6

Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study

J. F. MILLER, P.THOMSON, T. FOGARTY

Dept. of Computer Studies, Napier University, 219 Colinton Road, Edinburgh, EH14 1DJ,
Email: {j.miller, p.thomson, t.fogarty}@dcs.napier.ac.uk

6. 1 I NTRODUCTION

6.1.1 Synopsis

A Genetic Algorithm is presented which is capable of evolving 100% functional arithmetic circuits. Evolved designs are presented for one-bit, two-bit adders with carry, and two and three-bit multipliers and details of the 100% correct evolution of three and four-bit adders. The largest of these circuits are the most complex digital circuits to have been designed by purely evolutionary means. The algorithm is able to re-discover conventionally optimum designs for the one-bit and two-bit adders, but more significantly is able to improve on the conventional designs for the two-bit multiplier. By analysing the history of an evolving design up to complete functionality it is possible to gain insight into evolutionary process. The technique is based on evolving the functionality and connectivity of a rectangular array of logic cells and is modelled on the resources available on the Xilinx 6216 FPGA device. Further work is described about plans to evolve the designs directly onto this device.

Keywords: Circuit design, Arithmetic Circuits, Evolvable Hardware, Evolutionary Algorithms.

6.1.2 Conventional Circuit Design versus Evolutionary Design

The design of electronic circuits is generally a complex task requiring knowledge of large collections of *domain-specific* rules. In particular the process of implementing a digital electronic circuit in hardware has typically involved transforming the original logical specification into a form suitable for the target technology (i.e. choosing the gate types), minimising the representation, optimising the representation with respect to user defined constraints (i.e. timing characteristics, fan-in/outs, etc.) and finally carrying out technology mapping onto the target device. This latter step typically involves

placing and routing of the component gates which comprise the complete design. It should be emphasised that during all these stages great care has to be taken to maintain the logical functionality of the original circuit specification.

Recent research has begun to show that it is possible to design such circuits in a radically different way. This new approach is perhaps best expressed as a *black-box* view of the problem. In this view one regards the problem of implementing the circuit as being equivalent to designing a black-box with inputs and outputs with the property that on presentation of the original input signals the desired outputs are delivered. The key new feature of this technique is that the details inside the box are encoded into chromosomes and subjected to the usual processes of evolutionary algorithms. In this technique the fitness of a particular chromosome is measured purely as the degree to which the black-box outputs behave in the desired way.

This new field of research has come to be known as *evolvable hardware*. This generic term is used to describe a number of different approaches to developing electronic circuitry by using evolutionary techniques. Where the various groups - of which there are only about four or five around the world - differ is in the area of application. Some are attempting direct evolution on to existing components, whilst others are attempting to create functionality or operating characteristics in simulation with a view to placing these on real components at a later time. For a complete review of the whole area see [SSMTP-US97].

6. 2 EVOLVING THE FUNCTIONALITY OF ELECTRONIC CIRCUITS

Up until now, most electronic systems of any complexity were created by a designer who had been trained in a particular way to understand the operation of individual electronic components, and who would, therefore, be able to use these rules of behaviour to construct larger systems from the basic parts. This was true whether the system to be created was purely analogue (responding to real-world signals), purely digital (responding to binary streams), or some combination of the two.

This method of working is somewhat constrained both by the training and experience of the designer and by the domain-specific knowledge which he may or may not possess. For example, some designers will be more expert in the analogue domain, some more expert in the digital domain. Instead, those who advocate the use of evolution to assist in the design process are not so concerned with this type of expertise, but merely seek to set up the appropriate conditions which will allow solution to evolve naturally. Figure 6.1 demonstrates the difference between the two approaches.

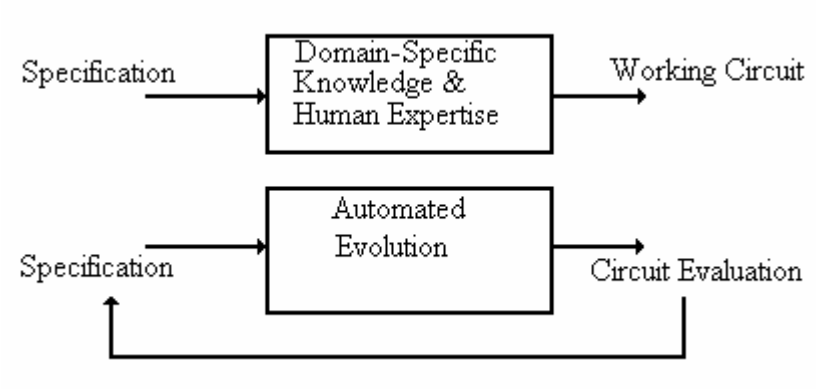

Figure 6.1 Conventional circuit design versus automated evolution

Clearly, the designer will also work through a number of iterations (*build*, *test* and *debug*) before the design is finalised. However, the finished product will only be as good as the designer's own knowledge and experience allows. The evolutionary design, on the other hand, will inevitably allow the exploration of a much richer set of possibilities. This point was well expressed by Greene [Gre97]

> "Evolutionary and adaptive search is by no means **limited** to parameter optimisation - more exciting and wide-ranging possibilities arise in the search for structure, topologies and algorithms. Designers are constrained not only by the limitations of materials and information, but also more subtly by habits of thought, conventional wisdom, and limitations of human imagination and comprehensibility. We still operate largely under assumptions of linearity, and consider only modularity and hierarchically-structured systems, though it is clear in many cases that the resulting performance is inferior to that attainable if we were able to transcend these limitations and exploit the vastly augmented design space and emergent properties attendant upon less constrained and more holistic conceptions. Automated designs may ultimately offer possibilities of escaping these constraints and accessing this richer world of possibilities, though for the present there is much useful work to be done to a more modest agenda."

The reason for the recent increase of research activity in the evolvable hardware field is probably due in no small part to the upsurge in availability of *programmable* electronic components. Unlike traditional components, these devices have no fixed operation or functionality when first obtained. Instead it is the responsibility of the user/designer to decide - via the appropriate programming - what that functionality should be either during or after implementation within a given system. In many instances, these devices, once programmed, are even then not dedicated to that particular operational characteristic, but may afterwards be re-programmed to adopt yet another different functionality. Typical amongst this range of products are the *field-programmable gate array* or FPGA devices manufactured by Xilinx. These devices are primarily for digital design, however analogue based *field-programmable analogue arrays* are beginning to become available[BM96, KK96, PH96].

One of the benefits of any device which is re-programmable is that it allows the possibility of circuit evolution. This is especially true of the Xilinx 6000 series FPGAs, where the function of the device may be re-programmed in-circuit. These devices are known a fine-grained FPGAs, which means that each is made up of a large (typically square) array of very simple digital logic cells. This is as opposed to the better known Xilinx 3000 and 4000 components which have much larger, and consequently more complex cells (which are known as *Configurable Logic Blocks* or *CLBs*).

One of the reasons for the 6000 family's appeal is that it becomes possible to re-configure the internal structure of this device's cells in a simple manner and in such a way that the device cannot be damaged. Consequently it is possible to place a random configuration (in the form of a serial bit stream) directly on to the device - the CLB based devices do not allow this.

The direct evolution of configuration bit streams is the approach taken by Thompson [Tho96]. He evolved a circuit which could discriminate between square wave input signals of 1 and 10 kHz. He used a GA to create bit-strings to define the configuration of a 10 x 10 array of cells on the Xilinx 6216 FPGA. Each chromosome consisted of 1800 bits (18 per cell). These bit-strings configured both the digital functionality of each internal cell, and the switching paths of the routing between cells (the connectivity).

Populations of bit-strings are fed to the device to configure it, and then behaviour at selected outputs is monitored in response to stimuli at appropriate inputs in order to assess fitness. The fitness is then measured as any particular configuration's ability (as determined by the chromosome) to handle some user defined behaviour at the device outputs.

This *intrinsic* evolution - as it has become known - does have one major drawback: the process is totally unconstrained. This means that any internal configuration that is possible is allowed, and , as a result, solutions tend to involve the underlying physical (non-digital) properties of the FPGA device. This creates a problem when it comes to repeating the design on a different physical device. In other words, the design which is evolved is specific to the particular device on which the fitness was measured. If an attempt is made to implement exactly the same design on another physical device, the circuit behaviour is invariably not the same as on the original. It has also been found that the actual circuit behaviour is also dependent upon the operating temperature of the device. Thompson is currently working on ways to overcome this particular difficulty.

