Evolutionary Development of Generic Multipliers: Initial Results

Michal Bidlo
Brno University of Technology
Faculty of Information Technology
Božetěchova 2, 61266 Brno, Czech republic
bidlom@fit.vutbr.cz

Abstract

A system is presented utilizing a simple genetic algorithm combined with a developmental model for the evolutionary design of generic structures of combinational multipliers. An artificial environment is introduced into the system interpreted as an external control of the developmental process allowing to design irregular structures (inspired by the irregularity observed in conventional multipliers). Two sorts of experiments were conducted in order to demonstrate the ability of the system to adapt to different environments. The approach presented in this paper poses the first case in the field of the evolutionary design when generic multipliers have been evolved by means of the development.

1. Introduction

Combinational multipliers represent a class of circuits which are generally hard to design using evolutionary techniques and the computing resources available nowadays. Typically, the search space is extremely large and rugged and the design is usually not scalable in case of low-level representations, e.g. gate-level design using traditional Cartesian Genetic Programming (CGP) [9]. The problem of scale, constituting a major issue in the evolutionary design of digital circuits, prevents from finding good solutions in reasonable time both due to the size of the search space and the number of inputs of the target circuit itself because of enormously time-consuming evaluation process (fitness calculation) in case of large circuits. Therefore, more effective representations have been investigated in order to overcome these issues and improve the scalability and evolvability of multipliers and other digital circuits as summarized in the following paragraph.

Miller et al summarized the principles in the evolutionary design of digital circuits and showed some results of evolved combinational arithmetic circuits, including multipliers, in [7]. A detailed study of the fitness landscape in case of the evolutionary design of combinational circuits using Cartesian Genetic Programming is proposed in [8]. 3×3 multipliers constitute the largest and most complex circuits designed by means of traditional CGP in these papers. Vassilev et al utilized a method based on CGP which exploits redundancy contained in the genotypes. Larger (up to 4×4 bits) and more efficient multipliers were evolved by means of this approach in comparison with the conventional designs [15]. Vassilev and Miller studied the evolutionary design of 3×3 multipliers by means of evolved functional modules rather than only two-input gates [16]. Their approach is based on Murakawa's method of evolving sub-circuits as the building blocks of the target design in order to speed up and improve the scalability of the design process [10]. Torresen applied the partitioning of the training vectors and the partitioning of the training set approach (so-called increased complexity evolution or incremental evolution) for the design of multiplier circuits. His approach was focused on improving the evolution time and evolvability rather than optimizing the target circuit. 5×5 multipliers were evolved using this method [14]. Stomeo et al devised a decomposition strategy for evolvable hardware which allows to design large circuits [13]. Among others, 6×6 multipliers were evolved by means of this approach. Aoki et al introduced an effective graph-based evolutionary optimization technique called Evolutionary Graph Generation [2]. The potential capability of this method was demonstrated through experimental synthesis of arithmetic circuits with different levels of abstraction. 16×16 multipliers were evolved using word-level arithmetic components (such as one-bit full adders or one-bit registers). Note that, in addition to these papers, evolution of different digital circuits, including combinational multipliers, was performed in order to investigate fault tolerance and other important features rather than to design large circuits, e.g. [6, 5, 4].

The goal of this paper is to utilize an artificial developmental model based on application-specific instructions in connection with the genetic algorithm in order to (1) reduce

the length of the chromosome needed for encoding the target circuits, (2) introduce an external control of the developmental process in form of a string of values interpreted as a biologically inspired environment for the construction of irregular structures, (3) enable to design generic combinational multipliers and (4) demonstrate the ability of the system to adapt to different environments retaining the capability to design generic structures. A concept of environment was proposed in [3], where the environment was utilized to affect the function of polymorphic circuits [1]. In this paper the environment is intended to influence the development of the circuit structure. The development of generic multipliers is inspired by the preceding research in which similar instruction-based developmental model was applied for iterative design of generic sorting networks [12]. In that case innovative algorithms were discovered in comparison with the conventional methods.

The approach presented in this paper differs from the existing methods of the evolutionary design of multipliers particularly in the following aspects. (1) The work is focused on the design of *arbitrarily large* multipliers rather than on optimizing their structure. The existing methods usually have dealt with a fixed number of inputs, possibly a limit of the circuit size has been involved for keeping a reasonable evolution time. (2) A developmental model is utilized in this paper in combination with an evolutionary algorithm to design the generic multipliers. Liu et al, for example, also applied a biological development model for the design of robust multipliers [6], however, only 2-bit multipliers were involved in the experiments. Moreover, his work was focused on designing robust fault-tolerant circuits rather than generic structures which represents the main difference in comparison with this paper. (3) Our approach introduces a kind of external environment in order to investigate an ability of adaptation of the developmental process and to create different general multiplier structures for distinct forms of environment.

