An Extrinsic Function-Level Evolvable Hardware Approach

Tatiana Kalganova

School of Computing, Napier University
219 Colinton Road, Edinburgh, EH14 1DJ, UK
`t.kalganova@dcs.napier.ac.uk`,
WWW home page: `http://www.dcs.napier.ac.uk/~tatiana/`

Abstract. [1] The function level evolvable hardware approach to synthesize the combinational multiple-valued and binary logic functions is proposed in first time. The new representation of logic gate in extrinsic EHW allows us to describe behaviour of any multi-input multi-output logic function. The circuit is represented in the form of connections and functionalities of a rectangular array of building blocks. Each building block can implement primitive logic function or any multi-input multi-output logic function defined in advance. The method has been tested on evolving logic circuits using half adder, full adder and multiplier. The effectiveness of this approach is investigated for multiple-valued and binary arithmetical functions. For these functions either method appears to be much more efficient than similar approach with two-input one-output cell representation.

1 Introduction

Evolvable Hardware (EHW) is technique to synthesize electronic circuits using genetic algorithms. The search for an electronic circuits realization of a desired transfer characteristic can be made in software as in *extrinsic* evolution, or in hardware as in *intrinsic* evolution. In extrinsic evolution the entire evolution process is implemented in the software simulator based on a model of the implementation technology. In intrinsic evolution the hardware actively participate in evolution process [1].

In the context of electronic synthesis the configuration of evolved circuit as well as connecting elements inside circuit is represented by chromosome. Any genetic operators are applied to the chromosome and obtained new circuits are compared with target logic function. The process in usually ended after a given number of generation or when the closeness to the target response has been reached. If connecting element is represented by primitive logic function, a *gate-level EHW approach* is applied. In the *function-level EHW approach*, high level hardware functions such as adders, multipliers, etc. rather than simple logic functions are used as primitive functions in evolution [2], [3]. Therefore the building

[1] To appear in R. Poli, W. Banzhaf, editors, *Proc. of the 3rd European Conference on Genetic Programming, EuroGP2000*, Edinburgh, UK, Springer-Verlag, Spring 2000.

block implements the multi-input one-output or multi-input multi-output logic function.

A variety of extrinsic EHW methods have been used to synthesise digital circuits (Table 1). In most cases primitive functions or multiplexers have been considered as connecting elements. In our work we use multi-input multi-output logic functions to define the behaviour of connecting elements.

Table 1. Summary of extrinsic EHW approaches for digital circuit design

Author	**Building Block**	**Fitness function**	**Evolutionary approach**	**Application**
Kitano H., 1996 [4]	$f(2,1,2)$	-	Evolutionary programming	Digital design.
Zebulum R.S. et al, 1996 [5]	$f(2,1,2)$	F1	GA	Binary logic design.
Murakawa M. et al, 1996 [6]	$f(n,1,2)$	F1	VGA	Adaptive equalization.
Miller J. et al, 1997 [7]	$f(2,1,2)$ $f(n,m,2)$	F1	Cartesian GP	Arithmetic binary circuit design.
Kalganova T. et al, 1998 [8]	$f(2,1,r)$ $f_m(r+1,1,r)$	F1	Cartesian GP	Multi-valued circuit design.
Damiani E. et al, 1999 [9]	$f(4,1,2)$	F1	Conventional GA	Hash function design.
Aguirre A.H. et al, 1999 [10]	$f_m(3,1,2)$	F1, F2	GP	Binary circuit design.
Kalganova T. et al, 1999 [11]	$f(2,1,2)$ $f_m(3,1,2)$	F1, F2	Cartesian GP	Binary circuit design.
Masher J. et al, 1999 [12]	$f(2,1,2)$	F1, F3	Conventional GA	Sorting network design.
Miller J., 1999 [13]	$f(2,1,2)$ $f(3,1,2)$	F4	Cartesian GP	Low pass filter design.
Proposed method	$f(2,1,r)$ $f(n,m,r)$	F1, F2	Cartesian GP	Multi-valued logic design.

VGA - an variable-length chromosome GA; GP - Genetic Programming $f(n,m,r)$ is an n-input m-output r-valued logic function; $f_m(3,1,2)$ is an logic function described the behaviour of multiplexer; F1 defines the correctness of outputs of logic circuit evolved; F2 is the minimal number of logic cells used; F3 is the correctness of input combinations; F4 is an error based fitness.

The proposed method is an extension of EHW approach applied for binary circuit design [7] and multi-valued logic (MVL) functions [8]. Some aspects of this approach have been investigated in the past. Thus, it has been found that functional set of logic gates [14] as well as circuit layout and connectivity restrictions [8] influence on the GA performance. Some attempts to evolve circuit layout together with circuit functionality have been reported in [11]. A dynamic fitness function has been proposed in [11], that allow us to evolve functional complete circuit with minimal number of logic gates employed.

In this paper we proposed to use multi-input multi-output logic functions as logic cell (so called building block) in an extrinsic evolvable hardware approach. We limit our focus to binary and multi-valued combinational logic circuit design problems. We introduce the new chromosome representation, that allows us to evolve logic circuits using high-level logic functions. A number of binary and multi-valued circuit structures evolved are discussed. The experimental results show us that the proposed method performs better than one earlier reported in case if the suitable functional set of logic gates is chosen.

