that random search is the most efficient method. This is evident on considering that there are only two possible fitness values (16 or 8 correct output bits). Thus we can see that the search is radically affected by the choice of gates.

Experiments were carried out to see how easily this representation could be evolved using AND, NAND, OR, NOR. 100 runs of 1 million generations of $(1 + 4)$-ES were carried out (mutation equal to 2 genes per genotype). A geometry of 1×9 was chosen with

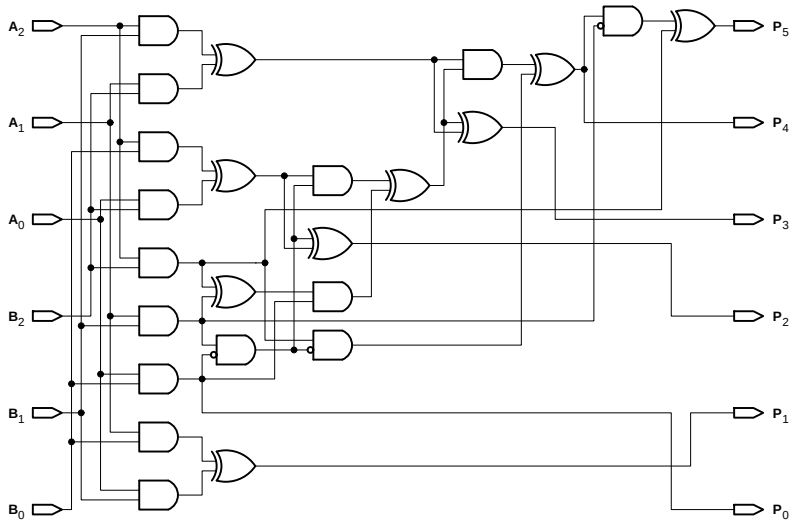


Figure 17: Evolved three-bit multiplier (24 two-input gates).

levels-back equal to 9. 15 solutions were found which had a perfect fitness of 16, 48 were found with fitness 14, and 37 with fitness 13. All of the perfect solutions were of the form shown in Figure 19b.

5 Conclusions

It has been argued in this paper that one can explore a much larger space of possible designs by employing an evolutionary algorithm together with a process of assembling and testing the designs. This has been demonstrated in the case of digital circuit design and in particular, arithmetic circuits. These circuits are modular in construction so that much larger systems can be built from smaller building blocks. This paper has examined some fundamental questions concerning the role of evolutionary algorithms as a novel methodology for design. It has tried to indicate a possible answer to the question: *Can new principles of design be discovered using artificial evolution?* The paper has shown how the principle of *ripple carry* in binary adder circuits can be deduced by studying evolved circuits for one and two-bit adders. Currently the goal of discovery of a new and efficient general principle for the construction of binary multipliers remains out of reach. However designs for three-bit multipliers have been evolved that are 20% more efficient (in gate usage) than the best conventional alternatives. Unfortunately in spite of the extraordinary speed of fitness evaluation it is time consuming to evolve *correct* three-bit multiplier circuits. One needs to examine about 50 million genotypes to achieve a high probability of success. Thus it becomes essential to understand more about the nature of the fitness landscapes. This work is undertaken in a sequel paper, Part II (Miller *et al.*, 2000).

Even with a computer that could deliver large numbers of correct designs one would have a problem of data mining the evolved circuits to extract principles. It is not feasible for an expert to study and compare hundreds of unconventional designs. An automated

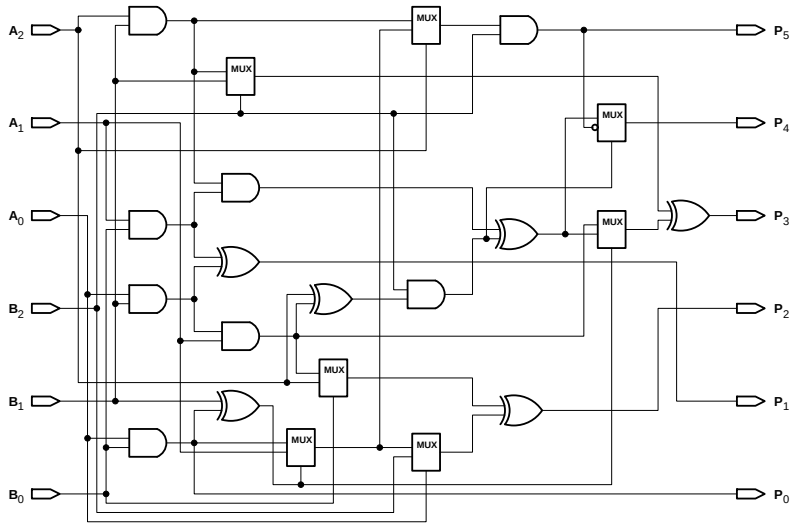


Figure 18: Evolved three-bit multiplier (21 gates = 14 two-input gates + 7 `MUX`).