2. The Development

In nature, the development is a biological process of ontogeny representing the formation of a multicellular organism from a zygote. It is influenced by the genetic information of the organism and the environment in which the development is carried out.

In the area of computer science and evolutionary algorithms in particular, the computational development has been inspired by that biological phenomena. Computational development is usually considered as a non-trivial and indirect mapping from genotypes to phenotypes in an evolutionary algorithm. In such case the genotype has to contain a prescription for the construction of target object. While the genetic operators work with the genotypes, the fitness calculation (evaluation of the candidate solutions) is applied on phenotypes created by means of the development. The utilization of environment in the computational development may be understood as an external information (in addition to genetic information included in the genotype) and as an additional control mechanism of the development. The principles of the computational development together with a brief biological background are summarized in [11].

3. Development of Generic Multipliers

The method of the development is inspired by the construction of conventional combinational multipliers for which generic algorithms exist. Figure 1 shows a typical 4×4 combinational multiplier designed by means of the conventional approach [17]. It is evident that the first level of AND gates and the following sequence of adders are specific in comparison with the rest of the circuit, which poses a kind of irregularity. However, the rest of the circuit exhibits regular sequences which can be expressed by means of iterative algorithm utilizing variables. Moreover, the whole design can be easily parametrized by means of the width (the number of bits) of the operands. Theferore, this concept is assumed to be convenient for the design of generic multipliers using development and evolutionary algorithm.

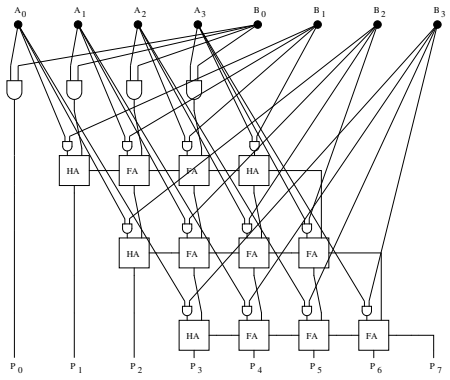

Figure 1. A 4×4 conventional combinational multiplier. $\mathbf{A_0} \ldots \mathbf{A_3}, \mathbf{B_0} \ldots \mathbf{B_3}$ represent the bits of the operands, $\mathbf{P_0} \ldots \mathbf{P_7}$ denote the bits of the product.

A *building block* represents a basic component of the circuit to be developed. The general structure of the block is shown in Figure 2a. Each building block contains three inputs from which one or two may be unused depending on the type of the block. There are two outputs at each building block from which one may be meaningless, i.e. permanently set to logic 0, depending on the block type. The outputs are denoted symbolically as $out0$ and $out1$. In case of the block containing only one output, $out0$ represents

the effective output and $out1$ is permanently set to logic 0. The circuit is developed inside a *grid* (rectangular array) which proved to be a suitable structure for the the design of combinational multipliers (see Figure 2b). Figure 3 shows the set of building blocks utilized for the experiments presented in this paper. For the interconnection of the blocks the position (row, col) in the grid is utilized. The inputs of the blocks are connected to the outputs of the neighboring blocks by referencing the symbolic names of the outputs or via indices to the primary inputs of the circuit, depending on the block type. Feedback is not allowed. For example, $out1(row, col-1)$ means that the input of the block at the position (row, col) in the grid is connected to the output denoted $out1$ of the block on its left-hand side. The connections to the primary inputs are determined by the indices v_0 and v_1. Let $A = a_0a_1a_2, B = b_0b_1b_2$ represent the primary inputs (operands A and B) of a 3×3 multiplier. For instance, an AND gate with $v_0 = 1$ and $v_1 = 2$ has its inputs connected to the second bit (a_1) of operand A and the third bit (b_2) of operand B. In case of the building blocks at the borders of the grid (when $row = 0$ or $col = 0$), where no blocks with valid outputs occur (for $row-1$ or $col-1$), the appropriate inputs of the blocks at (row, col) are set to 0. In this way, for example, full adder (Fig. 3f) at $(0, 0)$ is degraded to AND gate, the buffer (Fig. 3b) at $(1, 0)$ becomes the source of logic 0 etc.