2 A Problem Statement

Any design problem can be represented as follows. Let $\mathbb{C}_1$ and $\mathbb{C}_2$ be the sets of values and let $X = \{x_0, x_1, \cdots, x_{n-1}\}$ be an input vector of n variables, where x_k takes on values from $\mathbb{C}_1$. Let $Y = \{y_0, y_1, \cdots, y_{m-1}\}$ be an output vector of m variables, where $y_k \in \mathbb{C}_2$. Let $\mathbb{G}$ be the set of primitive operations over values from sets $\mathbb{C}_1$ and $\mathbb{C}_2$. The cardinality of X defines the number of members of X and denoted as $|X|$. Then, $|X| = n$. Let f be the function such that $\mathcal{F} : \mathbb{C}_1^n \longrightarrow \mathbb{C}_2^m$. Let function $\mathcal{F}$ be represented as a matrix mapping denoted as $X \to Y$, where X is a $(k \times n)$ matrix of all the given inputs and k is the number of input combinations, and Y is a $(k \times m)$ matrix of the corresponding m outputs. Then the synthesis of function can be stated as follows. Design a sequence of operations that accomplishes the mapping $X \to Y$. This mapping is achieved by applying a sequence of primitive or complex operations. This statement of design problem can be applied to any type of functions. We will consider the logic design of binary and multi-valued logic functions.

Binary circuit design: Let $\mathbb{B} = \{0, 1\}$ be the set of binary logic values. Then the design of binary circuits can be represented by the equations given above, if $\mathbb{C}_1 = \mathbb{B}$, $\mathbb{C}_2 = \mathbb{B}$.

Multi-valued circuit design: Let $\mathbb{R} = \{0, 1, \cdots, r-1\}$ be the set of r-valued logic values. Then the multi-valued circuit design can be described by the equations given above, if $\mathbb{C}_1 = \mathbb{R}$ and $\mathbb{C}_2 = \mathbb{R}$. The primitive two-input multi-valued logic operators are defined as follows:

$$\begin{array}{rcl}
NOT & : & !x_1 = \overline{x_1} = (r-1) - x_1 \\
SUCCESSOR & : & ?x_2 = x_2 + 1 \pmod{r} \\
MIN & : & x_1 \cdot x_2 = min(x_1, x_2) \\
MAX & : & x_1 \vee x_2 = max(x_1, x_2) \\
MODSUM & : & x_1 \oplus x_2 = x_1 + x_2 \pmod{r} \\
MODPRODUCT & : & x_1 \otimes x_2 = x_1 * x_2 \pmod{r} \\
TSUM & : & x_1 \diamond x_2 = min(x_1 + x_2, r-1) \\
TPRODUCT & : & x_1 \star x_2 = min(x_1 + x_2 - (r-1), 0).
\end{array}$$

where "+" is an ordinary addition. The TPRODUCT operator has been first introduced in [15], [16]. The symbolic representation of MVL operators mentioned above are shown in Fig. 1. More information about multiple-valued logic can be found in [17].

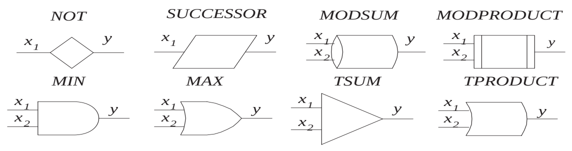

Fig. 1. Symbols and analytic representation of two-input r-valued logic gates

3 Function-Level Extrinsic Evolvable Hardware Approach

There are several issues of interest concerning the use of cartesian generic programming [18] in this domain, such as the encoding of building blocks implemented multi-input multi-output logic functions, mutation operator, repair procedures. The representation chosen for our work allows us to synthesise the logic circuits using multi-input multi-output logic sub-functions defined in advance.

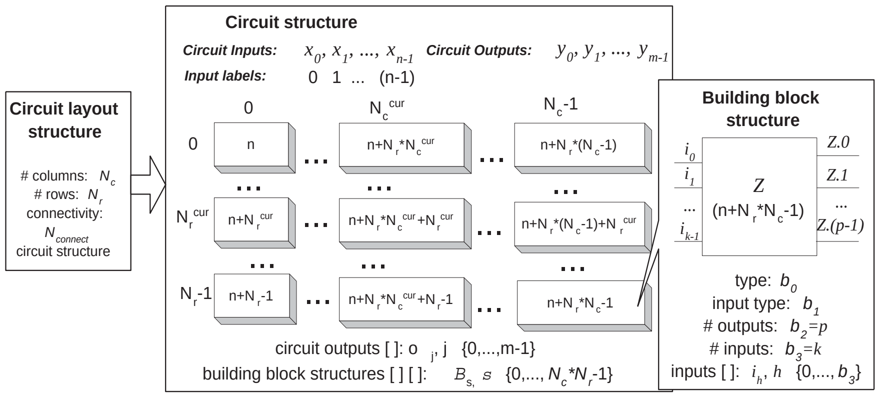
Circuit layout structure
# columns: N_c
# rows: N_r
connectivity: N_connect
circuit structure
Circuit structure
Circuit Inputs: x_0, x_1, ..., x_{n-1}
Circuit Outputs: y_0, y_1, ..., y_{m-1}
Input labels: 0 1 ... (n-1)
circuit outputs [ ]: o_j, j {0,...,m-1}
building block structures [ ] [ ]: B_s, s {0,..., N_c*N_r-1}
Building block structure
Z (n+N_r*N_c-1)
type: b_0
input type: b_1
# outputs: b_2=p
# inputs: b_3=k
inputs [ ]: i_h, h {0,..., b_3}

Fig. 2. Schematic of chromosome structure: N_c, N_r are the number of columns and rows respectively; N_c^{cur}, N_r^{cur} are current column and row respectively.