As an alternative to the method adopted by Thompson, some researchers have taken a Genetic Programming approach to evolving circuit solutions. Koza [Koz92] evolved *programs* which described a three-input multiplexer and a two-bit binary adder without carry rather than as a list of interconnecting logical cells. A drawback of this approach is that the representation would have to undergo placement and routing before it could be implemented onto the target device. Iba et al. [IIH96], on the other hand, attempt to evolve both the functionality and connectivity of interconnected AND, OR and NOT gates for intended use on a *programmable logic array* (PLA) device (this is a device which has an input field of AND gates whose outputs are fed to an output field of OR gates to form sum-of-product logic solutions). Using this technology, some of the circuits that were created were invalid due to the presence of short-circuits. This type of problem could not arise using Xilinx 6000 FPGAs because of the internal structure of the device's cells. Additionally, with the 6000 family, it is not necessary to configure (program) the entire array of cells on the device in order to alter the part's functionality. Instead, individual cells may be identified and re-programmed as necessary. Attempts to use gate-level only functionality in the evolutionary process has become known as *extrinsic* evolution - as opposed to the intrinsic evolution of Thompson's method where actual physical properties of the device are involved.

Another approach taken to evolving hardware had been the *cellular automaton* approach of the Swiss group at EPFL [GSMSST96]. A cellular automaton consists of an array of cells, each of which can be in one of a fixed number of states. The states of all the cells are updated synchronously according to a local and identical rule. The new state of a cell is determined by the former states of its neighbours according to a rule table. In a non-uniform cellular automaton the rule table may not be identical for all cells. They implemented an evolving, one-dimensional non-uniform cellular automaton consisting of 56 binary-state cells. Each of the cells contained a genome which represented its rule table. The genomes were initialised at random and thereafter subjected to an evolutionary process. The task chosen was for the states of all the cells to oscillate between being 'off' or 'on' after a fixed number of time steps after the states of the cells having been initialised at random. They found that their system was able to accomplish this task despite the fact that the task was globally defined and the cells behaviour was determined locally. Notably, they built the entire system in hardware.

Koza et al. [KABK97] has shown that it is possible to produce designs for quite complex analogue electronic circuits, namely: low-distortion operational amplifier, lowpass, crossover and asymmetric bandpass filters and a cube-root circuit. The starting point in these designs was typically a simple embryonic electrical circuit containing fixed parts appropriate to the problem and certain wires capable of subsequent modification. An electrical circuit is progressively developed by applying the functions in a circuit-constructing program tree to the modifiable wires of the embryonic circuit (and subsequently the modifiable wires and components). The functions in the circuit-constructing program trees included (a) connection-modifying functions which changed the topology of the circuit, (b) functions which insert components into the circuit and (c) arithmetic-performing functions which modified the numerical value of components. The behaviour of each circuit was evaluated using the SPICE simulation program rather than producing a real circuit who's properties were evaluated.

6.3 DISCOVERING NEW DESIGNS FOR ARITHMETIC CIRCUITS BY EVOLVING THE FUNCTIONALITY AND CONNECTIVITY OF A RECTANGULAR ARRAY OF LOGIC CELLS

6. 3.1 Introduction

The starting point in this technique is to consider, for each potential design, a geometry (of a fixed size array) of *uncommitted* logic cells that exist between a set of desired inputs and outputs. The uncommitted logic cells are typical of the resource provided on the Xilinx XC6216 FPGA part under consideration. An uncommitted logic cell refers to a two-input, single-output logic module with no fixed functionality. The functionality may then be chosen, at the implementation time, to be any two-input variable logic function.

There are two aspects required to define any combinational logic network. The first is the cell-level functionality and the second is the inter-connectivity of the cells between circuit inputs and outputs. A *chromosome* is defined as a set of interconnections and gate level functionality for these cells from outputs back toward the inputs based upon a numbered rectangular grid of the cells themselves, as in Figure 6.2. This procedure was carried out in such a way that *all individual cell inputs could only be connected to cell outputs which were of a lower number* within this scheme. This is important because it eliminates any possibility of *feedback* connections which would give rise to non-combinational time-dependent behaviour. The inputs that are made available are logic '0', logic '1', all primary inputs and primary inputs inverted. To illustrate this consider a 3 x 3 array of logic cells between two required primary inputs and two required outputs.

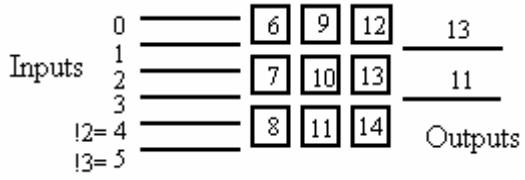

Figure 6.2 A 3 x 3 geometry of uncommitted logic cells with inputs, outputs and netlist numbering.

The inputs 0 and 1 are standard within the chromosome, and represent the fixed values, logic '0' and logic '1' respectively. The inputs (two in this case) are numbered 2 and 3, with 2 being the most significant. The lines 4 and 5 represent the inverted inputs 2 and 3 respectively. The logic cells which form the array are numbered column-wise from 6 to 14. The outputs are numbered 13 and 11, meaning that the most significant output is connected to the output of cell 13 and the least significant output is connected to the output of cell 11. These integer values, whilst denoting the physical location of each input, cell or output within the structure, now also represent connections or *routes* between the various points. In other words, this numbering system may be used to define a netlist for any combinational circuit. Thus, a chromosome is merely a list of these integer values, where the *position* on the list tells us the cell or output which is being referred to, while the value tells us the connection (of cell or input) to which that point is connected, and the cells functionality.

Each of the logic cells is capable of assuming the functionality of any two-input logic gate, or, alternatively a 2-1 *multiplexer* (MUX) with single control input. In the geometry shown in Figure 6.2 a sample chromosome is shown below:

0 2 -1 1 3 -5 2 4 3 2 6 7 0 8 -10 7 8 -4 6 11 9 6 4 -9 2 11 7 13 11

Figure 6.3 A typical netlist chromosome for the 3 x 3 geometry of Figure 6.2

Notice, in this arrangement that the chromosome is split up into groups of three integers. The first two values relate to the connections of the two inputs to the gate or MUX. The third value may either be positive - in which case it is taken to represent the control input of a MUX - or negative - in which case it is taken to represent a two-input gate, where the modulus of the number indicates the function according to Table 6.1 below. The first input to the cell is called A and the second input called B for convenience. For the logic operations, the C language symbols are used: (i) & for AND, (ii) | for OR, (iii) ^ for exclusive-OR, and (iv) ! for NOT. There are only 12 entries on this table out of a possible 16 as 4 of the combinations: (i) all zeroes, (ii) all ones, (iii) input A passed straight through, and (iv) input B passed straight through are considered trivial - because these are already included among the possible input combinations, and they do not affect subsequent network connections in cascade.

Table 6.1 Cell gate functionality according to negative gene value in chromosome.

Gene value	Gate Function
-1	A & B
-2	A & !B
-3	!A & B
-4	A ^ B
-5	A | B
-6	!A & !B
-7	!A ^ B
-8	!A
-9	A | !B
-10	!B
-11	!A | B
-12	!A | !B

This means that the first cell with output number 6 and characterised by the triple {0, 2, -1} has its A input connected to '0', its B input connected to input 2, and since the third value is -1, the cell is an AND gate (thus in this case always produces a logical output of 0). Picking out the cell who's output is labelled 9, which is characterised by the triple {2, 6, 7}, it can be seen that its A input is connected to input 2 and its B input is connected to the output of cell 6, while since the third number is positive the cell is a MUX with control input connected to the output of cell 7.

For many circuits there will be cells that are not actually connected to any of the outputs, as indeed is the case for cells with outputs 9, 10, 12 and 14 in the example above. These are redundant for the circuit they define, and may be removed when the chromosome is analysed. There are other forms of cell redundancy possible (e.g. the cell with output 6 above is redundant since its output is locked at logical 0. There are even subtler forms of redundancy which cause a particular cells output to be stuck at either one or zero even though the inputs to the cell are not fixed. This happens when the inputs change *together.*

6.3.2 A chromosome representation which preserves the feed-forward, time-independent property.

The netlist structure of the chromosome has a property imposed on it when randomly initialised and when mutated which ensures that the circuit corresponding to the chromosome is a valid combinational circuit (i.e. is feed-forward with all inputs specified). It also ensures that after crossover the new chromosomes will also have this property preserved. This is achieved in the following way. Let *n*

denote the number of cells in the rectangular array and n_r, and n_c be the number of rows and columns of cells in the rectangular array respectively. Let n_i represent the number of inputs and *l* represent the number of levels back (on the left) to which a cell in a particular column *c* or an output may be connected. Define *V(c)* to be the set of integers *v* such that,

$$c > l, \quad 2n_i + 2 + (c-l)n_r \le v \le cn_r + 2n_i + 1$$
$$c \le l, \quad 0 \le v \le cn_r + 2n_i + 1$$

and *G* to be the set of integers {-1, …, -12}.