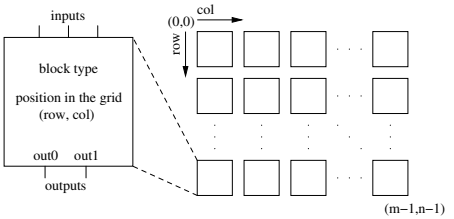

Figure 2. (a) Structure of a building block. (row, col) determines the position of the block in the grid – see part (b). The connection of the inputs depends on the type and position of the block. (b) Grid of the building blocks with m rows and n columns for the development of generic multipliers.

The development of the circuit is performed by means of a *developmental program*. This program, which is the subject of evolution, consists of simple application-specific *instructions*. The instructions make use of numeric literals $0, 1, \ldots, max\_value$, where max_value is specified by the designer at the beginning of evolution. In addition to the numeric literals, a *parameter* and some *variables* of the developmental system can be utilized. The parameter represents the width (the number of bits) of the operands – inputs

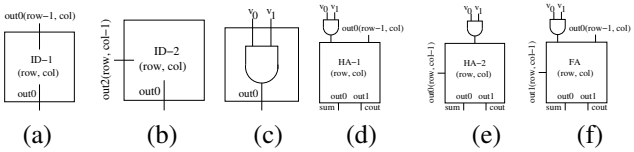

Figure 3. Building blocks for the development of combinational multipliers. (a, b) buffers, (c) AND gate, (d, e) half adders, (f) full adder. (row, col) denotes position in the grid. $\mathbf{v_0}$ and $\mathbf{v_1}$ determine indices of primary input bits. Connection of different inputs of the blocks are shown. Unused inputs and outputs are not depicted (they are considered as logic 0).

of the multiplier. The parameter is referenced by its symbolic name w, whose value is specified by the designer at the beginning of the evolutionary process. For example, in case of designing a 4×4 multiplier, the parameter possesses this value, i.e. $w = 4$. The values of parameter is invariable during the evolutionary process. There are four variables integrated into the developmental system denoted v_0, v_1, v_2 and v_3, whose values are altered by the appropriate instructions during the execution of the program (developmental process). Table 1 describes the instruction set utilized for the development. The SET instruction assigns a value determined by a numeric literal, parameter or another variable to a specified variable. Instructions INC, respective DEC are intended for increasing, respective decreasing the value of a given variable. The difference can be specified only by a numeric literal. Simple loops inside the developmental program are provided with the REP instruction whose first argument determines the repetition count and the second argument states the number of instructions after the REP instruction to be repeated. Inner loops are not allowed, i.e. REP instructions inside the repeated code are interpreted as NOP (no operation) instructions. The GEN instruction generates a building block of the type specified in the argument. Note that, if a block containing AND gate is generated (e.g. AND gate itself, FA), the inputs of the AND gate are connected to the primary inputs indexed by the values of variables v_0, v_1 as shown in Figure 3. In case when v_0 or v_1 exceeds the correct values, the appropriate input of AND gate is connected to logic 0. If (row, col) do not exceed the grid boundaries, the block is generated at that position, otherwise no block is generated. After executing GEN, col is increased by one.

In fact, the developmental program may consist of several *parts*, which may consist of different number of instructions. Let us define the length of a program (or a part of a program) as the number of instructions it is composed of. These parts are executed on demand with respect to an *en-*

Instruction	Arguments	Description
0: SET	$variable, value$	Assign $value$ to $variable$. $variable \in \{v_0, v_1, v_2, v_3\}$, $value \in \{0, 1, \ldots, max\_value, w, v_0, v_1, v_2, v_3\}$.
1: INC	$variable, value$	Increase $variable$ by $value$. $variable \in \{v_0, v_1, v_2, v_3\}$, $value \in \{0, 1, \ldots, max\_value\}$.
2: DEC	$variable, value$	If $variable \geq value$, then decrease $variable$ by $value$. $variable \in \{v_0, v_1, v_2, v_3\}$, $value \in \{0, 1, \ldots, max\_value\}$.
3: REP	$count, number$	Repeat $count$-times $number$ following instructions. All REP instructions in $number$ following ones are interpreted as NOP instructions (inner loops are not allowed).
4: GEN	$block$	Generate $block$ on the actual position (row, col); increase col by 1.
5: NOP		An empty operation.

Table 1. Instructions utilized for the development

vironment. A single execution of a part of program is referred to as a *developmental step*. The meaning of the environment is to enable the system to develop more complex structures which may not be fully regular. The environment is represented by a finite sequence of values specified by the designer at the beginning of the evolution, e.g. $env = (0, 1, 2, 2)$. The number of different values in the environment usually equals the number of parts of the developmental program. In addition, there is an *environment pointer* (let us denote it e) determining a particular value in the environment during the development time. Each part of the program is executed deterministically, sequentially and independently on the others according to the environment values. However, the parameter and the variables of the developmental system are shared by all the parts of program.