3.1 Encoding

A combinational logic circuit is represented as a rectangular array of building blocks (Fig. 2). Each building block is uncommitted and can be removed from the actual circuit design if they prove to be redundant. The building block can implement any primary logic operation or multi-input multi-output logic function defined in advance. The inputs to any building block in the combinational network may be the primary inputs or any outputs of building blocks which are in columns to the left of the building block in question. The circuit inputs $x_0, x_1, \cdots, x_{n-1}$ are numbered $0, 1, \cdots, (n-1)$ respectively. The building blocks which from the array are numbered column-wise from n to $(n+N_c*N_r-1)$, where N_c and N_r are the number of columns and rows in rectangular array. The outputs of building block $\mathcal{B}(z)$ are labeled from $(z+0/D_{order})$ to $(z+(p-1)/D_{order})$, where z is the building block in question, p is the number of outputs in building block $\mathcal{B}_z$ and D_{order} defines the decimal order of the maximum number of outputs allowed to be used in building block. For example, if the maximum number of output in building block can not exceed 99, then $D_{order} = 100$. If 9 outputs are allowed to be used in building block, then $D_{order} = 10$. Thus, each output of building block is defined by real number. In the work reported we consider evolving binary and MVL functions. Table 2 shows the set of both binary and MVL functions employed in evolution.

Table 2. Gate functionality according to the $b_0(z)$ gene in chromosome; "-" denotes that the logic function is not assigned for encoding value.

Gene functionality, $b_0(z)$	Gate logic function	
	binary	multi-valued
0	Logic constant	Logic constant
1	-	$SUCCESSOR : ?x_0$
2	$NOT : \overline{x_0}$	$NOT : !x_0$
3	-	Literal: $L(x_0, c)$
4	-	Clockwise Operator: $c \leftarrow x_0$
5	-	Counter Clockwise Operator: $c \rightarrow x_0$
6	Wire: x_0	Wire: x_0
7	$AND : x_0 \cdot x_1$	$MIN : x_0 \cdot x_1$
8	$OR : x_0 \vee x_1$	$MAX : x_0 \vee x_1$
9	$EXOR : x_0 \oplus x_1$	$MODSUM : x_0 \oplus x_1$
10	-	$MODPRODUCT : x_0 \otimes x_1$
11	-	$TSUM : x_0 \diamond x_1$
12	-	$TPRODUCT : x_0 * x_1$
13	-	$x_0 + x_1$ (over $GF(r)$
14	-	$x_0 * x_1$ (over $GF(r)$)
15	Multiplexer	T-gate
16	-	1-digit multiplier
17	1-digit full adder	1-digit full adder
18	2-digit multiplier	2-digit multiplier
19	2-digit full adder	2-digit full adder
20	3-digit multiplier	3-digit multiplier
21	3-digit full adder	3-digit full adder
22	Half adder	Half adder

The chromosome is represented by a 3-level structure: 1) Circuit layout structure; 2) Circuit structure; 3) Building block structure.

Circuit layout structure: At the first level the global characteristics of the circuit are defined. These are connectivity parameter $N_{connect}$, the number of rows N_r and columns N_c. Note that these parameters are allowed to be variable, but in this paper we consider that the circuit layout is defined in advance and it is not allowed to be changed during evolution process.

Circuit structure: At the second level the array of building blocks $\mathcal{B}_i$ is created and the circuit outputs $\mathcal{O} = \{o_0, o_1, \cdots, o_m\}$ are determined.

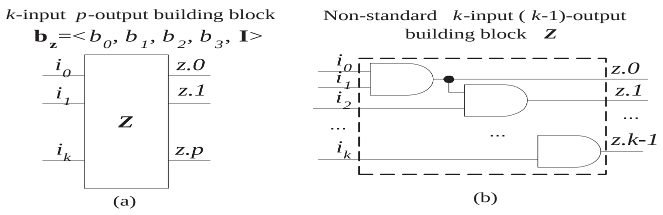
k-input p-output building block
b_z=<b_0, b_1, b_2, b_3, I>
i_0
i_1
i_k
Z
z.0
z.1
z.p
(a)
Non-standard k-input (k-1)-output building block Z
i_0
i_1
i_2
...
i_k
...
z.0
z.1
...
z.k-1
(b)

Fig. 3. Schematic of building block structure.

Building block structure: Finally, the third level represents the structure of each building block in the network $\mathcal{N}$. This data consists of the functional gene b_0, the type of inputs b_1, the number of outputs b_2 and inputs b_3 and the input connections i_h, (Fig. 2, Fig. 3(a)) . In this work the gene b_1 defining the type of inputs in building block has not been taken into account. The number of inputs and outputs in building block depend on the functional type of building block b_0 and are defined if the value of functional gene is known. Thus, if functional gene defines the primitive logic function, then variable number of inputs in building block can be used. In this case the primitive logic gates are connected as shown in Fig. 3(b). For example, let " $\vee$ " defines the 2-input 1-output logic primitive function and $b_3 = 4$. The number of outputs in building block is 3. These outputs can be analytically represented as follows:

$$o_0(z) = i_0 \vee i_1;$$
$$o_1(z) = (i_0 \vee i_1) \vee i_2;$$
$$o_2(z) = ((i_0 \vee i_1) \vee i_2) \vee i_3.$$

If multi-input multi-output logic functions defined in advance is employed, then the number of inputs and outputs in building block are fixed and can not be changed. For example, the binary two-bit full adder has 5 inputs and 3 outputs. So, if the functional gene of building block is 19 ($b_0 = 19$), then $b_2 = 3$ and $b_3 = 5$.

