Evolutionary Design of Generic Combinational Multipliers Using Development

Michal Bidlo

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

Abstract. Combinational multipliers represent a class of circuits that is usually considered to be hard to design by means of the evolutionary techniques. However, experiments conducted under the previous research demonstrated (1) a suitability of an instruction-based developmental model to design generic multiplier structures using a parametric approach, (2) a possibility of the development of irregular structures by introducing an environment which is considered as an external control of the developmental process – inspired by the structures of conventional multipliers and (3) an adaptation of the developing structures to the different environments by utilizing the properties of the building blocks. These experiments have represented the first case when generic multipliers were designed using an evolutionary algorithm combined with the development. The goal of this paper is to present an improved developmental model working with the simplified building blocks based on the concept of conventional generic multipliers, in particular, adders and basic AND gates. We show that this approach allows us to design generic multiplier structures which exhibit better delay in comparison with the classic multipliers, where adder represents a basic component.

1 Introduction

The design of combinational multipliers has been often concerned as a non-trivial task for demonstrating the capabilities of evolutionary systems. Gate-level representation has been usually utilized whose search space is typically rugged and hard to explore using the evolutionary algorithms. In case of applying a direct encoding (a non-developmental genotype–phenotype mapping) it is extremely difficult to achieve scalability of the evolved solutions, i.e. to obtain larger instances of the circuits, for example, when the traditional Cartesian Genetic Programming (CGP) is utilized [1]. Therefore, more effective representations have been investigated in order to overcome these issues and, in general, improve the scalability and evolvability of digital circuits, as summarized in the following paragraph.

Miller et al outlined the principles in the evolutionary design of digital circuits and showed some results of evolved combinational arithmetic circuits, including multipliers, in [2]. A detailed study of the fitness landscape in case of the evolutionary design of combinational circuits using Cartesian Genetic Programming

is proposed in [3]. 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 [4]. Vassilev and Miller studied the evolutionary design of 3×3 multipliers by means of evolved functional modules rather than only two-input gates [5]. 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 [6]. 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 [7]. Stomeo et al devised a decomposition strategy for evolvable hardware which allows to design large circuits [8]. 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 [9]. The potential capability of this method was demonstrated through experimental synthesis of arithmetic circuits at different levels of abstraction. 16×16 multipliers were evolved using word-level arithmetic components (such as one-bit full adders or one-bit registers).

An instruction-based developmental system was introduced in [10] for the design of arbitrarily large multipliers. Genetic algorithm was utilized to evolve a program for the construction of generic multipliers using a parametric approach. Basic AND gates and higher building blocks based on one-bit adder were utilized. A concept of *environment* (an external control of the developmental process) was introduced in order to design irregular structures. An interesting phenomenon of *adaptation* to different environments was observed. The results presented in [10] pose the first case when generic multipliers were evolved using the development.

In this paper an improved developmental model is presented that is based on the system introduced in [10]. Simplified building blocks are utilized in order to clarify the circuit structure and not to restrict the search space (only basic AND gates and pure half and full adders are considered). The adaptation is exploited (as discussed in [10]) for the design of different circuit structures depending on the environment chosen. In particular, the experiments are devoted to the design of carry-save multipliers which exhibit shorter delay in comparison with the classic multipliers as described in [11]. Note that the evolutionary development of classic generic multipliers was introduced in [10].

2 Biologically Inspired 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 ge-

netic 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 a 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 the 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 [12].

3 Development of Efficient Generic Multipliers

The method of the development is inspired by the construction of conventional combinational multipliers for which generic algorithms exist. Figure 1 shows two typical designs of a 4×4 multiplier constructed by means of the conventional approach [11]. It is evident that the circuits contain parts which differ from the rest of the circuit, i.e. they represent a kind of irregularity. In particular, it is a case of the first level ("row") of AND gates occuring in both multipliers in Fig. 1a,b and the last level of adders occuring in the carry-save multiplier (Fig. 1b). However, the rest of the circuit structure exhibits a high level of regularity that can be expressed by means of an iterative algorithm utilizing variables and parameters related to a given instance of the multiplier. For example, the number of bits of the operands determines the number of AND gates and adders in the appropriate circuit level or the number of levels of the multiplier. Theferore, this concept is assumed to be convenient for the design of generic multipliers using the development and an evolutionary algorithm.