The gene value *g(x)* at position *x* (measured from the left and starting at 0) is chosen as follows:

Cell inputs

$0 \le x < n$, $g(x) \in V(c)$ *if* $(x+1) \bmod 3 = 0,1$

$g(x) \in V(c) \cup G$ *if* $(x+1) \bmod 3 = 2$

Outputs

$n \le x$ $g(x) \in V(n_c)$

For example, for the first column of cells in the chosen geometry, the inputs to the actual MUX/gates may only be connected to the actual circuit inputs (including '0','1' and the inverted inputs). However provided that the levels-back parameter *l* is greater than 1 the second column inputs may be connected to both the circuit inputs and the outputs of the first column of cells. If *l* had been chosen to be 1 then the cells in column 2 could only have their inputs connected to the outputs of the cells in column 1. Thus in general the possible range of values for the genes increases with each new column. This has the effect whereby genes in the higher positions within the chromosomes have a greater range of values to choose from, provided *l* is chosen to allow this. It is precisely because of this scheme that the chromosomes will always represent feed-forward, combinational circuits. Decreasing *l* has the effect of reducing the number of possible circuit solutions that may be found, but also should ease any potential routing congestion which may ensue when implementing solutions on the actual FPGA part. This is important, as it is often this type of routing difficulty which prevents optimised circuits from being implemented on these devices.

6.3.3 Properties of the Genetic Algorithm

The test of whether the evolved circuits perform the desired logic translation of inputs to outputs is achieved by running *all* test inputs through the network, and comparing the results with the desired functionality in a bit-wise fashion. A PLA file (truth table) contains the target function, and this is used as a basis for comparison. The percentage of total correct outputs in response to appropriate inputs is then used as the *fitness* measure for the genetic algorithm. In other words, the nearer the evolved circuit comes to performing with desired functionality, the fitter it is deemed to be. Size two *tournament selection* was used for parent selection but with the modification that the winner of the tournament was only chosen with a certain probability. This probability was termed the *tournament discriminator*. If this probability is set to unity, then the tournament becomes the standard tournament sized two selection mechanism. However, if this probability is set to less than unity, then the *selection pressure* on the population is reduced. If the probability equals 0.5, then the selection pressure on the population becomes zero, and the GA reduces to a random search. Intermediate probability values lead to a less rapid drop in genetic diversity in the population and so reduces the onset of convergence. In experiments it was found that setting the tournament discriminator to 0.7 gave the most favourable results.

Initially large populations were chosen (between 1000 and 2000). However it was later discovered that relatively small populations performed better. The breeding rate defined as the percentage of population subjected to crossover and replaced by children, was found to be best set at 100%. The percentage of all genes in the population which were mutated before breeding was chosen to be 5%. Experiments confirmed this as a good figure. Contrary to initial expectations it was found to be beneficial to employ elitism. Uniform crossover was used.

6.3.4 Evolving small populations over very large numbers of generations gives the best results

Table 6.2 below shows the results of some experiments in attempting to correctly evolve the functionality of a two-bit adder with carry. This is already a quite complex circuit with five inputs and three outputs and requiring 32 input and output conditions for full specification. In all of the experiments the total number of chromosome evaluations (accumulated over all runs) is fixed at 12,000,000. In the table the fourth column lists the mean fitness of the best individuals' averages over the runs and the fifth column their corresponding standard deviations. The sixth column lists the percentage of runs which terminated with a chromosome which represented a 100% correct two-bit adder circuit.

Table 6.2 Performances of runs of GA with 12,000,000 evaluations.

Population	#generations	#runs	Mean fitness	Standard deviation	Percentage 100% cases
10	100,000	12	92.71	5.67	33.3
15	80,000	10	97.29	2.52	40.0
30	40,000	10	96.14	4.52	50.0
60	5,000	40	88.46	3.81	2.5
60	10,000	20	93.28	5.23	30.0
60	20,000	10	94.06	4.61	30.0
60	40,000	5	94.17	5.65	40.0
120	5,000	20	89.63	4.50	10.0
240	2,500	20	87.76	4.13	5.0

It is clear from these figures that a population size of between 15 and 60 with a large number of generations (between 40,000 and 80,000) performs best, and quite markedly better than the larger populations. It might be inferred from this that a great deal of local searching is going on and that perhaps 100% breeding is not very important. Experiments have shown however that performance is markedly worse when the breeding rate is lowered.

6.4 EVOLVED CIRCUIT DESIGNS FOR ARITHMETIC CIRCUITS

6.4.1 Early aspects of chromosome representation

In the early stages of this research a simplified chromosome was used. Two genetic algorithms were written, one which used only the two inputs gate types of Table 6.1, and the other which allowed only single-control multiplexers. Experiments showed that it was easy to evolve the sum component of the one-bit full adder using two-input gate combinations, but difficult to evolve the carry. Conversely, with the multiplexer it was easy to evolve the carry component, but difficult to evolve the sum. This behaviour is not all that surprising as for instance, it is unlikely that the GA would be able to correctly synthesise a multiplexer to act as an exclusive-OR gate simply because to do this requires that the two

inputs to the multiplexer are inversions of one another. These findings led to the development of a chromosome representation which allow both types of logic gates. It became easier to evolve 100% functionality immediately after having utilised this more complicated representation, despite the fact that the search space had been greatly expanded by this change. This finding tends to imply that in attempting to evolve digital circuits it is not the size of the search space which is so important but rather a chromosome representation which allows more *routes* to the desired solution. Clearly the more complex representations possess greater redundancy as evidenced by the fact that on many occasions evolved chromosomes have achieved 100% functionality by using *multiplexers* as ex-OR, AND, and OR gates.

6.4.2 The one-bit adder with carry

Evolving a fully functional one-bit adder using a geometry of 2 rows and 3 columns proved to relatively easy and the design shown in Figure 6.4 was obtained.

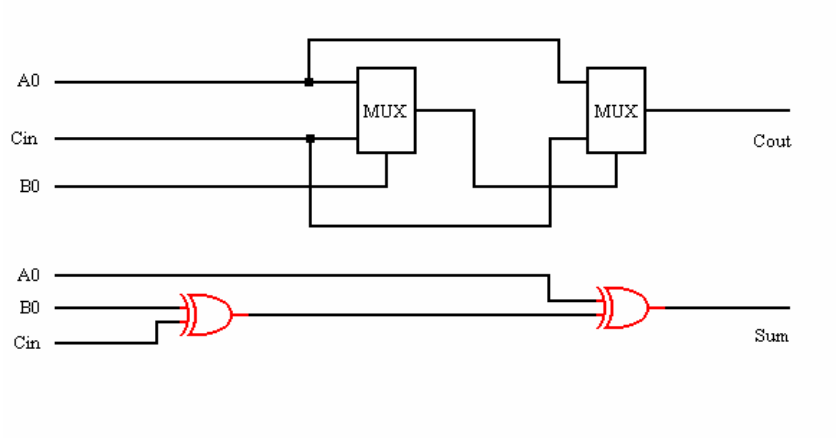

Figure 6.4 The first evolved one-bit full adder with carry

The sum component in this circuit is the familiar sum-bit circuit of conventional adders, the carry circuit is a little different and came as a surprise, as in terms of cell counts, the whole adder is more efficient than the five gate conventional one-bit full adder, but this doesn't employ multiplexers.

When the geometry was constrained to 2 x 2 and the GA run twenty times (over 2000 generations) the design shown in Figure 6.5 was obtained. This was a gratifying result to obtain as it is clear that this design is an optimum solution. Evolving the one-bit adder was easier to do on a larger geometry but resulted in a less efficient circuit. Actually the ease with which the GA was able to discover 100% functional solutions was intimately related to the size of the geometry. Choosing too small a geometry ran the risk that no 100% solutions could be found because it was physically impossible to build the required functionality with that few gates. While using too large a geometry simply gave the GA too many possibilities to work with and it struggled to find the fully functional solutions.

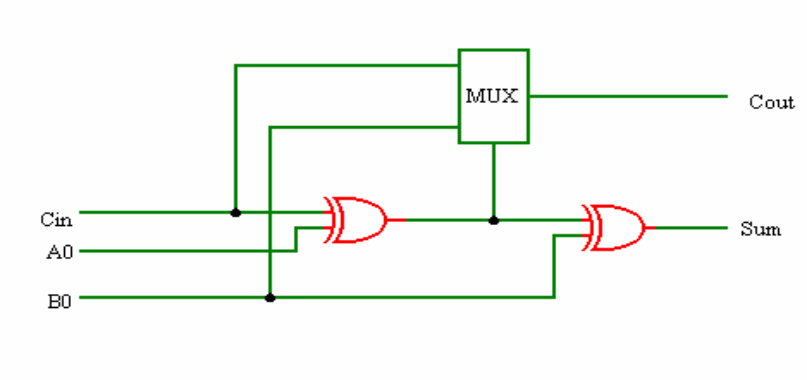

Figure 6.5 The evolved optimum one-bit full adder with carry

6.4.3 The two-bit full adder with carry

The evolution of a design for the two-bit full adder with carry was achieved in two distinct ways. The first with the sum and carry components separated (Figures 6.6 and 6.7) and the second with these two combined (Figure 6.8).