approach is essential. This work was begun in (Job *et al.*, 1999) and a more detailed presentation and latest results appear in Part II in the next issue of this journal.

Acknowledgements

Dominic Job would like to thank British Telecommunications PLC for supporting this research work.

References

Akers, S. B. (1978). Binary decision diagrams. *IEEE Transactions on Computers* **C-27**, 509–516.

Atmar, J. W. (1976). *Speculation on the Evolution of Intelligence and its Possible Realisation in Machine Form.* Ph.D. thesis, New Mexico State University, Las Cruces, NM.

Bäck, T., Hoffmeister, F. and Schwefel, H. P. (1991). A survey of evolutionary strategies. In: Belew, R. and Booker, L. (eds.), *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 2–9. San Francisco, CA: Morgan Kaufmann.

Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D. (1998). *Genetic Programming: An Introduction.* San Francisco, CA: Morgan Kaufmann.

Blickle, T. (1997). *Theory of Evolutionary Algorithms and Application to System Synthesis.* Ph.D. thesis, Eidgenossiche Technische Hochschule, Zurich.

Brace, K. S., Rudell, R. L. and Bryant, R. E. (1990). Efficient implementation of a bdd package. In: *Proceedings of 27th ACM/IEEE Design Automation Conference*, pp. 40–45.

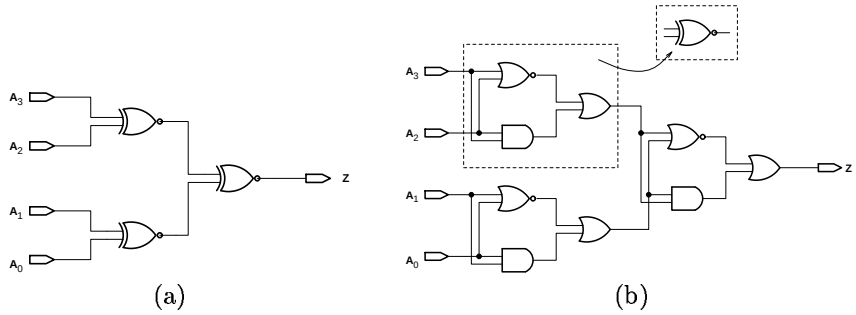


Figure 19: Two representations of the four-bit parity function with (a) gate `XNOR` and (b) gates `AND`, `OR`, and `NOR`.

Brayton, R., Hachtel, G. D. and Sangiovanni-Vincentelli, A. L. (1990). Multilevel logic synthesis. *Proceedings of the IEEE* **78**, 264–300.

Brayton, R., Rudell, R., Sangiovanni-Vincentelli, A. and Wang, A. (1987). Mis: A multiple-level optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits* **CAD-6**, 1062–1081.

Brayton, R. K., Hachtel, G. D., McMullen, C. T. and Sangiovanni-Vincentelli, A. L. (1984). *Logic Minimization Algorithms for VLSI Synthesis*. MA: Kluwer Academic Publishers.

Bryant, R. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* **C-35**, 677–691.

Bryant, R. (1991). On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers* **40**, 205–213.

Chen, X. and Hurst, S. L. (1982). A comparison of universal-logic-module realizations and their application in the synthesis of combinatorial and sequential logic networks. *IEEE Transactions on Computers* **C-31**, 140–147.

Coen, E. (1999). *The Art of Genes. How Organisms Make Themselves*. Oxford, UK: Oxford University Press.

Devadas, S. (1993). Comparing two-level and ordered binary decision diagram representations of logic functions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **12**, 722–723.

Devadas, S., Ghosh, A. and Keutzer, K. (1994). *Logic Synthesis*. New York: McGraw-Hill Inc.