Experiments were conducted under the previous research dealing with the evolutionary design of generic multipliers using an instruction-based developmental model [10]. The building blocks utilized in that method include an adder put together with a basic AND gate which, however, may pose an unsuitable approach preventing the evolution from finding better solutions. Nevertheless, general programs were evolved for the construction of multipliers whose structure corresponds to that of the classic combinational multiplier shown in Fig. 1a. The obtained results showed the possibility of designing generic multipliers using an evolutionary algorithm with the development which gave rise an interesting area deserving of future investigation. Therefore, new features of the developmental model will be introduced in this paper in order to design more efficient multipliers than that were evolved in [10]. The approach presented herein is based on the structure of the carry-save multiplier (see Fig. 1b) which exhibit shorter delay in comparison with the classic multiplier shown in Fig. 1a [11].

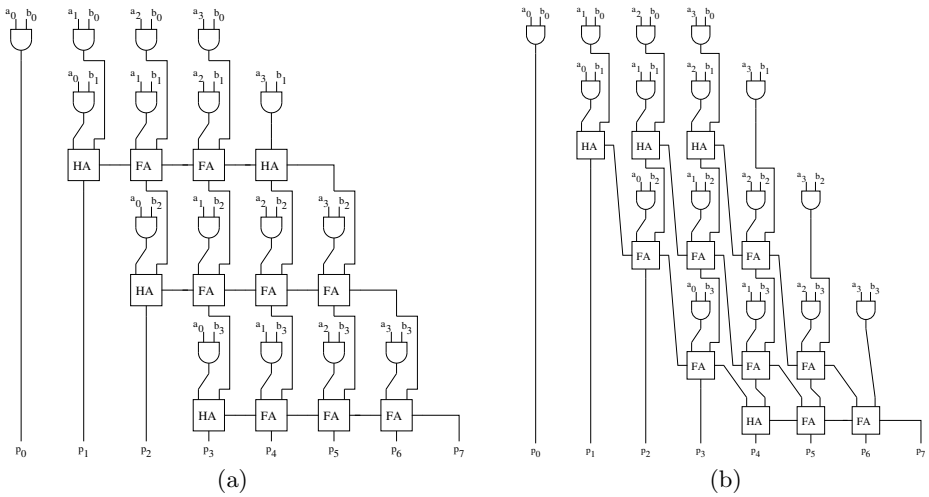


Fig. 1. 4×4 conventioanl multipliers: (a) classic combinational multiplier, (b) more efficient carry-save multiplier. $a_0, \ldots, a_3$, respective $b_0, \ldots, b_3$ denote the bits of the first, respective the second operand, $p_0, \ldots, p_7$ represent the bits of the product.

In particular, simplified building blocks will be introduced including only basic AND gates and pure one-bit adders and an enhanced instruction for creating the circuit structure will be presented that is able to generate two building blocks at a time — in addition to the single-block generative instruction introduced in [10] — due to the increased complexity of the construction algorithm needed for the design of generic multiplier structure using the simplified building blocks.

4 Instruction-Based Developmental System

A simple two-dimensional grid consisting of a given number of rows and columns was chosen as a suitable structure for the development of the target circuits. The building blocks are placed into this grid by means of a developmental program. In order to handle irregularities, an external control of the developmental process has been introduced that is called an environment.

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 logic 0.

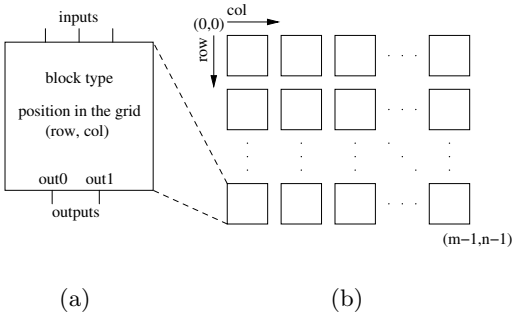


Fig. 2. (a) Structure of a building block. (row, col) determines the position of the block in the grid – see part (b). The connection of the inputs depends on the type and position of the block. (b) A grid of the building blocks with m rows and n columns for the development of generic multipliers.