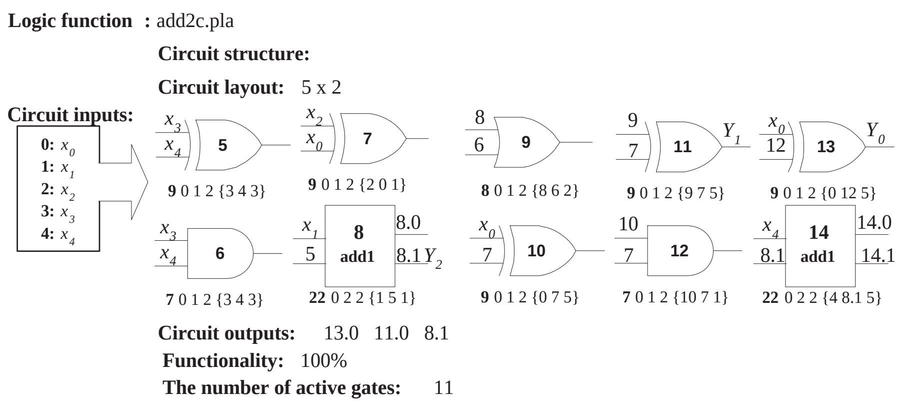

Fig. 4. An example of the phenotype and corresponding genotype of a chromosome with 5x2 circuit layout

3.2 An example

An example of chromosome representation with the actual circuit structure is given in Fig. 4. This circuit represents the full 2-bit adder evolved using AND, OR, EXOR and half adder binary logic functions. This function has 5 inputs and 3 outputs and is implemented on a combinational network with 5x2 circuit layout ($N_c \times N_r$). The labels of circuit inputs 0, 1, 2, 3, 4 correspond to the input variables x_0, x_1, x_2, x_3 and x_4 respectively. The type of building block is defined by functional gene shown in bold. The encoding table of functional gene is given in Table 2.

Each cell is assigned an individual address. Thus the building block located in 0th column and 0th row is labeled as 5. The building block located in 4th column 1st row is labeled as 14. Each output of building block is labeled with real number. The main part of this number defines the code of building block and the fractional part determines the position of output in building block. For example, the 8th building block $\mathcal{B}_8$ located in 1st row and 1st column has 2 outputs. The first output is numbered as 8.0 and the second one as 8.1. The number of circuit outputs is defined by the number of outputs in the logic function implemented. Let us examine the encoding of the 14th building block represented as $< 22\ \ 0\ \ 2\ \ 2\ \ \{4\ \ 8.1\ \ 5\} >$. We refer to this representation of the building block as *building block genotype*. The functional gene defining the type of this gate is 22. This corresponds to the half adder in encoding table (Table 2). The examined cell has two inputs and two outputs. The first input is connected to the input x_4 and second to the second output of building block 8, labeled as 8.1. The inputs of building block 8 are connected to input variable x_1 and output of building block 5. The logic function of building block 5 depends on the input variables x_3 and x_4. Therefore the logic function implemented in building block 14 depends on three input variables: x_1, x_3 and x_4. The outputs of circuit are connected to the outputs of building blocks 13, 11 and 8.1.

3.3 Fitness Function

Our goal is to produce a fully functional design and minimize the number of building blocks actually used in circuit. A *fully functional design* produces the expected behavior stated by its truth table. Therefore, we decided to use two-stage fitness strategy [11]. At the beginning of search, only compliance with the truth table is taken into account. Once the first functional solution appears, we switch to a new fitness function in which fully functional circuits with less cost are rewarded. The *cost* or size of the fully functional circuit $\mathcal{N}$ is defined as

$$cost(\mathcal{N}) = \sum_{j=0}^{j<N_c*N_r-1} cost(\mathcal{B}_j) \tag{1}$$

and the cost of building block $\mathcal{B}_j$ is calculated as

$$cost(\mathcal{B}_j) = \begin{cases} N_j^p, & \mathcal{B}_j \text{ is committed building block} \\ 0, & \mathcal{B}_j \text{ is uncommitted building block.} \end{cases} \tag{2}$$

where N_j^p is the minimum number of primitive logic cells required to implement the logic function described behaviour of building block $\mathcal{B}_j$.

We consider the building block $\mathcal{B}_j$ as a sub-circuit with the structure that is not allowed to be changed. So, the cost of building block does not take into account whatever all outputs of building block has been involved or not. For example, let the two-bit multiplier be represented as building block $\mathcal{B}_j$ and the first digit of this two-bit multiplier be *only* involved in circuit $\mathcal{N}$. The cost of two-bit multiplier is 7 [7]. Despite the first digit of two-bit multiplier is implemented using only one primitive logic gate, the cost of building block $\mathcal{B}_j$ is 7. Members of the population with changed genotype have their fitness calculated.

3.4 Evolutionary algorithm

The circuit evolution has been performed using a rudimentary $(1+\lambda)$ evolutionary strategy (ES) with uniform mutation [19]. In this case a population of random chromosomes is generated and the fittest chromosome is selected. The new population is then filled with mutated versions of this.

Initialisation: The initial population is generated randomly. During initialisation of cell inputs and circuit outputs is performed in accordance with the levels-back constraint and the type of variables which are able to be present throughout all circuit. Thus if the logic constants are allowed as input connections throughout the circuit, then during initialisation procedure the inputs of gates can be chosen from the set of inputs constrained by levels-back or from the set of logical constants. The same procedure is true for the primary and inverted primary inputs.