At the beginning of the evolutionary process the value of the parameter w and the form of the environment env are defined by the designer. By the inspiration from conventional multipliers the number of developmental steps needed for creating a working multiplier and the length of the environment will correspond to w. The developmental program, whose number of parts and their lengths are also specified a priori by the designer, is intended to operate over these data in order to develop multiplier of a given size. As evident, the different sizes of multipliers are created by setting the parameter and adjusting the environment. Hence the circuit of a given size is always developed from scratch; it is a case of *parametric developmental design*. The following algorithm will be defined in order to handle the developmental process.

1. Initialize $row, col, v_0, v_1, v_2, v_3$ and e to 0.
2. Execute $env(e)$-th part of program.
3. Increase e and row by 1, set col to 0.
4. If neither e, nor row exceed, go to step 2.
5. Evaluate the resulting circuit.

4. Evolutionary System Setup

For the experiments presented herein a simple genetic algorithm was utilized in combination with the developmental system described in Section 3.

A chromosome consists of a linear array of the instructions, each of which is represented by the operation code and two arguments (the utilization of the arguments depends on the type of the instruction). The array contains n parts of the developmental program stored in sequence, whose lengths (the number of instructions) correspond to $l_0, l_1, \ldots, l_{n-1}$. The number of the parts and their lengths are determined by the designer. In general, the structure of a chromosome can be expressed as $i_{0,0}i_{0,1}\ldots i_{0,l_0-1}; \ldots; i_{n-1,0}i_{n-1,1}\ldots i_{n-1,l_{n-1}-1}$, where $i_{j,k}$ denotes the k-th $i$nstruction of j-th part of program for $k = 0, 1, \ldots, l_j - 1$ and $j = 0, 1, \ldots, n-1$. During the application of the genetic operators the parts of the program are not distinguished, i.e. the chromosome is handled as a single sequence of instructions. The chromosomes possess constant length during the evolution. The population consists of 32 chromosomes which are generated randomly at the beginning of evolution. Tournament selection operator of base 2 is utilized.

Mutation of a chromosome is performed by a random selection of an instruction followed by a random choosing a part of the instruction (operation code or one of its arguments). If the operation code is mutated, entire new instruction will replace the original one, otherwise one of its arguments is mutated. The mutation is performed with the probability 0.03, only one instruction per chromosome is muated.

A special crossover operator was applied which exhibits a significant positive influence on the evolutionary process in comparison with standard one-point or uniform crossover or with the case when no crossover is utilized. Two parent chromosomes are selected and an instruction is selected randomly in each of them (i_1, i_2). A position (index) is chosen randomly in each of the two offspring (c_1, c_2). After the

crossover, the first, respective the second offspring contains the original instructions from the first, respective the secont parent with the exception of i_1, respective i_2, which is put at the position c_2 in the second offspring, respective c_1 in the first offspring. The crossover occurs with the probability 0.9 and is illustrated in Figure 4.

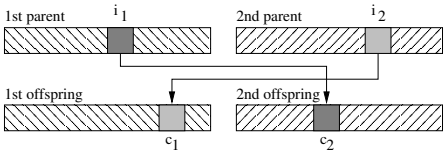

Figure 4. Crossover of two chromosomes. $\mathbf{i_1, i_2}$ denote the instructions to be crossed, $\mathbf{c_1, c_2}$ pose the offspring positions the instructions will be placed to.

The fitness function is calculated as the number of bits processed correctly by the multiplier developed by means of the program stored in the chromosome. The experiments were conducted with the evolution of programs for the construction of 4×4 multipliers, i.e. the parameter $w = 4$. There are $2^{4+4} = 256$ possible test vectors and the multipliers produce 8-bit results. Therefore, the maximum fitness representing a working solution equals $256 \cdot 8 = 2048$. If a working solution is not evolved in two millions of generations in case of the first sort, possibly in one million of generations in the second sort of experiments (see the next section), the evolution is restarted with new population. After the evolution the resulting program is verified in order to determine whether it is able to create larger multipliers, typically up to the size 14×14 bits. This size of circuit was determined experimentally, allowing to perform a sufficient number of developmental steps for demonstrating the correctness of the evolved program and keeping a reasonable verification time. If a program shows this ability, it will be considered as general.

5. Experimental Results and Discussion

The experiments were conducted on a common PC equipped with a 2.0 GHz processor, 512 MB RAM and running Gentoo Linux, kernel 2.6.18-r6. The evolution of a single solution required 15–20 minutes in average.