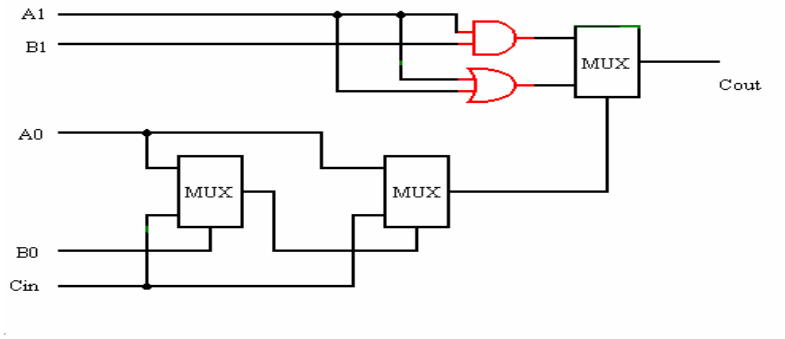

Figure 6.6 The carry component of the first evolved two-bit full adder with carry

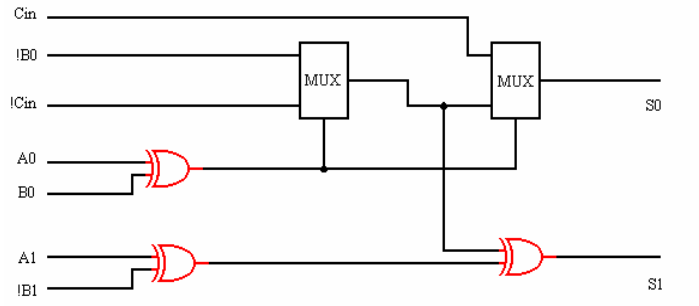

Figure 6.7 The sum component of the first evolved two-bit full adder with carry

Notice that in the first of these designs the carry circuit uses the one-bit full adder carry circuit of Figure 6.4 to control whether the AND or the OR of the two most significant data bits is to be propagated to the output carry. This is interesting as it shows that the evolutionary process is extracting principles

which it can re-use when constructing higher order circuits of the same class. It is precisely this sort of property that may mean that it possible to generalise from a number of smaller examples produced by evolutionary means. Thus holding out the tantalising possibility that completely new designs of arithmetic circuits may be possible which have not been hitherto obtained by human designers.

In the second of the two designs for the two-bit adder with carry presented below, it becomes clear how evolutionary process is generalising and producing a hierarchical sequence of designs that the human designer can subsequently examine to determine the general rule.

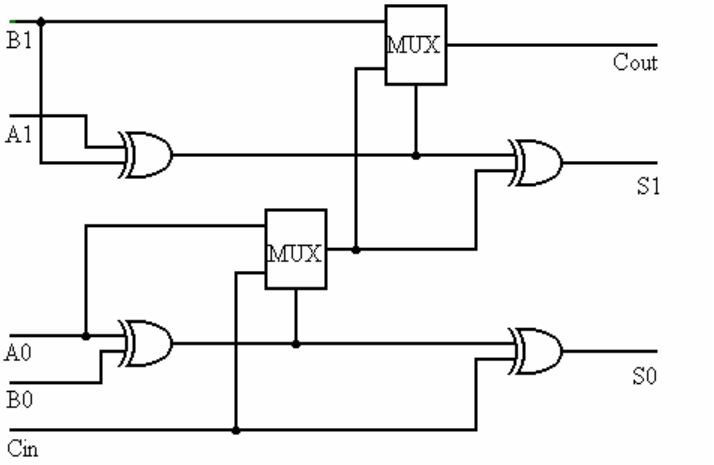

Figure 6.8 The second evolved two-bit full adder with carry.

This design is none other than the ripple-carry adder shown below.

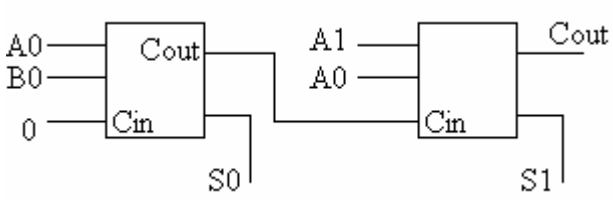

Figure 6.9 The two-bit ripple-carry adder.

Where the boxes represent the one-bit adder circuit seen in Figure 6.5. The circuit was evolved using a 3 x 3 geometry and the experiment parameter settings were as follows:

population size = 50, number of generations = 50,000, number of runs = 20, breeding rate = 100%, mutation rate = 5%, elitism, levels back = 2

In this run of 20, 5 solutions were obtained which were 100% functional. The chromosomes for these solutions, and the generation at which the 100% solution appeared are listed below, and the number of cells required are listed in Table 6.3 . The first chromosome corresponds to Figure 6.8.

Table 6.3 100% functional 2-bit adder chromosomes

Generation	#cells	Chromosome
28766	6	7 2 9 11 6 8 5 0 -10 3 5 13 11 13 -7 9 11 -10 17 15 12 14 13 -7 15 12 -4 18 20 19
11853	8	6 5 -7 3 0 8 2 4 11 8 3 12 13 6 12 7 9 -4 15 16 13 17 16 -4 14 16 17 20 19 15
1644	8	4 9 7 8 6 -7 5 3 -4 5 10 13 2 4 7 3 6 14 0 16 14 17 12 -7 17 16 12 20 19 15
15340	7	1 11 -12 8 5 -4 7 9 -4 8 14 -8 4 5 14 6 3 13 17 16 15 12 13 -7 14 15 17 18 20 19
7532	8	3 8 11 2 7 9 4 2 9 5 3 12 12 10 -4 9 2 -7 12 15 -6 15 14 13 17 13 15 19 20 16

It should be noted that after a chromosome has been obtained it is subject to analysis and any redundant gates are removed. This may occur when cells are not involved in a circuit which connects to an output or when because of a certain input a gate is made redundant as in the case of a input of 0 to and AND gate or an input of 1 to an OR gate. The recognition of these sorts of redundancies may indeed have a 'knock-on' effect where because one gate is redundant, another gate connected to it may also become redundant. A program is run after the genetic algorithm which checks for all these conditions and reduces the number of cells in the circuit.

Higher order adders have been evolved, none of which proved to be more efficient than a cellular based design. It should be stressed though, that at the time, the efficiency of the resulting circuit was not the objective, only the efficiency of production of 100% functional solutions. Correct three-bit adders have been evolved, this happened twice (the best required 12 cells and the other 15 cells) in 20 runs of 90,000 generations using a 4 x 5 geometry of cells. The best solution required 47,852 generations and the other 30,260. The four-bit adder was evolved also, the carry circuit and sum circuit being separated prior to evolution. The carry circuit required 19 cells and was evolved on an 8 x 8 geometry (clearly too large) with a population of 50. It was the only 100% result achieved in 20 runs of 80,000 generations, though it only required 40,878 generations. The sum circuit was evolved on a 9 x 9 geometry, with a population size of 80. It was achieved at generation 51,048 in a batch of 20 runs of 60,000 generations. The solution required 39 cells.

6.4.4 The two-bit full multiplier

Conventional two-bit multipliers

A two-bit multiplier multiplies the binary numbers (A2,A1) by (B2,B1) to produce the four-bit binary number (P4,P3,P2,P1) where A2,B2,P4 are the most significant bits. Modelling the multiplication process on the familiar long-multiplication method

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

one arrives at the conventional design for a two-bit multiplier in the form of a cellular array [Alm94] using one-bit adder circuits and AND gates for the one-bit partial products.

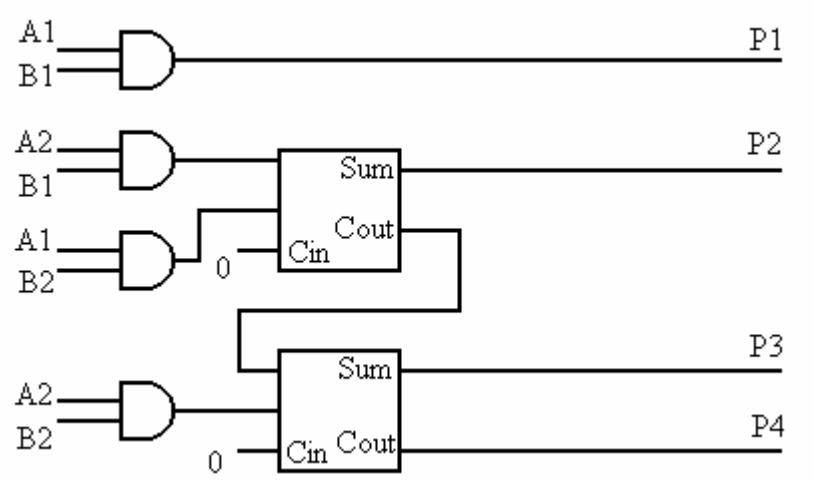

Figure 6.10 The 2-bit cellular multiplier circuit

Due to the carry-in on the two one-bit adder modules being 0, the modules reduce to 1-bit adders without carry-in (often known as half-adders),

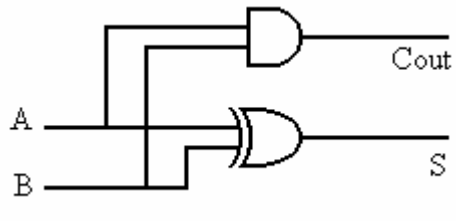

Figure 6.11 1-bit adder (no carry-in)