Mutation: The circuit mutation allows us to change the following three features of the circuit: (1) Cell input; (2) Cell type; (4) Circuit output. Each of these parameters is considered as an elementary unit of the genotype. Cell type is defined by the functional gene and the functionality gene. The mutation rate defines how many genes in the population are involved in mutation. The chromosome contains 4 different types of genes, whose number is :

$$N_{genes} = \sum_{i=1}^{\lambda}(4 \cdot N_{gates}^{3} + N_{outputs}) \qquad (3)$$

where $N_{outputs}$ is the number of outputs in the circuit, N_{gates}^{i} is the number of gates in the i-th chromosome, λ is the population size.

4 Experimental Results

In this section we will consider some experimental results obtained for function- and gate-level EHW. Two applications to EHW approach has been examined: 1) Binary arithmetic circuit design; 2) Multi-valued arithmetic circuit design. For the purposes of this paper, 5 examples were chosen to illustrate our approach The performance of function-and gate- level EHW approaches has been compared. The initial data for the experiments is given in Table 3.

Table 3. Initial data

	Logic functions				
	binary			**multi-valued**	
Circuit	mult2	mult3	add2c	add3_3c	mult3_3
Circuit layout	10x1	30x1	15x1	10x1	10x1
Connectivity parameter	10	30	15	10	10
Population size	5	5	5	5	5
Number of generations	5 000	100 000	15 000	15 000	25 000
Number of GA runs	100	100	100	100	100
Mutation rate	5%	5%	5%	5%	5%

Two-bit full adder, add2c.pla. One of the principles used by human to construct larger adders is known as the *ripple-carry* principle. The block diagram for a two bit adder is shown in Fig. 5(a). Each of the blocks in Fig. 5(a) are identical to one-bit full adder. The two bit full adder evolved using one-bit full adder is produced by connecting the two smaller adders in a configuration identical to that shown in Fig. 5(a). This structure has been appeared in the most circuit designs (approximately 70 % of all fully functional designs) evolved using one-bit full adder as a building block. This demonstrates that the ES finds principles of the ripple-carry adder.

The circuit shown in Fig. 5 (b) has been evolved using one-bit full adder and two-bit multiplier as building blocks. This circuit requires 18 logic cells. Note that only third output of two bit multiplier is used. This shows that there is not necessary that all outputs of multi-input multi-output building blocks can be exploited in the circuit.

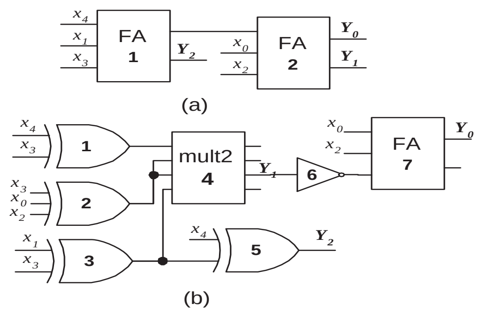

Fig. 5. Evolved 2-bit full adder; FA is one-bit full adder, mult2 is two-bit multiplier.

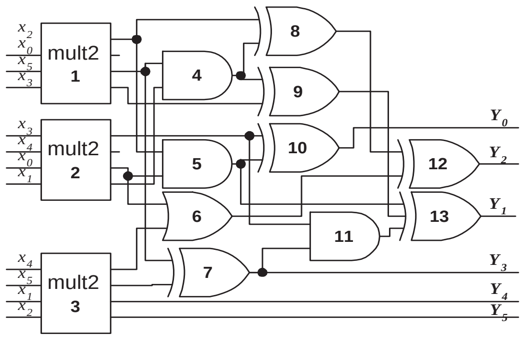

Fig. 6. Evolved three-bit multiplier (most efficient - 26 gates); mult2 is two-bit multiplier.

Three-bit multiplier, mult3.pla. The 3-bit multiplier can calculate the product of two integers a and b in range 0-7. The conventional 3-bit multiplier designed at gate-level requires 30 primitive logic gates [20]. The most efficient 3-bit multiplier evolved at gate-level requires only 26 gates [20]. Note that this structure contains 3 logic cells implemented AND logic function with one inverted input. In our approach we count only the number of NOT, AND, EXOR and OR gates. Therefore in our calculations, AND with one inverted input requires 2 primitive logic gates: AND and NOT. From this point of view the circuit structure reported in [20] contains 29 primitive logic gates, such as AND, OR, EXOR and NOT. It is more important to point out that this solution has been evolved after 3,000,000 generations, whereas in case of using the function-level EHW the fully functional solution has been evolved after 100,000 generations. In this particular example the evolution process has been improved in 30 times. It shows that using function-level EHW allows us to improve the ES performance.

The most efficient evolved 3-bit multiplier at function level is shown in Fig. 6. This circuit requires 32 gates. The cost of this circuit has been calculated regardless the outputs used in two-bit multiplier building blocks. Note that the second output of multipliers 1 and 2 is not used. This output requires 4 logic gates to be implemented. Only one logic gate is used to implement another output of two-bit multiplier ([21]). This means that 3 logic gates are employed to implement the second output and are not used in implementation of other circuit outputs. Therefore, the circuit shown in Fig. 6 requires 26 primitive logic gates (32-2*3=26), such as AND, OR and NOT. Therefore we can conclude that this is the most efficient circuit structure evolved using both gate and function level EHW.