In the first sort of experiments 3-part programs (6+12+12 instructions) were evolved utilizing the environment $env = (0, 1, 2, 2)$ for controlling the development. 1000 independent experiments were conducted from which 67% working solutions (i.e. the programs for constructing 4×4 multipliers) were evolved and 18% of them were classified as general programs. Figure 5 shows a multiplier together with detailed logic schemes of the building blocks (half adder from Fig. 3e and full adder from Fig. 3f) involved by the evolutionary algorithm. This multiplier was constructed by means of one of the most efficient programs that was evolved in this sort of experiments. The program is shown in Table 2. Let us go through the program in order to understand the developmental process.

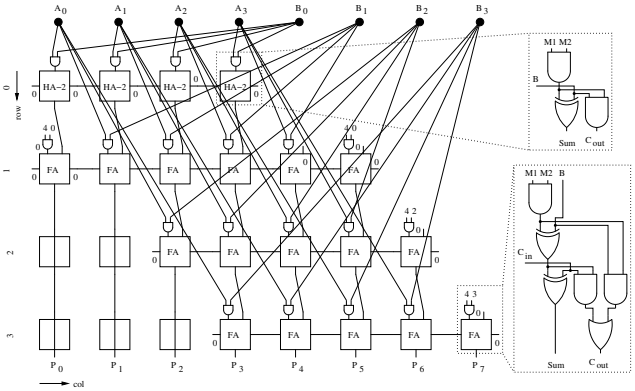

Figure 5. A 4×4 multiplier created by means of evolved program shown in Table 2 using the environment $\mathbf{env} = (\mathbf{0, 1, 2, 2})$. $\mathbf{A_0 \ldots A_3, B_0 \ldots B_3}$ represent the bits of the operands, $\mathbf{P_0 \ldots P_7}$ denote the bits of the product. Logic schemes of the half adder and full adder-based building blocks utilized by the evolved program are shown. M1 and M2 denote the multiplicands whose partial product represents the first operand of the full adder, B denotes the second operand, $\mathbf{C_{in}}$ poses the input carry, Sum and $\mathbf{C_{out}}$ represent the resulting sum and output carry.

At the beginning of the development, the following setup is specified by the designer: $w = 4, env = (0, 1, 2, 2)$. The following initialization is performed by the system: $v_0 = 1, v_1 = 0, v_2 = 0, v_3 = 0, row = 0, col = 0, e = 0$.

At this point $env(e) = 0$, therefore, Part 0 of the program will be executed. The instruction 0 should repeat zero times instructions 1 and 2 (because $v_1 = 0$), therefore, this code has no effect. Instruction 3 will repeat four times (because $w = 4$) instructions 4 and 5 which create row 0 of the multiplier (blocks HA-2) with the inputs of AND gates of these blocks connected to the primary inputs (operands of the multiplier) specified by the actual values of v_0 and v_1. While v_1 retains 0, v_0 is increased by 1 by instruction 5 and col is increased by 1 automatically by the system in each pass (in general, after executing a GEN instruction). Since there are no more instructions to be executed in Part 0, the system increases row and e by 1 and the construction of row 0 of the circuit is finished. Note that the variables

Line	Part 0	Part 1	Part 2
0:	REP v_1 2	GEN FA	REP v_1 2
1:	GEN FA	SET v_3 0	REP v_0 2
2:	INC v_0 1	SET v_0 v_3	GEN ID-1
3:	REP w 2	INC v_1 1	GEN ID-1
4:	GEN HA-2	REP w 2	INC v_0 1
5:	INC v_0 1	GEN FA	INC v_1 1
6:		INC v_0 1	REP w 1
7:		SET v_1 0	SET v_0 v_2
8:		GEN FA	REP w 2
9:		DEC v_1 0	GEN FA
10:		INC v_1 1	INC v_0 1
11:		REP v_0 2	GEN FA

Table 2. Evolved general program by means of which the multiplier from Figure 5 was created. In this case $\mathbf{w} = 4$, the program consists of 3 parts executed according to the environment $\mathbf{env} = (\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{2})$.

hold their actual values, i.e. $v_0 = 4$ and the others equal 0.