Consequently the two-bit cellular multiplier looks like (Figure 6.12)

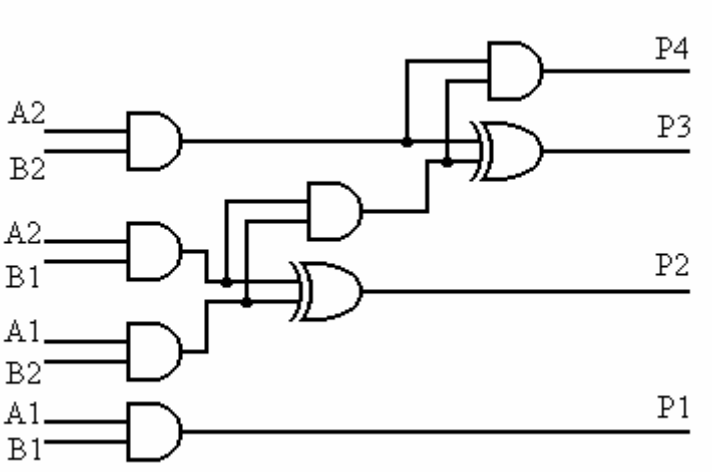

Figure 6.12 Gate layout for two-bit cellular multiplier

In many technologies the speed of multiple-input gates is almost the same as the speed of two-input gates. In such cases it is advantageous to build a multiplier which exploits this fact. This can be simply done by minimising using the rules of Boolean logic (using Karnaugh maps) the original truth table for the two-bit multiplier [Alm94]. When this is carried out one obtains a circuit requiring 10 gates comprising 1 4-input AND gate, 6 3-input AND gates, 1 2-input AND gate, 1 4-input OR gate, and 1 2-input OR gate. In a technology such as provided in the Xilinx 6000 family FPGAs which requires that *all* logic gates are synthesised from 2-input, 1-control multiplexers, using multiple-input gate designs is pointless as it confers no speed advantage. Indeed this argument carries over to "faster" adder designs, such as carry-lookahead adders. These designs again confer no speed advantage over a conventional ripple-carry adder. Thus the most efficient known 2-bit multiplier for use on a technology such as the Xilinx 6000 family can be seen in Figure 6.12, and it requires 8 gates or cells. Note that each elementary addition involved in the multiplication process is accomplished using an ex-OR gate.

Evolved two-bit multipliers

Figures 6.13 - 6.15 show the three most efficient circuits which were obtained in 50 runs of the GA with a population size of 80, each run terminating at 60,000 generations (if 100% hadn't been achieved). The remaining parameters were the same as stated in the section on the two-bit adder. In the 50 runs 32 runs produced designs which were 100% functional. The cell counts for these 32 are shown below:

Table 6.4 Numbers of cells required and frequency of occurrences for evolved two-bit multiplier circuit using a 3 x 4 array of cells

#cells	7	8	9	10	11	12
frequency	3	5	8	10	5	1

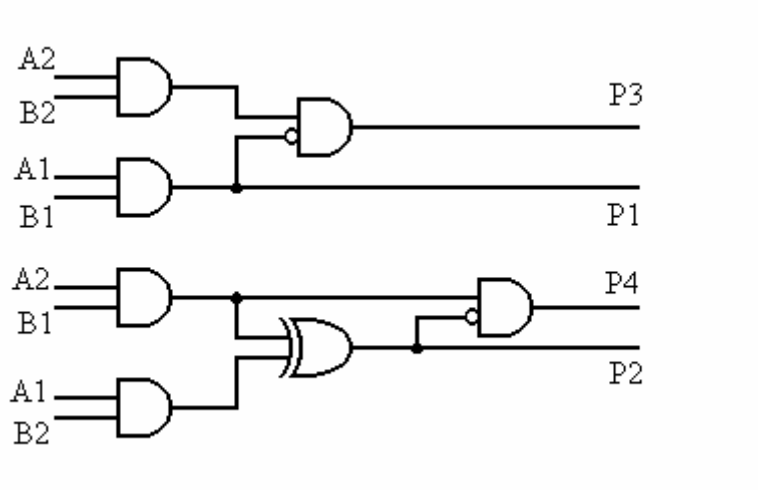

Figure 6. 13 Evolved two-bit multiplier A
(The small circles represent inversion)

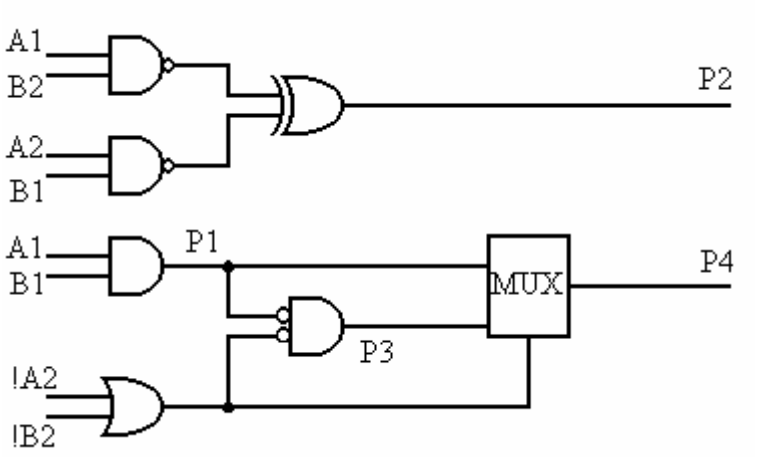

Figure 6.14 Evolved two-bit multiplier B

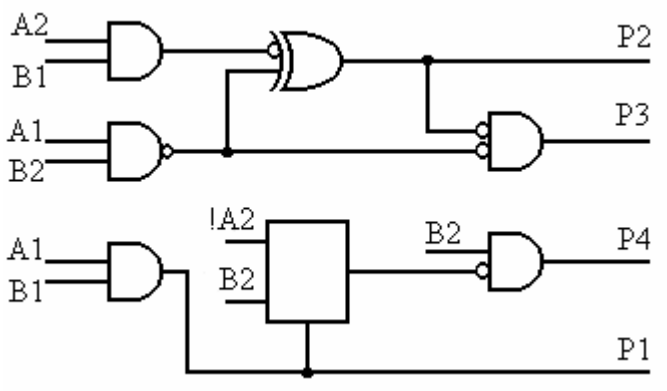

Figure 6.15 Evolved two-bit multiplier C

Firstly, note that all these designs require only 7 cells, hence they are more efficient than the designs produced by human designers! It is interesting to note that in all circuits the multiplier has been synthesised as two separate sub-circuits. In multiplier A one sub-circuit produces P1 and P3 and the other produces P2 and P4. While in circuit B, one sub-circuit produces P2 only and the other produces P1, P3 and P4. Incidentally note also that the sub-circuit in B which produces P2 is equivalent to the circuit in the conventional cellular multiplier in figure 11, as the two inversions on the AND gate outputs cancel out at the ex-OR gate. In multiplier C, one sub-circuit produces P2 and P3 while the other sub-circuit produces P1 and P4. One is tempted to ask: *How many intrinsically different 7-cell multiplier circuits are there?* Also: *Is 7 cells the theoretical minimum?* This division of the multiplier into sub-circuits is interesting and quite subtle. Take for example the way in which P3 is calculated in circuit A. What have A2,B2 and A1,B1 got to do with P3? This is highly non-intuitive. Also note that P2 is calculated within circuit A in an identical way to the conventional design but then P4 uses this circuit in an extraordinary way! It is clear that evolution is coming up with very novel ways of building these circuits and one can only wonder:

Are there principles evident in these designs which can be extracted and which could lead to some natural generalisation of one of these circuits which would enable the construction of higher order multipliers?

It is the contention here that an affirmative answer to the above question may indeed be possible, though it would certainly be helpful to obtain a number of efficient designs for the three-bit multiplier! A cellular array form of the three-bit multiplier would require 20 cells. A very large number of attempts to directly evolve the three-bit multiplier have been made and so far only one 100% functional solution has been obtained. This was evolved using a 5 x 7 geometry and required 28 cells. The solution shown below in Figure 6.16, occurred at generation 156,885 on a batch of ten runs of 200,000 generations.