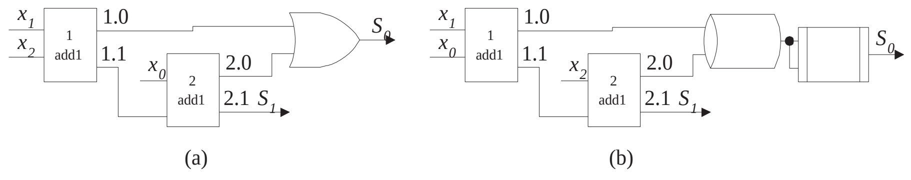

Fig. 7. Evolved 3-valued 1-digit full adders; add1 is 3-valued half adder.

One-digit 3-valued full adder, add3_3c.pla. Evolving a fully functional 3-valued one-digit adder using a circuit layout of 10 columns and 1 row with connectivity parameter equaled 10 proved to be relatively easy and the designs shown in Fig. 7 were obtained. The circuit shown in Fig. 7 (a) contains only 3 building blocks. Note that the optimal implementation of half adder contains 4 primitive logic cells [8]. Therefore the circuit in question requires 9 primitive logic cells. It is interesting to note that in this structure all outputs of half adder have been used actively. In case when the MAX gate is not allowed to be involved in evolution, the circuit structure shown in Fig. 7 (b) has been evolved. This structure contains 4 building blocks and requires 10 primitive logic cells. Note that both structures mentioned above use all outputs of half adder.

The 1.5-digit Multiplier, mult3_3.pla. An 1.5-digit multiplier multiplies the r-valued numbers (A_1A_0) by B_0 to produce the two-digit r-valued number (P_1P_0), where A_1, B_0 and P_1 are the most significant digits. Thus this is a circuit with 3 inputs and 2 outputs and it requires 27 input and output conditions for full specification in case of 3-valued logic. An example of circuit structures evolved using proposed method are shown in Fig. 8. It is interesting to note that in case of circuit shown in Fig. 8(a) ES uses the outputs of half adder and one-digit multiplier as sub-functions and some of its outputs are not in use. Thus the first output of 3-valued 1-digit multiplier labeled as 6 does not employed. At the same time all outputs of half adder are used. The circuit contains 7 building blocks and involves 12 primitive logic cells. Note that the most efficient 1-digit multiplier evolved requires 3 primitive logic cells. Fig. 8(b) shows the circuit evolved using only half adder. This circuit requires 6 building blocks and 12 primitive logic cells. Note that the implementation of digit P_1 is the same for both cases. It is interesting to note that the circuit evolved with half adder and one-digit multiplier requires less number of primitive logic cells then the alternative logic function evolved using only half adder. The circuits shown in Fig. 8 can not be obtained using the rules of standard multiplication process.

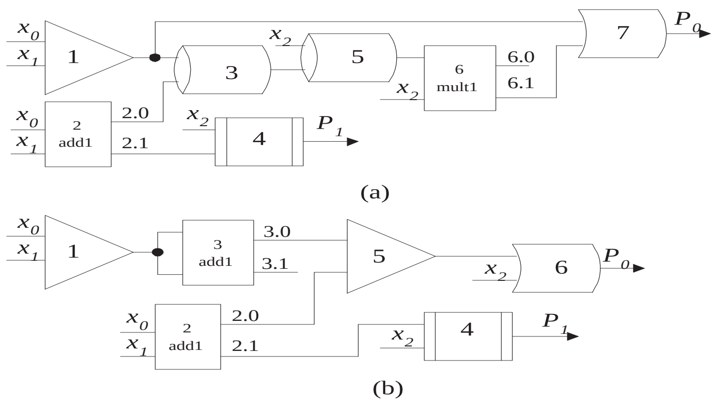

Fig. 8. Evolved 3-valued 1.5-digit multiplier. add1 is a 3-valued half adder; mult1 is 3-valued 1-digit multiplier.

Binary circuit design In this section we will discuss some experimental results obtained for two-bit full adder (add2c.pla), two- (mult2.pla) and three- bit multipliers mult3.pla evolved at gate and function level EHW. The ES performs the fixed number of generations for both approaches. The functional set of logic gates for gate-level EHW is a subset of {AND, OR, EXOR, NOT}. Note that using functional set of NOT, AND, OR, EXOR(i.e. 2-7-8-9 according to encoding

Table 4. Comparison function- and gate-levels EHW. **av.F1** is the mean functionality fitness function of the best chromosomes obtained 100 runs; **av.F2** is the mean number of active logic gates in the best chromosomes obtained after 100 runs; **av.100F2** is the mean number of active logic gates in the fully functional designs evolved; **#100% cases** is the number of fully functional circuits evolved.