Now, $env(e) = 1$ for $e = 1$, therefore, Part 1 of the program will be executed in order to develop row 1 of the multiplier. Instruction 0 of Part 1 generates full adder (FA block), where the inputs of AND gate of this block should be connected to bits 4 and 0 of the operands (according to the variables $v_0 = 4, v_1 = 0$). Note, since v_0 exceeds the operand width, the first input of AND gate of this FA block will be considered as logic 0 causing permanent logic 0 at the output of the AND gate, i.e. the AND gate of this block is meaningless (see Fig. 5). Instructions 1 and 2 actually set v_0 to 0. Then, v_1 is increased by 1 by instruction 3. Instructions 4, 5 and 6 generate four FA blocks with the inputs of AND gates of these blocks connected to the appropriate operand bits. Note that instruction 7 sets v_1 to 0 which, in fact, voids the result of instruction 3. An FA block is generated by instruction 8 (again, its AND gate is meaningless). Instruction 9, decreasing v_1 by 0, has no effect, v_1 is increaded by 1 by instruction 10 and instruction 11 is meaningless since there is no instructions to repeat. Row 1 is completed with the actual values of $v_0 = 4, v_1 = 1$ and other variables possessing zeros.

The row 2 of the circuit will be constructed using Part 2 of the program according to the next environment value $env(e) = 2$ for $e = 2$. Instruction 0 initiates a loop repeating once instructions 1 and 2. Instruction 1 is interpreted as no operation because inner loops are not allowed and instruction 1 generates an ID-1 block. In addition, instruction 3 creates one more ID-1 block in the next column. Value of v_0, respective v_1 is increased by one by instruction 4, respective 5. In fact, the only effect of the loop initiated by instruction 6, repeating instruction 7, is to set v_0 to 0 (according to v_2 which equals 0). This operation actually voids the result of instruction 4. Four FA blocks are generated by instruction 9 inside the loop started by instruction 8. Instruction 9, which is also a part of the loop body, determines the connection of the inputs of AND gates generated inside these blocks. The last instruction 11 generates an FA block with a redundant AND gate. Now row 2 is finished. The variables $v_0 = 4, v_1 = 2$ and the other ones equal 0.

According to $env(e) = 2$ for $e = 3$ the last row of the circuit will be generated by executing Part 2 of the program. The development proceeds in the same way as described in the previous paragraph, considering the values of variables resulted from the previous developmental step.

This program showed the ability to construct generic multipliers. Note that, in general, for w-bit operands the environment would have the form $env = (0, 1, 2, \ldots, 2)$ containing $w - 2$ twos and w developemntal steps would be needed to construct a working multiplier.

It is evident that the multiplier shown in Fig. 5 could be optimized considering the inputs of the building blocks. For instance, half adders in row 0 of the circuit can be replaced by simple AND gates since the first input of these adders are permanently connected to logic 0. Similarly, full adders at positions (1, 1), (1, 4), (2, 2) and (3, 3) actually represents half adders and full adders at positions (1, 0), (1, 5), (2, 6) and (3, 7) can be replaced by identity functions. In fact, the circuit corresponds to the conventional multiplier after this optimization (compare with Figure 1).

The second sort of experiments was devoted to evolutionary design of 1-part developmental program consisting of 10 instructions. A new form of the environment was specified in order to demonstrate the adaptation of the program being evolved to the new conditions of creating generic multipliers. Again, 1000 independent experiments were conducted from which 97% working solutions were obtained. 85% of the evolved programs were classified as general. An evolved 4×4 multiplier adapted to the new environment $env = (0, 0, 0, 0)$ is shown in Figure 6. Table 3 shows the appropriate developmental program. This program showed the ability to construct generic multipliers. Note that, in general, for w-bit operands the environment would have the form $env = (0, \ldots, 0)$ containing w twos and w developemntal steps would be needed to construct a working multiplier.

Experiments for the evolution of 3×3 multipliers were conducted, however, no general solution was obtained. Although basic AND gates and ID functions were available in the set of building blocks, they were rarely used in the design and adders were generated instead. This behavior could be explained by predominating occurence of adders which pushes the evolution to design regular structures, utilizing the properties of the building blocks and their interconnection. The evolved programs exhibit certain degree