Drechsler, R., Göckel, N. and Becker, B. (1996). Learning heuristics for obdd minimisation by evolutionary algorithms. In: *Parallel Problem Solving from Nature IV*, vol. 1141 of *Lecture Notes in Computer Science*, pp. 730–739. Heidelberg: Springer.

Drechsler, R., Sarabi, A., Theobald, M., Becker, B. and Perkowski, M. A. (1994a). Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. In: *Proceedings of the Design Automation Conference*, pp. 415–419.

Drechsler, R., Theobald, M. and Becker, B. (1994b). Fast ofdd based minimization of fixed polarity reed-muller expressions. In: *Proceedings of the European Design Automation Conference*, pp. 2–7.

Flockton, S. and Sheehan, K. (1999). A system for intrinsic evolution of linear and non-linear filters. In: Stoica, A., Keymeulen, D. and Lohn, J. (eds.), *Proceedings of the 1st Nasa/DoD Workshop on Evolvable Hardware*, pp. 93–100. Los Alamitos, CA: IEEE Computer Society.

Flockton, S. J. and Sheehan, K. (1998). Intrinsic circuit evolution using programmable analogue arrays. In: Sipper, M., Mange, D. and Pérez-Uribe, A. (eds.), *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, vol. 1478 of *Lecture Notes in Computer Science*, pp. 144–153. Heidelberg: Springer-Verlag.

Friedman, S. J. and Supowit, K. J. (1990). Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers* **C-39**, 710–713.

Fujita, M. and Matsunaga, Y. (1993). Variable ordering of binary decision diagrams for multilevel logic minimization. *Fujitsu Scientific and Technical Journal* **29**, 137–145.

Green, D. (1986). *Modern Logic Design*. Reading, MA: Addison-Wesley.

Grimbleby, J. B. (1995). Automatic analogue network synthesis using genetic algorithms. In: *Proceedings of the 1st International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA)*, pp. 53–58. London: IEE.

Grundy, D. L. (1994). A computational approach to vlsi analog design. *Journal of VLSI Signal Processing* **8** (1), 53–60.

Higuchi, T. and Iwata, M., eds. (1996). *Proceedings of the 1st International Conference on Evolvable Systems: From Biology to Hardware*, vol. 1259 of *Lecture Notes in Computer Science*, Berlin. Springer-Verlag.

Higuchi, T., Niwa, T., Tanaka, T., Iba, H., de Garis, H. and Furuya, T. (1993a). *Evolvable Hardware – Genetic-Based Generation of Electric Circuitry at Gate and Hardware Description Language (HDL) Levels*. Tech. Rep. 93–4, Electrotechnical Laboratory, Tsukuba, Japan.

Higuchi, T., Niwa, T., Tanaka, T., Iba, H., de Garis, H. and Furuya, T. (1993b). Evolving hardware with genetic learning: A first step towards building a darwin machine. In: Meyer, J.-A., Roitblat, H. L. and Stewart, W. (eds.), *From Animals to Animats II: Proceedings of the 2nd International Conference on Simulation of Adaptive Behaviour*, pp. 417–424. Cambridge, MA: MIT Press.

Iba, H., Iwata, M. and Higuchi, T. (1997). Machine learning approach to gate-level evolvable hardware. In: Higuchi, T. and Iwata, M. (eds.), *Proceedings of the 1st International Conference on Evolvable Systems: From Biology to Hardware*, vol. 1259 of *Lecture Notes in Computer Science*, pp. 327–343. Heidelberg: Springer-Verlag.

Jain, J., Abadir, M., Bitner, J., Fussell, D. S. and Abraham, J. A. (1992). Ibdds: An efficient functional representation for digital circuits. In: *Proceedings of the European Conference on Design Automation*, pp. 440–446.

Job, D., Shankararaman, V. and Miller, J. (1999). Hybrid ai techniques for software design. In: *Proceedings of 1999 Conference on Software Engineering & Knowledge Engineering*, pp. 315–319.

Kajitani, I., Hushino, T., Nishikawa, D., Yokoi, H., Nakaya, S., Yamauchi, T., Inuo, T., Kajihara, N., Iwata, M., Keymeulen, D. and Higuchi, T. (1998). A gate-level ehw chip: Implementing ga operations and reconfigurable hardware on a single lsi. In: Sipper, M., Mange, D. and Pérez-Uribe, A. (eds.), *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, vol. 1478 of *Lecture Notes in Computer Science*, pp. 1–12. Heidelberg: Springer-Verlag.