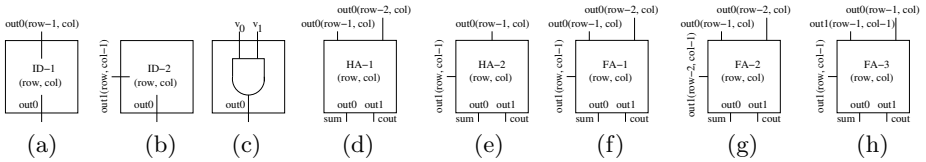


Fig. 3. Building blocks for the development of generic multipliers. (a, b) buffers – identity functions, (c) AND gate, (d, e) half adders, (f, g, h) full adders. (row, col) denotes the position in the grid. $\mathbf{v_0}$ and $\mathbf{v_1}$ determine indices of primary input bits. Connection of different inputs of the blocks are shown. Unused inputs and outputs are not depicted (set to logic 0). Note that the full adders (g, h) are new to this paper and are inspired by and intended for the design of carry-save 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 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 value of the 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 one or two building blocks of the type specified by its arguments. If (row, col) do not exceed the grid boundaries, the block is generated at that position. In case of generating two blocks, the second one is placed to $(row+1, col)$. If the grig boundary exceeds, no block is generated out of the grid. The possibility of generating two building blocks during an execution of the GEN instruction represents a new feature introduced in this paper in comparison with the system presented in [10], where only one block could be generated. This variant has been chosen in order to reduce the complexity of the developmental process when the simplified building blocks have been introduced. Note that this approach does nowise restrict the capabilities of the construction algorithm because the number and types of the blocks to be generated are determined independently by means of the evolutionary algorithm. In case of generating an AND gate, its inputs are connected to the primary inputs indexed by the actual values of variables v_0, v_1 as shown in Figure 3c. If v_0 or v_1 exceeds the bit width of the operands, the appropriate input of the AND gate is connected to logic 0. The inputs of the other building blocks are determined by the position (row, col) in the grid they are generated to. 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 *environment*. A single execution of a part of program is referred to as a *developmental step*. The meaning

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	$block1, block2$	Generate $block1$ to the actual position (row, col). If $block2$ is non-emty block, generate $block2$ to $(row + 1, col)$. Increase col by 1.
5: NOP		An empty operation.

Table 1. Instructions utilized for the development

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 corresponds to 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. 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. If a GEN instruction was executed, increase row by 2 in case of generating two building blocks simultaneously or by 1 if only single blocks were generated. Increase e by one and set col to 0.
4. If neither e, nor row exceed, go to step 2.
5. Evaluate the resulting circuit.

5 Evolutionary System Setup

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 choice of a part of the instruction (operation code or one of its arguments). If the operation code is to be mutated, entire new instruction will replace the original one, otherwise one argument is mutated. The mutation algorithm ensures proper values of arguments depending on the instruction type (see Table 1). The mutation is performed with the probability 0.03, only one instruction per chromosome is mutated.

A special crossover operator has been applied with probability 0.9, working as follows. 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.

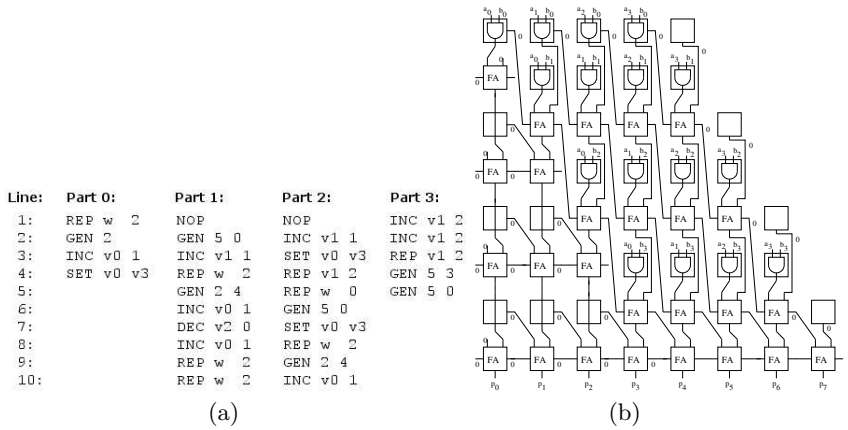
The fitness function is calculated as the number of bits processed correctly by the multiplier developed by means of the program stored in the chromosome. The experiments were conducted with the evolution of programs for the construction of 4×4 multipliers, i.e. the parameter $w = 4$. There are $2^{4+4} = 256$ possible test vectors and the multipliers produce 8-bit results. Therefore, the maximum fitness representing a working solution equals $256 \cdot 8 = 2048$. 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.

6 Experimental Results and Discussion

The experiments were devoted to the design of carry-save multipliers which exhibit better properties in comparison with the classic multipliers. In [10] an external information called an environment was introduced into the development for additional control of the developmental process. Moreover, an ability of adaptation of the design to different environments was observed allowing to create different multiplier structures. This feature was utilized in the experiments presented herein in order to investigate the ability of the evolutionary design system to construct carry-save multipliers. The selection of the evolved programs and circuits presented in this paper is based on their generality (i.e. the ability to construct generic multipliers) and the resemblance to the carry-save multiplier structure with respect to the circuit delay and the number of building blocks the developed multipliers are composed of.