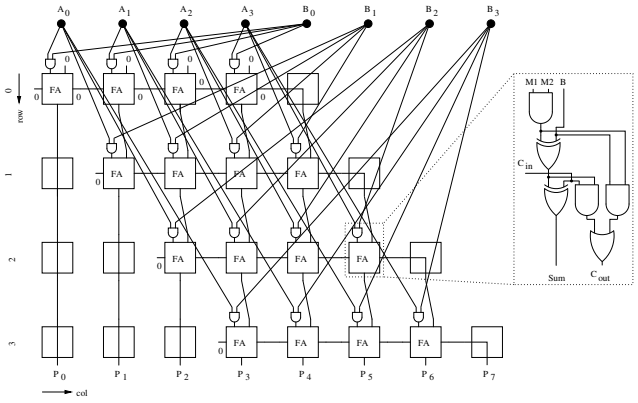

Figure 6. A 4×4 multiplier created by means of evolved program shown in Table 3 adapted to the environment $\mathbf{env} = (0, 0, 0, 0)$. $\mathbf{A_0 \ldots A_3, B_0 \ldots B_3}$ represent the bits of the operands, $\mathbf{P_0 \ldots P_7}$ denote the bits of the product. Logic scheme of the fundamental full adder-based building block (see Fig. 3f) utilized by the evolved program is shown. M1 and M2 denote the multiplicands whose partial product represents the first operand of the full adder, B denotes the second operand, Sum and $\mathbf{C_{out}}$ represent the sum and output carry of the full adder.

of redundancy, which is caused by the determination of the program length based on the conventional design. Therefore, there is an additional posibility for reducing the search space. Despite the worse level of evolvability as seen in the progress of the average population fitness shown in Figure 7, a very good success rate was observed both in the case of the evolution of initial solutions and the occurence of general programs among these solutions after verification especially in the second sort of experiments, which indicates the suitability of the proposed representation to evolve generic structures. However, the selection of building blocks represent a crucial issue for successful evolution of this kind

0: REP v_1 1	4: INC v_0 1	8: SET v_0 v_2
1: GEN ID-1	5: INC v_3 0	9: GEN ID-2
2: REP p_1 2	6: INC v_1 1	
3: GEN FA	7: SET v_3 v_0	

Table 3. Evolved general program by means of which the multiplier from Figure 6 was created. In this case $w = 4$, there is only one program part operating in the environment $\mathbf{env} = (0, 0, 0, 0)$.

of circuits. Both the programs presented herein showed the ability to construct generic multipliers, which has never been seen before in the field of the evolutionary design.

An environment was integrated into the developmental model in order to allow the system to construct irregular structures (inspired by the conventional multipliers). The system demonstrated a capability of adaptation to another environment that allowed to design generic multipliers exhibiting a high level of regularity in their structures using a program consisting of only one part. Moreover, the adaptation was observed to many other irregular environments and even to random binary environments (i.e. the environments consisting only of values 0 and 1), retaining the ability of the system to develop *generic* multipliers by means of a single program, whose parts are executed according to the environment. Note that this feature is significantly influenced by the grid chosen for representing the circuits and by the general structure and properties of a building block, particularly the facility of degradation of more complex blocks (e.g. full adders) to simpler blocks (e.g. AND gates, ID functions etc.) according to the inputs of the blocks. However, this is a significant information with respect to the future research. For example, the development of generic combinational multipliers possessing exactly that structure shown in Figure 1 would not be possible without applying the environment. A variety of building blocks exist which could be involved in the design process in order to develop more complex generic circuits exibiting irregularities. Therefore, the approach utilizing a form of environment suggests a big space deserving of the subsequent investigation.

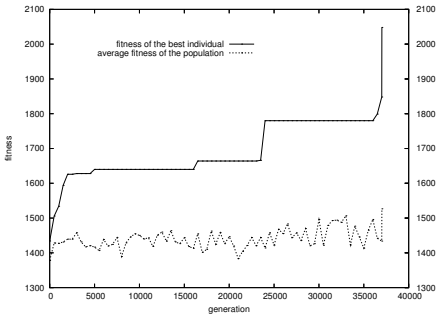

Figure 7. A typical progress of the fitness during the evolution of multipliers using the proposed developmental system

6. Conclusions

In the paper an evolutionary developmental method was presented for the design of generic multipliers. A specific form of environment was integrated into the developmental

model representing an external control of the developmental process which is intended as a tool enabling the design of irregular structures. The approach presented herein poses the first case of the evolutionary design of generic structures of multipliers using the development. Moreover, the environment was utilized in order to demonstrate adaptation of the development, retaining its ability to design generic multipliers. The experiments confirmed the capability of adaptation in connection with the proposed circuit representation. General programs were evolved for the construction of multipliers which exhibit a high degree of regularity in the circuit structure. Since the multipliers of different sizes are constructed every time from scratch by means of an evolved program, utilizing the bit width of the operands as a parameter for determining the circuit structure, it is a case of parametric developmental design.