Kitano, H., ed. (1994). *Massively Parallel Artificial Intelligence*, Cambridge, MA. MIT Press.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

Koza, J. R., Bennett III, F. H., Andre, D. and Keane, M. A. (1996). Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In: Gero, J. S. and Sudweeks, F. (eds.), *Artificial Intelligence in Design*, pp. 151–170. MA: Kluwer Academic Publishers.

Koza, J. R., Bennett III, F. H., Andre, D. and Keane, M. A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.

Kruiskamp, M. W. and Leenaerts, D. (1995). Darwin: Cmos opamp synthesis by means of genetic algorithms. In: *Proceedings of the 32nd Design Automation Conference*, pp. 433–438. New York: Association for Computing Machinery.

Kunz, W. and Menon, P. (1994). Multi-level logic optimization by implication analysis. In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 6–13.

Lala, P. K. (1996). *Practical Digital Logic Design and Testing*. NJ: Prentice Hall.

Lee, C. Y. (1959). Representations of switching circuits by binary decision programs. *Bell Systems Technical Journal* **38**, 985–999.

Mange, D. and Tomassini, M., eds. (1998). *Bio-Inspired Computing Machines: Towards Novel Computational Architectures*. Presses Polytechniques et Universitaires Romandes.

McCluskey, E. (1956). Minimization of boolean functions. *Bell System Technical Journal* **35**, 1417–1444.

Miller, J. F. (1999). An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M., Honavar, V., Jakiela, M. and Smith, R. E. (eds.), *Proceedings of the 1st Genetic and Evolutionary Computation Conference*, vol. 2, pp. 927–936. San Francisco, CA: Morgan Kaufmann.

Miller, J. F., Job, D. and Vassilev, V. K. (2000). Principles in the evolutionary design of digital circuits — part ii. *J. Genetic Programming and Evolvable Machines* **1** (2).

Miller, J. F., Luchian, H., Bradbeer, P. V. G. and Barclay, P. J. (1994). Using a genetic algorithm for optimising fixed polarity reed-muller expansions of boolean functions. *International Journal of Electronics* **76**, 601–609.

Miller, J. F. and Thomson, P. (1998a). Aspects of digital evolution: Evolvability and architecture. In: Eiben, A. E., Bäck, T., Schoenauer, M. and Schwefel, H.-P. (eds.), *Parallel Problem Solving from Nature V*, vol. 1498 of *Lecture Notes in Computer Science*, pp. 927–936. Berlin: Springer.

Miller, J. F. and Thomson, P. (1998b). Aspects of digital evolution: Geometry and learning. In: Sipper, M., Mange, D. and Pérez-Uribe, A. (eds.), *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, vol. 1478 of *Lecture Notes in Computer Science*, pp. 25–35. Heidelberg: Springer-Verlag.

Miller, J. F., Thomson, P. and Fogarty, T. (1997). Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In: Quagliarella, D., Periaux, J., Poloni, C. and Winter, G. (eds.), *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pp. 105–131. Chechester, UK: Wiley.

Moreno, J. M. (1994). *VLSI Architectures for Evolutive Neural Models*. Ph.D. thesis, Universitat Politecnica de Catalunya, Barcelona.

Motorola (1997). *Motorola Semiconductor Technical Data: Advance Information Field Programmable Analog Array 20-cell Version MPAA020*. Motorola Inc.

Murakawa, M., Yoshizawa, S., Adachi, T., Suzuki, S., Takasuka, K., Iwata, M. and Higuchi, T. (1998). Analogue ehw chip for intermediate frequency filters. In: Sipper, M., Mange, D. and Pérez-Uribe, A. (eds.), *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, vol. 1478 of *Lecture Notes in Computer Science*, pp. 134–143. Heidelberg: Springer-Verlag.

Poli, R. (1997). Evolution of graph-like programs with parallel distributed genetic programming. In: Bäck, T. (ed.), *Proceedings of the 7th International Conference on Genetic Algorithms*, pp. 346–353. San Francisco, CA: Morgan Kaufmann.

Poli, R. (1999). Sub-machine-code gp: New results and extensions. In: Poli, R., Nordin, P., Langdon, W. B. and Fogarty, T. (eds.), *Proceedings of the 2nd European Workshop on Genetic Programming*, vol. 1598 of *Lecture Notes in Computer Science*, pp. 65–82. Heidelberg: Springer-Verlag.