In the first sort of experiments a subset of the bulding blocks from Fig. 3 was chosen for the design of the carry-save multipliers (see Fig. 1b). Therefore, only the blocks (a, b, c, d, g, h) were involved in the design process. Considering the irregular structure of the conventional carry-save multiplier, a program consisting of four parts is to be evolved. The parts of the program are executed according to the environment $env = (0, 1, 2, 2, 3)$, which is specified a priori with respect to the structure of carry-save multiplier. Therefore, the construction of the circuit is performed as follows. Considering Fig. 1b, the first level of the AND gates is created using part 0. The second level of AND gates together with the following level of adders are constructed by means of part 1. According to the environment, the next levels of AND gates and adders are created by means of double application of part 2. Finally, part 3 is utilized to design the last level of adders. Two hundreds of independent runs of the evolutionary algorithm were conducted from which 18% evolved a correct program for the construction of 4×4 multipliers. 60% of the evolved programs were classsified as general, i.e. able to create arbitrarily large multiplier. Figure 4 shows (a) one of the best evolved general program and (b) a 4×4 multiplier constructed by means of that program.

At the beginning of the development, the system is initialized: the variables are set to 0, the parameter is set to 4, row and column positions are initialized to 0 and the number of rows and columns is limited to 8 – no gate may be generated behind the grid boundaries. According to the first element of the environment (0), part 0 of the evolved program is executed. (see Fig. 4a) The first REP instruction initiates a loop repeating 4 times (for $w = 4$ – designing a 4×4 multiplier) two instructions after the REP instruction. In each pass, an AND gate (code 2 in the argument of GEN instructions at line 2) is generated with its inputs connected to the primary inputs of the circuit indexed by the values of variables v_0, v_1. Moreover, v_0 is increased by 1 (line 3) so that the AND gates generated in different passes possess the different first input. After executing a GEN instruction, the column position is increased by 1. After finishing part 0, the row position is increased by 1 and the column position is set to 0. According to the next element of the environment (1), part 1 will be executed. Note, however,



(a) (b)

Fig. 4. (a) An evolved general program, (b) 4×4 multiplier exhibiting the carry-save structure created by means of this program. Note that blank rectangles represent empty blocks (not generated by any instruction) whose outputs are considered as logic 0.

that the GEN instruction at line 2 of part 1 generates two building blocks into the actual column, the second block "under" the first one: full adder is generated into the second row of the first column (code 5 in the first GEN argument) and the identity function is generated into the third row of the first column (code 0 in the second argument). Since there have been building blocks generated into two rows, the row position is increased by 2 after finishing part 1. In case of executing part 3, only the full adders are generated (code 5 of the GEN instructions at lines 4 and 5) as there is no space left in the grid for the second level of blocks specified by the second argument of the GEN instructions – the number of rows of the grid was limited to 8 for this experiment.

It is evident that the multiplier shown in Fig. 4 could be optimized with respect to the inputs of some building blocks (e.g. adders possessing only one non-zero input could be replaced by the identity functions as demonstrated in [10]). After this optimization the circuit corresponds to the carry-save multiplier shown in Fig. 1b.

The second sort of experiments was devoted to the design of multipliers using the full set of building blocks shown in Fig. 3 and the same form of environment like in the previous experiment. Therefore, this setup corresponds to both variants of the multipliers from Fig 1. The prefix (0, 1, 2, 2) of the environment may be utilized for the evolution of classic multiplier structures shown in Fig. 1a. Again, 200 independent experiments were conducted from which 37% of working programs were obtained and 54% of them were classified as general. However, the experiments showed that the evolution of efficient carry-save multipliers is extremely difficult using this setup. Although there is all the resources available as in the first sort of experiments, no valid carry-save structure was obtained. The evolution generated the carry-save components very rarely and not to the positions at which they could be usefully utilized during the circuit operation.

The classic structures (Fig. 1a) were evolved instead. An example of a general program together with a 4×4 multiplier is shown in Fig. 5 which represents the same type of the classic multiplier structure that was evolved in [10].