Although the building blocks and the grid for the circuit representation were chosen with the inspiration from the structure of classical combinational multipliers, it is possible to adjust them for the development of other circuits. Experiments have been now conducted devoted to the design of so-called carry-save combinational multipliers which exhibit better properties, in particular the delay of the circuit, in comparison with the classical ones (see [17], pages 446–449 for details). In addition, more sophisticated circuits could be designed as the building blocks for the development of other classes of circuits, e.g. adders, medians, parity circuits and so on. The advanced experiments may even include the building blocks into the evolutionary process together with the developmental programs. Furthermore, fitness landscape could be investigated in different cases of circuits design by means of this system. These issues represent the central items for our future research.

Acknowledgement

This work was partially supported by the Grant Agency of the Czech Republic under contract No. 102/06/0599 *Methods of Polymorphic Digital Circuit Design*, No. 102/05/H050 *Integrated Approach to Education of PhD Students in the Area of Parallel and Distributed Systems* and the Research Plan No. MSM 0021630528 *Security-Oriented Research in Information Technology*.

References

[1] Adrian Stoica et al. Polymorphic electronics. In *Proc. of International Conference on Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science, volume 2210*, pages 291–302. Springer-Verlag, 2001.

[2] T. Aoki, N. Homma, and T. Higuchi. Evolutionary synthesis of arithmetic circuit structures. *Artificial Intelligence Review*, 20:199–232, 2003.

[3] M. Bidlo and L. Sekanina. Providing information from the environment for growing electronic circuits through polymorphic gates. In *Proc. of Genetic and Evolutionary Computation Conference – Workshops 2005*, pages 242–248. Association for Computing Machinery, 2005.

[4] M. Hartmann and P. C. Haddow. Evolution of fault-tolerant and noise-robust digital designs. *IEE Proceedings – Computers and Digital Techniques*, 151:287–294, 2004.

[5] M. Hartmann, P. C. Haddow, and F. Eskelund. Evolving robust digital designs. In *Proc. of The 2002 NASA/DoD Conference on Evolvable Hardware*, pages 36–45. IEEE Computer Society, 2002.

[6] H. Liu, J. F. Miller, and A. M. Tyrrell. A biological development model for the design of robust multiplier. In *Proc. of the 2nd European Workshop on Evolutionary Computation in Hardware Optimisation, EvoHOT 2005, Lecture Notes in Computer Science, volume 3449*, pages 195–204. Springer-Verlag, 2005.

[7] J. F. Miller and D. Job. Principles in the evolutionary design of digital circuits – part I. *Genetic Programming and Evolvable Machines*, 1(1):8–35, April 2000.

[8] J. F. Miller and D. Job. Principles in the evolutionary design of digital circuits – part i. *Genetic Programming and Evolvable Machines*, 3(2):259–288, July 2000.

[9] J. F. Miller and P. Thomson. Cartesian genetic programming. In *Proc. of the 3rd European Conference on Genetic Porgramming, Lecture Notes in Computer Science, vol 1802*, pages 121–132, Berlin Heidelberg New York, 2000. Springer.

[10] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at function level. In *Proc. of the 4th International Conference on Parallel Problem Solving from Nature, PPSN 1996, Lecture Notes in Computer Science, volume 1141*, pages 206–217, London, UK, 1996. Springer-Verlag.

[11] S. Kumar (ed.) and P. J. Bentley (ed.). *On Growth, Form and Computers*. Elsevier Academic Press, 2003.

[12] L. Sekanina and M. Bidlo. Evolutionary design of arbitrarily large sorting networks using development. *Genetic Programming and Evolvable Machines*, 6(3):319–347, 2005.

[13] E. Stomeo, T. Kalganova, and C. Lambert. Generalized disjunction decomposition for evolvable hardware. *IEEE Transactions on Systems, Man and Cybernetics – Part B*, 36:1024–1043, 2006.

[14] J. Torresen. Evolving multiplier crcuits by training set and training vector partitioning. In *Proc. of the 5th International Conference on Evolvable Systems: From Biology to Hardware, ICES 2003, Lecture Notes in Computer Science, volume 2606*, pages 228–237. Springer-Verlag, 2003.

[15] V. Vassilev, D. Job, and J. Miller. Towards the automatic design of more efficient digital circuits. In *Proc of the Second NASA/DoD Workshop on Evolvable Hardware*, pages 151–160, Palo Alto, CA, 2000. IEEE Computer Society.

[16] V. Vassilev and J. F. Miller. Scalability problems of digital circuit evolution. In *Proc. of the 2nd NASA/DoD Workshop of Evolvable Hardware*, pages 55–64, Los Alamitos, CA, US, 2000. IEEE Computer Society.

[17] J. F. Wakerly. *Digital Design: Principles and Practice*. Prentice Hall, New Jersey, US, 2001.