Poli, R., Page, J. and Langdon, W. B. (1999). Smooth uniform crossover, sub-machine code gp and demes: A recipe for solving high-order boolean parity problems. In: Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E. (eds.), *Proceedings of the 1st Genetic and Evolutionary Computation Conference*, vol. 2, pp. 1162–1169. San Francisco, CA: Morgan Kaufmann.

Quine, W. (1952). The problem of simplifying truth functions. *American Mathematical Monthly* **59**, 521–531.

Sasao, T. (1993). *Logic Synthesis and Optimization*. MA: Kluwer Academic Publishers.

Schwefel, H.-P. (1981). *Numerical Optimization of Computer Models*. Chichester, UK: John Wiley & Sons.

Shen, A., Devadas, S. and Ghosh, A. (1995). Probabilistic manipulation of boolean functions using free boolean diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **14**, 87–97.

Sipper, M. (1997). *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*, vol. 1194 of *Lecture Notes in Computer Science*. Berlin: Springer-Verlag.

Stoica, A., Fukunaga, A., Hayworth, K. and Salazar-Lazaro, C. (1998). Evolvable hardware for space applications. In: Sipper, M., Mange, D. and Pérez-Uribe, A. (eds.), *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, vol. 1478 of *Lecture Notes in Computer Science*, pp. 166–173. Heidelberg: Springer-Verlag.

Stoica, A., Keymeulen, D., Tawel, R., Salazar-Lazaro, C. and Li, W. (1999). Evolutionary experiments with a fine-grained reconfigurable architecture for analog and digital cmos circuits. In: Stoica, A., Keymeulen, D. and Lohn, J. (eds.), *Proceedings of the 1st Nasa/DoD Workshop on Evolvable Hardware*, pp. 76–84. Los Alamitos, CA: IEEE Computer Society.

Teller, A. and Veloso, M. (1995). *PADO: Learning tree structured algorithms for orchestration into an object recognition system*. Tech. Rep. CMU-CS-95-101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Thalmann, N. M. and Thalmann, D., eds. (1994). *Artificial Life and Virtual Reality*. John Wiley.

Thompson, A. (1997). An evolved circuit, intrinsic in silicon, entwined with physics. In: Higuchi, T. and Iwata, M. (eds.), *Proceedings of the 1st International Conference on Evolvable Systems: From Biology to Hardware*, vol. 1259 of *Lecture Notes in Computer Science*, pp. 390–405. Heidelberg: Springer-Verlag.

Thompson, A. (1998). *Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution*. London: Springer-Verlag.

Thompson, A., Layzell, P. and Zebulum, R. S. (1999). Explorations in design space: Unconventional electronics design through artificial evolution. *IEEE Transactions on Evolutionary Computation* **4** (3). To appear.

Thomson, P. and Miller, J. F. (1996). Symbolic method for simplifying and-exor representations of boolean functions using a binary decision technique and a genetic algorithm. *IEE Proceedings in Computers and Digital Techniques* **143**, 151–155.

Tomassini, M. and Sanchez, E., eds. (1996). *Towards Evolvable Hardware: The Evolutionary Engineering Approach*, vol. 1062 of *Lecture Notes in Computer Science*, Berlin. Springer-Verlag.

Vassilev, V. K. (1997). An information measure of landscapes. In: Bäck, T. (ed.), *Proceedings of the 7th International Conference on Genetic Algorithms*, pp. 49–56. San Francisco, CA: Morgan Kaufmann.

Vassilev, V. K., Fogarty, T. C. and Miller, J. F. (1999). Information characteristics and the structure of landscapes. *Evolutionary Computation* To appear, available at http://www.dcs.napier.ac.uk/~vesselin/papers/ecjs.ps.gz.

Zebulum, R. S., Pacheco, M. A. and Vellasco, M. (1998). Analog circuits evolution in extrinsic and intrinsic modes. In: Sipper, M., Mange, D. and Pérez-Uribe, A. (eds.), *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, vol. 1478 of *Lecture Notes in Computer Science*, pp. 154–165. Heidelberg: Springer-Verlag.

Zebulum, R. S., Pacheco, M. A. and Vellasco, M. (1999). Artificial evolution on active filters: A case study. In: Stoica, A., Keymeulen, D. and Lohn, J. (eds.), *Proceedings of the 1st Nasa/DoD Workshop on Evolvable Hardware*, pp. 66–75. Los Alamitos, CA: IEEE Computer Society.