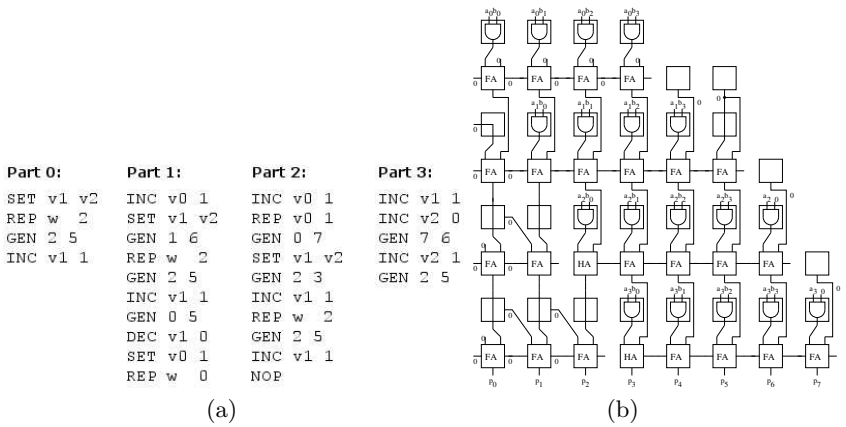


Fig. 5. (a) An evolved general program, (b) a 4×4 multiplier based on the structure of the classic combinational multiplier. Blank rectangles represent empty blocks with the outputs possessing logic 0.

The experiments presented in this section represent a continuation of the successful research in the field of the evolutionary design of generic multipliers using development. The phenomenon of adaptation of the developmental process to different environments during the evolution introduced in [10] enabled us to design various multiplier structures. In particular, the carry-save structure was rediscovered in this paper, exhibiting shorter delay in comparison with the classic multiplier, which was the main goal in the new sorts of our experiments. Although the carry-save multipliers showed to be very hard to evolve, the evolutionary developmental system demonstrated the ability to design this class of multipliers using a reduced set of building blocks. Moreover, simplified building blocks were introduced in this paper together with an improved developmental model in comparison with [10]. Therefore, there is a smaller limitation of the evolutionary process which, however, leads to more difficult evolution because of lower level of abstraction in the circuit representation. The success of the evolution of carry-save multipliers demonstrates an ability of the experimental system to design different circuit structures with more complex interconnection of their components which represents a promising area for the next research.

7 Conclusions

In view of the successful experiments, there is a big potential for the application of this model to other classes of well-scalable circuits, e.g. adders, median and

sorting networks etc. Therefore, the future research will be focused on adjusting the existing system to the specific circuit structures in order to investigate the evolution in the designs involving other building blocks and environments with respect to the construction of generic combinational circuits.

Acknowledgements. This work was supported by the Grant Agency of the Czech Republic under contract No. 102/07/0850 *Design and hardware implementation of a patent-invention machine*, 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. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proc. of the 3rd European Conference on Genetic Porgramming, Lecture Notes in Computer Science, vol 1802, Berlin Heidelberg New York, Springer (2000) 121–132
2. Miller, J.F., Job, D.: Principles in the evolutionary design of digital circuits – part I. Genetic Programming and Evolvable Machines **1**(1) (April 2000) 8–35
3. Miller, J.F., Job, D.: Principles in the evolutionary design of digital circuits – part i. Genetic Programming and Evolvable Machines **3**(2) (July 2000) 259–288
4. Vassilev, V., Job, D., Miller, J.: Towards the automatic design of more efficient digital circuits. In: Proc of the Second NASA/DoD Workshop on Evolvable Hardware, Palo Alto, CA, IEEE Computer Society (2000) 151–160
5. Vassilev, V., Miller, J.F.: Scalability problems of digital circuit evolution. In: Proc. of the 2nd NASA/DoD Workshop of Evolvable Hardware, Los Alamitos, CA, US, IEEE Computer Society (2000) 55–64
6. Murakawa, M., Yoshizawa, S., Kajitani, I., Furuya, T., Iwata, M., Higuchi, T.: 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, London, UK, Springer-Verlag (1996) 206–217
7. Torresen, J.: 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, Springer-Verlag (2003) 228–237
8. Stomeo, E., Kalganova, T., Lambert, C.: Generalized disjunction decomposition for evolvable hardware. IEEE Transactions on Systems, Man and Cybernetics – Part B **36** (2006) 1024–1043
9. Aoki, T., Homma, N., Higuchi, T.: Evolutionary synthesis of arithmetic circuit structures. Artificial Intelligence Review **20** (2003) 199–232
10. Bidlo, M.: Evolutionary development of generic multipliers: Initial results. In: Proc. of The 2nd NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2007, IEEE Computer Society (2007)
11. Wakerly, J.F.: Digital Design: Principles and Practice. Prentice Hall, New Jersey, US (2001)
12. S. Kumar (ed.), P. J. Bentley (ed.): On Growth, Form and Computers. Elsevier Academic Press (2003)