Circuit	n	m	**Functional set**	**av.F1**	**av.F2**	**av.100F2**	**#100% cases**
mult2.pla	4	4	2-7-8-9	98.2968	7.22	7.14	36
			2-7-8-9-22	98.2031	11.56	8	37
			2-7-8-9-17	97.7344	15.51	8.75	16
			2-7-8-9-17-22	98.3438	15.83	10.9355	31
mult3.pla	6	6	2-7-8-9	95.5686	32.2667	0	0
			2-7-8-9-18	97.1807	59.6087	33.3	**3**
			2-7-8-9-18-22	98.0547	58.8	42	**5**
			2-7-8-9-17-18-19-22	99.5734	95.1525	65.33	**6**
add2c.pla	5	3	2-7-8-9	93.75	10.25	11.1429	14
			2-7-8-9-22	93.9167	12.62	12.5833	**24**
			2-7-8-9-17-22	99.6042	10.69	10.4375	**96**
			2-7-8-9-17-18-22	99.7708	11.74	11.2128	**94**
add3_3c	3	2	2-7-8-11-12	63.5185	8.44	0	0
			2-7-8-11-12-22	98.7778	9.12	8.86	**77**
			2-7-8-9-10-11-12	98.7593	6.97	7	72
			2-7-8-9-10-11-12-22	99.7593	6.77	7	**96**
			2-9-10-11-12	86.1852	5.31	8	1
			2-9-10-11-12-16-22	99.2222	7.87	8.069	**86**
mult3_3	3	2	2-7-8-9-10-11-12	76.5556	6.64	0	0
			2-7-8-9-10-11-12-22	98.3889	8.4	9.39	**61**
			2-7-8-9-10-11-12-16-22	97.5741	8.69	9.56	**41**
			1-7-8-11-12	59.2592	8.4	0	0
			1-7-8-11-12-16-22	92.1851	13.89	12.4	**10**
			2-9-10-11-12	77.9444	8.3	0	0
			2-9-10-11-12-22	93.9999	7.94	9.8	**26**
			2-9-10-11-12-16-22	96.5741	8.13	10.11	**17**

table (Table 2)) in evolution corresponds performing EHW at gate-level. The number of inputs in building block for gate-level EHW can not be more then 2. Half bit adder, one-bit full multiplier, two-bit multiplier together with primitive logic gates have been used at function-level EHW. The experimental results obtained using initial data shown in Table 3 are summarized in Table 4. The number of fully functional solutions evolved at function-level EHW is shown in bold (4).

Let us consider how ES performs at gate- and function-level EHW during evolution of two-bit multiplier (4). Analysing these data we can conclude that in terms of the number of active primitive logic gates used in circuit, the gate- and function-level EHW perform better. Thus, the average number of active logic gates (av.100F2) in fully functional circuits evolved at gate-level is 7.14 and at function-level is 8 or more. In terms of the number of fully functional circuits evolved during 100 ES runs, both methods perform nearly the same: 31-37 fully functional designs have been evolved during 100 ES runs. Considering ES performance for three-bit multiplier carried out at gate- and function-level we can conclude that the function-level EHW executes much better. Thus during evolution at gate level the average functionality of evolved circuits is 95.5686 and at function level it has been significantly increased to 99.5734. No fully functional solutions have been evolved for three-bit multiplier at gate-level EHW. Some functional solutions has been found using function-level EHW. Comparing function- and gate-level EHW used to evolve two-bit full adder, we find that the number of fully functional designs obtained at function-level EHW has been significantly improved in comparison with the similar EHW performed at gate-level. The average functionality fitness function is higher for function-level EHW rather then for gate-level EHW.

Note that in all cases mentioned above the average best functionality fitness functions (av.F1) for tested logic functions we can conclude that the best functionality fitness function is lower in case when the gate-level EHW has been applied. Analysing the average number of active primitive logic gates in best fully functional chromosomes (av.100F2), we find that there is no significant difference between function- and gate-level EHW in case when three-bit multiplier and two-bit full adder have bit evolved. Note that in case of evolving the two-bit multiplier, the ES at gate-level performs better in terms of the number of active primitive logic gates in circuit. Thus, we can conclude that function-level EHW performs better then gate-level EHW in terms of the number of fully functional binary circuits evolved when the suitable functional set of logic gates has been chosen.

Multi-valued circuit design. The similar experimental results have been carried out for 3-valued one-digit full adder (add3_3.pla) and 3-valued 1.5 digit multiplier (mult3_3.pla).

Let us consider 3-valued one-digit full adder. Three different functional sets of primitive logic gates have been employed during gate-level EHW execution. At function-level EHW, the 3-valued half adder has been added to these functional sets. In all cases the function-level EHW method performs better. Thus, when we use functional set of {COMPLEMENT, MIN, MAX, TSUM, TPRODUCT} no fully functional solutions have been evolved. Adding only half digit adder in functional set improves the GA performance in 77 times. The same conclusion we can made about other functional sets of logic gates used. Three different functional sets of primitive logic gates have been used at gate-level EHW. Half adder and one-digit multiplier have been added to functional set of logic gates at function-level EHW. Analysis of 1.5 digit multiplier allows us to make the conclusion that the function-level EHW performs better in term of the number of fully functional circuits evolved. Note that no fully functional designs have been evolved at gate-level. This number has been significantly increased when function-level evolution has been applied.

Thus, we can conclude that the function-level EHW approach applied to multi-valued logic design performs better then the similar approach implemented at gate-level if the suitable functional set of logic cells has been chosen.

5 Conclusion

We have introduced a new representation of logic cell in extrinsic EHW. This representation allows us to evolve circuits at function-level evolvable hardware. WE showed that this approach can be applied to synthesize both multiple-valued and binary logic functions. The advantage of proposed method is that it does not restricted by the radix of logic or the set of logic cells (functions) chosen to describe the behaviour of building blocks.

We have compared the ES performance for function- and gate-level EHW methods applied to design binary and multi-valued logic circuits. The obtained results show that the function-level EHW performs better in terms of the number of fully functional solutions evolved. Also it has been shown that the most efficient 3-bit multiplier evolved at function level evolution contains 26 primitive logic gates. This design less the number of active primitive logic gates then the similar conventional one. Also it has been shown that the evolved design contains less the number of primitive logic gates such as NOT,AND, EXOR and OR then the similar most efficient design evolved at gate level [20]. The experimental results show that the proposed chromosome representation allows us easier to evolve arithmetic MVL functions if the appropriate set of standard logic functions has been chosen.