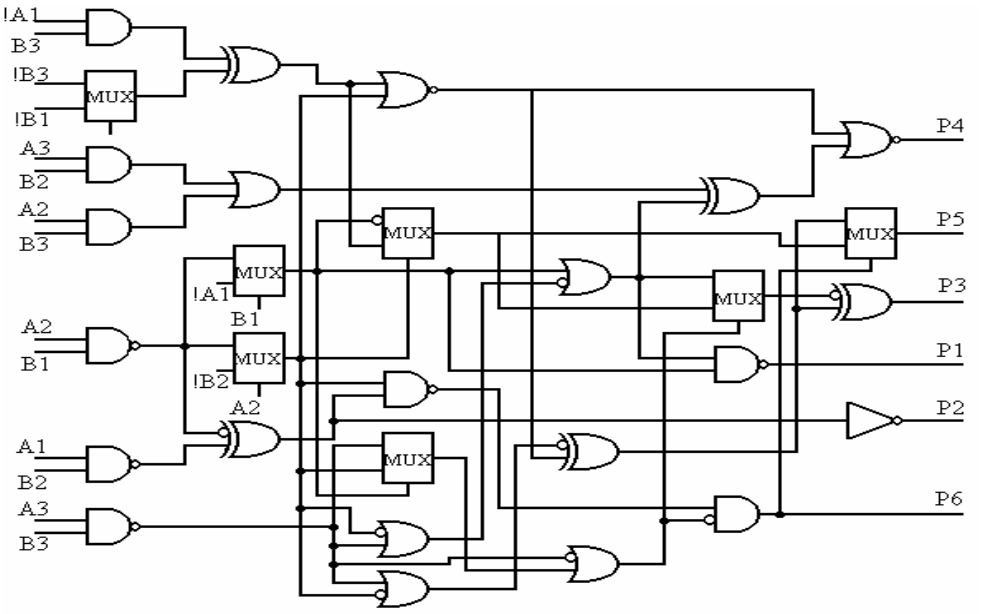

Figure 6.16 The evolved three-bit multiplier.

Surprisingly it doesn't prove difficult to obtain solutions which are 95% correct and on a number of occasions 99.22% was achieved (this means the solution had only three bits in 384 incorrect). The population size on this occasion was 100. The GA is being very successful at getting up to percentages of correct bits in the high nineties but is then failing to move from these points to 100%. A solution with

a fitness of 99.22% was examined and it was not possible to see how it could be fixed so that it could become 100%. This reveals the difficulty. The problem appears to be associated with a very deceptive fitness landscape. The reason why evolution should find the three-bit multiplier so exceptionally more difficult than the two-bit multiplier is still unclear. It may be that the geometries being chosen are inadvisable or, simply that the three-bit multiplier has a truth table which has six times the number of bits in the output field as the two-bit multiplier. So it could be merely that there is a large intrinsic growth in the difficulty. However, recall that the four-bit adder has been evolved a number of times - this has 2,560 bits in the output field! Clearly using the number of bits in the output field is not that good measure of difficulty. Of course one would expect that there would be many more ways of building an adder than a multiplier so adders should be naturally easier to evolve. Clearly a greater understanding of the mechanism by which the GA is able to evolve 100% solutions is required. The next section attempts to gain insight into precisely this matter.

6.4.5 Aspects of the genetic history of a design for a two-bit adder leading to a 100% functional solution.

It might be thought that the GA which has been described in this article would inevitably be carrying out some kind of local search, given that it has been the authors' finding that the use of a low population size and very high number of generations gives better results. To explore this contention, consider the effect on the functionality of a chromosome when a single gene in a netlist chromosome is altered at random. The consequences in loss of functionality may be catastrophic as for instance, a primary output might, as a result, be connected to the wrong cell output. This observation implies that at the very least the local fitness landscape of a particular chromosome is very noisy. Another important factor here is the fact that the number of possible neighbours to a chromosome (defining a neighbour to be a chromosome which differs from the original in a single gene) is very large indeed. These two factors make it all the more surprising that low population sizes and high numbers of generations are more effective. One's sense of mystification is heightened when one recalls that elitism also improved the results. In an attempt to gain some insight into what is going on it is interesting to examine the history of gene changes of the best in the population as it develops from initially containing a large number of random elements to the final highly evolved form associated with possessing 100% functionality. Table 6.5 below shows the evolution of the best chromosome of the population. Each new improvement in fitness and the generation at which it occurred are also shown. In addition the fitness changes are shown and the number of different genes between chromosomes, where only genes which contributed to the functionality of the resulting circuit were counted (see also Figure 6.17). The chromosomes relate to a 3 x 4 geometry of cells. The target function is the two-bit adder with carry. The GA parameters are as follows:

population size = 60, breeding rate = 100.00, mutation rate = 5.00%, number of generations = 40,000, tournament discriminator = 0.67, elitism is used, levels back = 2.

Clearly, as expected, genetic differences are greatest for relatively small numbers of generations (up to generation 48). After that there is surprisingly no obvious trend towards low genetic differences, although occasionally genetic differences are very small (generations 1423 and 1351 or 10544 and 10205). The figures relating to fitness change are interesting and quite unexpected as they are extraordinarily constant until generation 919 and then the change halves to 1.04 before shortly after doubling to 4.17. The last four circuits leading up to and including the 100% circuit are examined in Figure 6.17.

Table 6.5 The genetic history of the best chromosome up to 100% functionality for the two-bit adder

Genes 0-17	Generation	Fitness	Fitness Change	Genetic differences in circuit
9 8 10 3 11 -5 9 8 -1 7 14 12 6 5 -7 7 10 -2	1	58.33	0	0
10 6 5 1 11 -5 9 6 -11 9 11 -4 7 13 -7 2 14 -4	4	60.42	2.09	27
4 9 -2 4 6 -11 4 8 -8 7 14 -7 2 14 9 5 7 -12	8	62.50	2.08	35
9 3 -2 2 4 7 1 4 -2 2 14 -5 6 2 11 4 10 11	15	64.58	2.08	31
9 3 -2 7 10 -5 5 9 4 0 14 7 6 3 -7 4 12 4	48	66.67	2.09	23
9 11 -2 7 9 -5 8 8 8 8 10 -8 6 3 -7 4 7 4	124	68.75	2.08	17
2 11 -7 7 9 -5 8 8 -2 0 10 -8 6 3 -7 14 12 4	161	70.83	2.08	10
2 11 -7 7 9 -5 4 8 8 0 10 -8 6 3 -7 14 7 4	192	72.92	2.09	12
2 11 -7 7 9 -5 8 8 8 0 10 -8 6 3 -7 6 12 4	193	75.00	2.08	8
2 11 0 7 9 -5 4 5 8 0 10 -8 6 3 -7 2 1 4	316	77.08	2.08	5
2 11 0 7 2 -5 8 5 -10 0 10 -8 6 3 -7 12 7 4	358	79.17	2.09	11
3 9 6 4 2 -5 7 1 5 0 10 -7 6 3 -7 12 7 4	693	81.25	2.08	14
3 9 6 4 2 -5 7 1 5 0 10 -7 6 3 -7 12 7 4	775	83.33	2.08	10
3 1 6 4 2 -5 7 1 5 0 10 -9 6 3 -7 12 7 4	919	85.42	2.09	5
3 5 6 4 2 -5 7 5 -7 0 10 -9 6 3 -7 12 7 4	1351	86.46	1.04	13
2 5 6 4 2 -5 7 5 -7 0 10 -9 6 3 -7 12 7 4	1423	87.50	1.04	3
4 2 -5 0 0 -4 3 6 6 0 10 -9 6 3 -7 12 7 4	5417	89.58	2.08	16
2 2 2 10 4 -6 3 6 6 7 10 -8 6 3 -7 2 7 4	10205	93.75	4.17	12
4 2 1 11 8 -6 3 6 6 9 10 -8 6 3 -7 2 7 4	10544	97.92	4.17	2
4 9 -9 11 8 -6 3 1 6 2 10 -8 6 3 -7 2 7 4	14980	100.00	2.08	8

Genes 18 - 38	Output genes
13 14 -12 1 0 17 17 13 -6 0 20 -11 1 0 16	18 17 15 23 22 20
17 17 12 17 16 14 15 12 15 19 20 -2 1 18 17	16 17 -7 23 18 20
12 15 16 12 16 13 17 16 12 0 20 16 19 0 20	19 19 15 21 22 20
13 15 16 14 13 1 0 16 13 18 20 19 20 0 19	15 19 -2 20 18 21
13 14 16 17 13 -11 13 17 -11 18 16 20 1 0 18	19 17 19 20 18 21
13 15 16 0 14 14 13 15 -11 18 18 15 19 16 18	20 17 -10 20 23 21
13 15 16 14 14 14 13 15 -11 18 18 20 19 15 -1	20 17 -12 20 23 21
13 15 16 14 1 14 13 15 -11 18 16 20 19 15 -1	20 17 -12 20 23 21
13 15 16 14 1 14 13 15 -11 18 16 20 19 0 -1	20 17 -12 20 23 21
13 15 16 14 17 14 13 15 -11 18 16 20 20 16 -7	20 17 -4 20 23 21
13 15 16 0 17 -5 13 15 -11 18 16 20 19 18 -7	20 17 -4 20 23 21
13 15 16 0 12 -4 13 15 -1 18 16 20 16 16 -7	20 17 -4 20 23 21
13 15 16 1 13 -5 13 15 -1 18 16 20 15 16 -7	20 17 -4 20 23 22
13 17 16 0 13 12 13 15 -1 16 0 20 15 16 -7	20 17 -4 19 23 22
17 0 15 14 13 15 13 15 -1 19 16 18 15 16 -7	20 17 -4 21 23 22
17 0 15 14 13 15 13 15 -1 19 18 -2 15 16 -7	20 17 -4 21 23 22
17 13 15 12 14 17 17 0 -8 19 18 -2 15 16 -7	15 17 -4 21 23 22
17 0 15 12 14 17 13 14 15 19 18 -2 15 16 -7	20 17 -4 21 23 22
17 0 15 12 14 17 13 14 15 19 18 -2 15 16 -7	20 17 -4 21 23 22
17 0 16 12 14 17 13 14 15 19 15 18 15 16 -7	20 17 -4 21 23 22

On this occasion five runs were carried out with results shown in Table 6.6. It is clear that allowing 40,000 generations on this occasion was unnecessary. Note the very large variation in the generation for which the GA found the best chromosome. Also note that the last two runs show typical behaviour in that 100% functionality usually occurred late-on (the algorithm immediately terminating on achieving 100% percent functionality). The chromosome evolution shown in Table 6.5 corresponds to the fifth run in Table 6.6.

Table 6.6 Final results for five runs of the GA on the two-bit adder with carry problem.

Best chromosome found at generation	Percentage output bits correct (fitness)
515	87.50
8,839	95.83
6,902	87.50
21,114	100.00
14,980	100.00

As has been mentioned previously an analysis program has been written which can take a chromosome and produce a diagrammatic array of cells. This program strips out all cells which are not involved in producing a primary output. This was run on the last four chromosomes shown in Table 6.5 and the results are shown in the next figure.

```
4 12    X       17 18    19 21
2 -5    X        0  5    18 -2

 X    6 16    12 19     5 22
 X    3 -7    14 17    16 -7

 3 14    12 17    X       5 23
 6  6     7  4    X      17 -4

outputs 21  23  22,   fitness  89.58
```

```
 X       X       17 18    19 21
 X       X        0  5    18 -2

11 13    6 16     2 19     5 22
 8 -6    3 -7    14 17    16 -7

 3 14    2 17    13 20    20 23
 6  6    7  4    14  5    17 -4

outputs  21  23  22,  fitness  97.92
```

```
 X       X       17 18    19 21
 X       X        0  5    18 -2

10 13    6 16     2 19     5 22
 4 -6    3 -7    14 17    16 -7

 3 14    2 17    13 20    20 23
 6  6    7  4    14  5    17 -4

outputs 21  23  22,   fitness  93.75
```

```
4 12    X       17 18    19 21
9 -9    X        0 16     5 18

11 13    6 16    12 19     5 22
 8 -6    3 -7    14 17    16 -7

 3 14    2 17    13 20    20 23
 1  6    7  4    14  5    17 -4

outputs  21  23  22,  fitness  100.00
```

Figure 6.17 Cellular representation of last four chromosomes of Table 6.5

In Figure 6.17 X represents a cell which is redundant. The emboldened figures represent differences from the previous diagram, i.e. in the circuit at the bottom left corner only two figures are in bold, 11 and 8, which are the two inputs to the first cell in the second row - these are the only differences from the circuit in the top right position. It is clear that the genetic diversity in the whole population must be remaining at a reasonable level since 8 gene changes were required to transform the second best solution into the 100% functional solution. The implication being that the GA processes were able to preserve this genetic material in the population and eventually create the final solution with the aid of crossover. It is very unlikely that these 8 changes could have occurred by 8 genes changing in the just the right way, by mutation alone, as with the parameters given only 1.95 genes per chromosome would be mutated on average.

6.5 DIRECT EVOLUTION ON THE XILINX 6216 PART

The next stage of the work is to attempt to replicate the results achieved in the off-chip simulations described previously on the target device itself. The chosen target device, in this case, being the Xilinx 6216, fine-grained FPGA.

The reason that this is important is due to the fact that in attempting to design any particular combinational logic circuit using evolution, one also needs to demonstrate that the resultant design will actually fit on to the implementation platform. In the past, designers - including our own group [MT96] - have attempted to use evolutionary methods to synthesise logic solutions using rule-based techniques. In other words, the desired logic functionality is simplified using a combination of a Genetic Algorithm (usually to select input variable ordering) and established Boolean or AND-EXOR algebraic rules. On completion of this process one has a simplified or minimised representation of the required function, but little or no information about how this will fit on to a chosen implementation platform. For example, one may wish to use the Xilinx 6216 part to implement the design once simplified, but may find that it is impossible to *place and route* within the constraints imposed by the device's internal structure.

This is where many design or synthesis techniques fail. They do not take into account the nature of the device where the final design solution is to be physically implemented. This means - particularly for devices such as FPGAs, where resources are limited - that many of these designs cannot be implemented without further analysis and possible reconfiguration.

If the design is evolved directly on the device itself, within a given set of resources (i.e. it is constrained, for example, to a given number of cells), then this place and route aspect of the implementation process ceases to be a problem because one knows from the success or failure of the function to evolve, that the design either does or does not fit on to the part.

Of course, the method adopted for actually making the design evolve directly on the part depends very much upon the internal structure of the device under consideration. As mentioned previously, the Xilinx 6000 parts do have some extremely useful features when it comes to considering how the evolutionary process should be used to reconfigure internal structure. For example, the ability to reconfigure an individual cell without affecting those around it is probably the single most useful aspect of the chip's design when it comes to considering evolution.

In the 6000 family a cell is made up of two basic parts - a functional block which dictates cell functionality e.g. AND, OR, NOT, EXOR and so on, and a series of switches which dictate routing to that cell from either its neighbouring cells, or cells located elsewhere on the chip. Figure 6.18 shows the basic cell layout.

The functional block contains a single *D-type flip-flop*, and switching capable of implementing all two-input logic functions plus all possible two-input multiplexer functions. Figure 6.19 shows the internal switching arrangement for the functional sub-block.

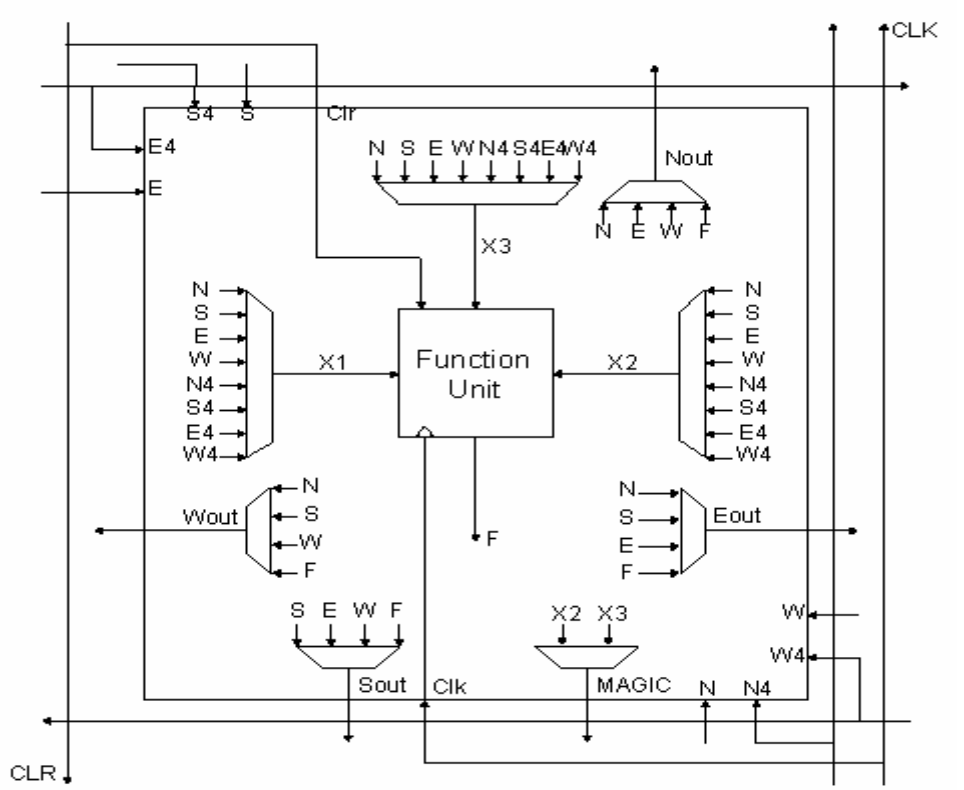

Figure 6.18 Xilinx 6216 part, internal cell layout

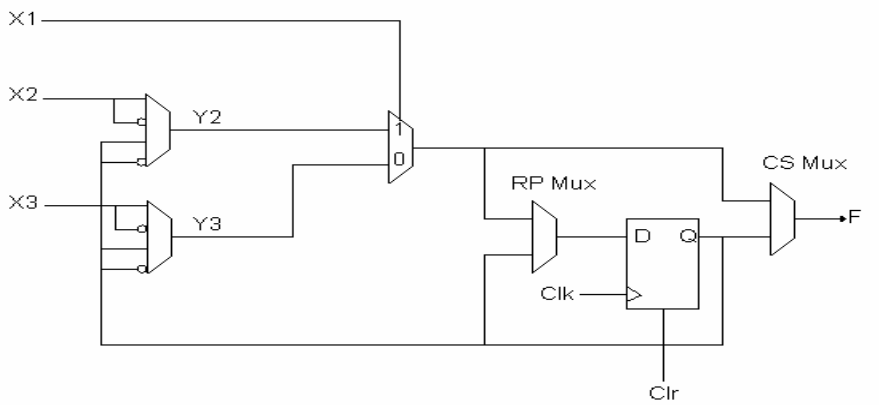

Figure 6.19 The functional sub-block of the 6216 logic cell

The user may configure each cell to be any two-input logic function desired. This is achieved by setting the switches that control the Y2 and Y3 outputs, and also by matching the inputs X1, X2 and X3 to the corresponding values. However, this is actually achieved by writing to a *random-access memory* (RAM) on the chip whose data controls the value of these signals. Since each cell RAM location is individually addressable, then the user may change the functionality of a single cell by updating the correct memory location directly. This means that only that particular cell will be affected. All other cells will remain as they were until their RAM address is written to. This is the way in which individual cells - or groups of cells - may be altered without affecting the other chip functions. Additionally, the routing control may also be altered in this manner, and so entire circuit functionality can be altered. This, of course, is a process which is necessary in evolution.