The future work can be focused on the using multi-input multi-output automatically defined logic functions as a building blocks in proposed method as well as attempts to evolve logic functions of large number of variables.

References

1. Stoica A., Keymeulen D., Tawel R., Salazar-Lazaro C., and Li W. Evolutionary experiments with a fine-grained reconfigurable architecture for analog and digital cmos circuits. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 76–84. IEEE Computer Society, July 1999.
2. Murakawa M., Yoshizawa S., Kajitani I., Furuya T., Iwata M., and Higuchi T. Hardware evolution at function level. In *Proc. of the Fifth International Conference on Parallel Problem Solving from Nature (PPSNIV)*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 1996.
3. Higuchi T., Murakawa M., Iwata M., Kajitani I., Liu W., and Salami M. Evolvable hardware at function level. In *Proc. of IEEE 4th Int. Conference on Evolutionary Computation, CEC'97*. IEEE Press, NJ, 1997.
4. Kitano H. Morphogenesis for evolvable systems. In Sanchez E. and Tomassini M., editors, *Towards Evolvable Hardware. The Evolutionary Engineering Approach.*, volume 1062 of *Lecture Notes in Computer Science*, pages 99–117. Springer-Verlag, 1996.

5. Zebulum R., M. Vellasco, and M. Pacheco. Evolvable hardware systems: Taxonomy, survey and applications. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 344–358, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.
6. Murakawa M., Yoshizawa S., and Higuchi T. Adaptive equalization of digital communication channels using evolvable hardware. In *Proc. Of the 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 379–389, Tsukuba, Japan, 1996. Springer-Verlag, Heidelberg.
7. Miller J. F., Thomson P., and Fogarty T. C. Genetic algorithms and evolution strategies. In Quagliarella D., Periaux J., Poloni C., and Winter G., editors, *Engineering and Computer Science: Recent Advancements and Industrial Applications.* Wiley, 1997.
8. Kalganova T., Miller J., and Fogarty T. Some aspects of an evolvable hardware approach for multiple-valued combinational circuit design. In Sipper M., Mange D., and Perez-Uribe A., editors, *Proc. Of the 2nd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES'98)*, volume 1478 of *Lecture Notes in Computer Science*, pages 78–89, Lausanne, Switzerland, 1998. Springer-Verlag, Heidelberg.
9. Damiani E., Tettamanzi A.G.B., and Liberali V. On-line evolution of fpga-based circuits: A case study on hash functions. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 26–33. IEEE Computer Society, July 1999.
10. Aguirre A.H., Coello Coello C.A., and Buckles B.P. A genetic programming approach to logic function sunthesis by means of multiplexer. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 46–53. IEEE Computer Society, July 1999.
11. Kalganova T. and Miller J. Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 54–63. IEEE Computer Society, July 1999.
12. Masher J., Cavalieri J., Frenzel J., and Foster J.A. Representation and robustness for evolved sorting networks. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 255–261. IEEE Computer Society, July 1999.
13. Miller J. On the filtering properties of evolved gate arrays. In Stoica A., Keymeulen D., and Lohn J., editors, *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, pages 2–11. IEEE Computer Society, July 1999.
14. Kalganova T., Miller J., and Lipnitskaya N. Multiple-valued combinational circuits synthesized using evolvable hardware approach. In *Proc. of the 7th Workshop on Post-Binary Ultra Large Scale Integration Systems (ULSI'98) in association with ISMVL'98*, Fukuoka, Japan, May 1998. IEEE Press.
15. Utsumi T., Kamiura N., Nata Y., and Yamato K. Multiple-valued programmable logic arrays with universal literals. In *Proc. of the 27th Int. Symposium on Miltiple-Valued Logic (ISMVL'97)*, pages 169–174, Nova Scotia, Canada, May 1997. IEEE-CS-Press.
16. Hata Y. and K. Yamato. Multiple-valued logic functions represented by tsum, tproduct, not and variables. In *Proc. of the 23th Int. Symposium on Miltiple-Valued Logic (ISMVL'93)*, pages 222–227. IEEE-CS-Press, May 1993.
17. D. C. Rine. *An Introduction to Multiple-Valued Logic.* North-Holland, Amsterdam, 1977.

18. Miller J. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 1 of *ISBN 1-55860-611-4*, pages 1135–1142, Orlando, USA, July 1999. Morgan Kaufmann, San Francisco, CA.
19. Miller J. Digital filter design at gate-level using evolutionary algorithms. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO'99)*, volume 1 of *ISBN 1-55860-611-4*, pages 1127–1143, Orlando, USA, July 1999. Morgan Kaufmann, San Francisco, CA.
20. Miller J.F., Kalganova T., Lipnitskaya N., and Job D. The genetic algorithm as a discovery engine: Strange circuits and new principles. In *Proc. of the AISB'99 Symposium on Creative Evolutionary Systems, CES'99*, ISBN 1-902956-03-6, pages 65–74. Edinburgh, UK, The Society for the Study of Arificial Intelligence and Simulation of Behaviour, April 1999.
21. Coello C. A., Christiansen A. D., and Hernndez A. A. Use of evolutionary techniques to automate the design of combinational circuits. *International Journal of Smart Engineering System Design*, 1999.