The routing control switches operate in a manner such that any cell may derive its X1, X2 and X3 inputs from any of its neighbours designated to lie to the North, South, East or West. It is not possible to route directly from a diagonal neighbour, except if one of the chip's main routing lines were to be used. The cells on the device - of which there are 64 by 64 in total laid out in a grid - are organised into sub-groupings of 4 by 4. Each of these sub-groupings has access to paths which can carry signals to any other point on the chip. It would be possible, therefore, to use one such path to route diagonal neighbours, but this would probably not be done. Instead, an orthogonal neighbour of the two cells would provide a straight-through connection.

Currently the primary concern has been to prove the *concept* of evolution on the chip, and so the work so far has involved merely working with a designated region as a "sea of gates". As a consequence potential routings across different regions of the device using the main routing paths have been ignored, and work has concentrated on connecting neighbours to form the desired functionality for particular applications.

Great care has to be taken when routing signal paths in to a particular cell. This is because of the way the cells are named, and the perspective from which the routing path is viewed. Cells are organised in a grid, and designated a co-ordinate according to the normal Cartesian system. Therefore, the cell in the bottom, left-hand corner of the device is designated as cell (0,0), while the cell in the top, right-hand corner is denoted (63,63). Now consider cell (1,1). If one wished to receive a signal into this cell from it neighbour located at (0,1), one would imagine that this would mean a signal coming into the cell from its neighbour located to the West. However, the routing is not viewed from the perspective of the destination of the signal path, but rather from its source. Therefore, the correct designation for this signal path is not West, but rather East, because (1,1) is to the East of (0,1). This means that from the viewpoint of the cell at (1,1), the desired input comes in on the East switch, and so this has to be configured to drive one of the required X inputs.

In order to assist in this conversion, the authors have written a rudimentary compiler that accepts configuration commands in the following form:

<x co-ordinate> <y co-ordinate> <gate function> <X1> <X2> <X3>

The software then converts this statement to the correct address and data to be stored in the RAM which will configure the cell. Therefore, if cell (1,1) was to become an AND gate which had inputs fed from its neighbours (0,1) and (1,0), then this would be written as follows:

1 1 AND E N S

The software would then derive the correct address and data to configure the (1,1) cell.

The next intention is to use this type of information to form the chromosomes which will be evolved to provide the on-chip evolution. The chromosomes will be written to a designated region of the chip, and then circuitry which is permanently in place - consisting of a counter and comparator - will test the functionality and automatically assess the fitness of the solution by comparison with the desired circuit specification. Xilinx have created a 6216 plug-and-play board, complete with an accessible interface, which is installed in a PC and may be written to and read from directly by the host computer. The experiments for conducting this on-chip evolution of combinational circuits are still in progress, and results will be reported in the very near future.

6.6 CONCLUSIONS

In this case study on arithmetic circuits it has been shown that by evolving a linear chromosome of cell functionalities and connectivities based on a rectangular array of logic cells it is possible to evolve both traditional and novel designs for arithmetic circuits. Indeed designs have emerged, in particular, for the two-bit multiplier which are not only interesting in their own right but may lead to generalisation, so that new designs for higher order multipliers may be produced. Clearly it is apposite in this regard to attempt to allow similar evolutionary processes to produce an optimum design for the three-bit adder as it is still to difficult to see how to generalise the two-bit cases. The three-bit multiplier has proved to be extraordinarily more difficult to evolve than the two-bit case. At present the reason for this is unclear and especially so when one bears in mind that several designs have been produced for a four-bit adder.

There are still many avenues for further work. Firstly, other ways of representing rectangular arrays of logic cells may be devised and also, the relationship between cell connectivity and the evolvability of designs has still to be explored, this would involve examining a suitable concept of cell-neighbourhood and exploring the impact of choosing different values for the ‘levels -back’ parameter. Generally it has been the case that choosing this to be two gives better results, but there is still more careful work to be done here. It is still surprising at just how effective is the genetic algorithm described in this chapter, however a great deal more of analysis needs to be done to discover firstly just why is it effective, and secondly to improve its effectiveness.

There are many wider issues to be considered also which relate to the problem of evolving much larger circuits. It is a feature of the current technique that one has to specify the functionality of the target circuit using a complete truth table, however this is impractical for circuits with large numbers of inputs. In general circuits are specified in the form of truth tables in which many input combinations are not explicitly specified (a ‘-’ is used in the input field to indicate that either a ‘0’ or a ‘1’ can be used), this of course has a consequence that functions with hundreds of input variables can then be specified without too many input conditions. Also often functions are specified directly only it they give a ‘1’ output for a given input combination, certainly it is with this form that logic optimisation algorithms almost exclusively work. However in the technique described here it is essential to consider state of the outputs for the entire input combinations.

It should be borne in mind that although evolving designs ‘in-circuit’ on an FPGA chip, such as Xilinx’s 6216, is interesting and worthwhile in its own right, and indeed may lead to faster evolution it doesn’t solve the problems of scale described in the previous paragraph. Neither is the ‘scalability problem’ alleviated by trying to evolve in a hierarchical way, as the amount of information in the specification does not change.

Finally, there is reason to be excited about future developments in this new field, but of course, many difficult problems will still have to be overcome.

REFERENCES

[Alm94] Almaini A. E. A., (1994) "Electronic Logic Systems", 3rd Edition, Prentice Hall, London.

[BM96] Bratt A., and Macbeth I., (February 1996) Design and implementation of a field programmable analogue array, in *Proceedings of FPGA'96*, Monterey, California.

[Gre97] Greene J., (July 1997) Simulated Evolution and Adaptive Search in Engineering Design - Experiences at the University of Cape Town, in *2nd Online Workshop on Soft Computing*.

[GSMSST96] Goeke M., Sipper M., Mange D., Stauffer S., Sanchez E., and Tomassini M., (September 1996) Online autonomous evolware, in *Proceedings of The First International Conference on Evolvable Systems: From Biology to Hardware (ICES96),* now published in *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, 1997.

[IIH96] Iba H., Iwata M., and Higuchi T., (September 1996) Machine Learning Approach to Gate-Level Evolvable Hardware, in *Proceedings of The First International Conference on Evolvable Systems: From Biology to Hardware (ICES96),* now published in *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, 1997.

[KK96] Kutuk K., and Kang S., (May 1996) A field-programmable analog array (FPAA) using switched-capacitor techniques, in *Proceedings of ISCAS'96*, Vol. IV, Atlanta.

[KABK97] Koza J. R., Andre D., Bennett III F. H., and Keane M. A., (April 1997) Design of a High-Gain Operational Amplifier and Other circuits by Means of Genetic Programming, in *Proceedings of the 6th International Conference on Evolutionary Programming (EP97),* now published in *Lecture Notes in Computer Science*, Vol. 1213, pages 125-135, 1997.

[Koz92] Koza J. R., (1992) *Genetic Programming*, The MIT Press, Cambridge, Massachusetts.

[MT96] Miller J. F., and Thomson P., (April 1996) Restricted Evaluation Genetic Algorithms with Tabu Search for Optimising Boolean Functions as Multi-Level AND-EXOR Networks in *Proceedings of AISB Workshop 1996*, now published in *Lecture Notes in Computer Science*, Vol. 1143, pages 85-101, 1996.

[PH96] Papathanasiou K., and Hamilton A., (November 1996) Novel PALMO Analogue Signal Processing IC Design Techniques, in IEE Colloqium on Analog Signal Processing, pages 5/1-5/6.

[SSMTP-US97] Sipper M., Sanchez E., Mange D., Tomassini M., Pérez-Uribe A., and Stauffer A., (April 1997) A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Harware Systems, in *IEEE Transactions on Evolutionary Computation*, Vol. 1, No 1., pages 83-97.

[Tho96] Thompson A., (September 1996) An evolved circuit, intrinsic in silicon, entwined with physics, in *Proceedings of The First International Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, now published in *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, 1997