Layout Driven Optimization of Datapath Circuits using Arithmetic Reasoning

Ingmar Neumann* Dominik Stoffel* Kolja Sulimma* Michel Berkelaar+ Wolfgang Kunz*

*University of Kaiserslautern
Department of Electrical and Computer Engineering
Electronic Design Automation Group
D-67653 Kaiserslautern

+Magma Design Automation Inc.
2 Results Way
Cupertino, CA 95014, USA

Abstract

This paper proposes a new formalism for layout-driven optimization of datapaths. It is based on preserving an arithmetic bit level representation of the arithmetic circuit portions throughout various design stages. The arithmetic bit level description takes into account the arithmetic nature of the data path and facilitates arithmetic reasoning to identify circuit transformations that are too complex to derive for Boolean reasoning. It is a bit-level representation so that it integrates well into standard design flows. Based on this representation we developed an optimization algorithm for cycle time. It takes interconnect delay into account and can be applied at late design stages. A prototype has been integrated into a commercial EDA environment. For circuits implementing complex arithmetic expressions we achieved performance improvements of up to 32%.

1 Introduction

This paper describes an optimization method that is targeted towards the adder tree structures that are common in data path circuits. Data paths circuits often form a significant part of a modern integrated circuit (IC), and in many cases the critical timing path of these ICs passes through them. Traditional logic synthesis techniques [1, 5, 7, 8], which perform well on the control parts of the logic of the IC, are not well suited to optimize data paths. Logic synthesis techniques based on algebraic optimizations are too limited to use the full spectrum of optimizations that are potentially available in data paths due to the many existing symmetries. Sometimes they make the situation even worse, e.g., by destroying the regularity typical for arithmetic circuits [6]. Traditional techniques based on Boolean reasoning, like rewiring [10-13], could potentially exploit the symmetries In practice, these methods are severely limited by the inherent computational intractability of finding all of these symmetries in data path structures like multipliers. Finding all symmetries in a multiplier is computationally as hard as formally verifying the multiplier!

We propose a method that opens up the full use of optimization potential given by the symmetries in the data paths with the computational efficiency of algebraic methods. To find the symmetries we propose to use an arithmetic bit level representation of the circuit. This representation models both arithmetic information and bit-level circuit structure. It has already been applied successfully in the context of formal equivalence checking [9]. The arithmetic bit level representation exposes all symmetry information we need, and as it does not change during optimization, we have an extremely fast way to use this global Boolean property to optimize data paths. In this paper, we examine its use in layout-driven synthesis. We present an approach for optimizing an arithmetic circuit in terms of timing by swapping arithmetically symmetric operands. Our algorithm is as well suited for being applied during layout independent logic synthesis as well as during combined synthesis / layout generation.

In the past, there was little reason to apply logic synthesis to arithmetic circuits. For implementing arithmetic expressions, module generators are available. These tools either apply dedicated constructive algorithms [2], [3], [4] or simply use hard-coded knowledge from textbooks on how to implement a certain type of arithmetic operation. In the past, these approaches have worked sufficiently well, at least for synthesizing stand-alone arithmetic circuits, where all input and outputs have the same data arrival and required time, respectively. If an arithmetic expression is embedded into a large circuit the situation becomes more complicated. The module generator must take these external constraints into account, and they may change substantially during the design process.

In recent years, a second, even more crucial difficulty has come up for conventional module generators. The increasing contribution of interconnect delay to cycle time invalidates the simple timing models being used by most constructive algorithms. Without any layout data, predicting interconnect delay with any degree of accuracy is virtually impossible. This makes pre-layout timing optimizations a game of chance.

To overcome these problems, in recent years a significant amount of effort has been spent in developing approaches for integrating logic optimization into the physical design flow. Post-placement optimization followed by placement modifications [8], initial placement of an unmapped netlist followed by logic synthesis and final placement [5], integrating optimization operations into the inner loop of a placement algorithm [7] are among the most promising. However, most of these methods are based on traditional approaches for logic optimization and, hence, they achieve adequate results for control logic only.

The remainder of this paper is organized as follows. Section 2 explains the arithmetic bit level structure in detail. Section 3 shows how we can use the structure to find candidates for global pin swaps, and show that the optimization potential opened up by this can change any adder tree structure into any other. Section 4 shows an application of the pin swaps in an algorithm that performs timing optimization on a placed data path structure. Section 5 shows experimental results which highlight the tremendous optimization potential that is opened up: up to 32% of cycle time improvement at only 2% of area cost (all measured in the fully routed result).

2 Arithmetic Bit Level

A circuit structure which occurs frequently in data paths are addition networks consisting of cascaded addition trees. An addition tree computes the modulo-2 sum of n 1-bit operands. Furthermore, it computes $\lfloor n/2 \rfloor$ carry signals which are fed as input signals into the next addition tree. By cascading addition trees the number of carries produced in one addition tree is reduced until we finally get an $\lceil \log_2(n) \rceil$-bit wide operand consisting of $\lceil \log_2(n)\text{-}1 \rceil$ signals driven by sum outputs and one signal driven by a carry output. This operand represents the result of the addition.

Typically an addition tree is implemented by using $(n\text{-}1)/2$ full adders for odd n and $n/2\text{-}1$ full adders and one half adder for even n. Fig. 1 shows two addition trees consisting of three full adders and one full adder, respectively.

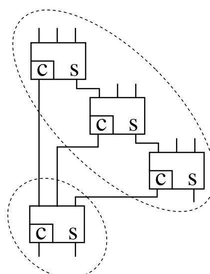

Fig. 1: Two addition trees

Addition trees are the main component in arithmetic circuits, not only in adders/subtractors but also in combinational multipliers as well as circuits for signal processing like digital filters. They have the remarkable property that all signals representing inputs to 1-bit adders belonging to the same addition tree are symmetric. They may be exchanged, therefore, without changing the result of the addition. This offers a large optimization potential. For example, by swapping two signals the circuit from Fig. 1 can be transformed into the circuit shown in Fig. 2. If we assume unit delay for each full adder, this will reduce the delay on the critical path from 4 to 3 in this example.

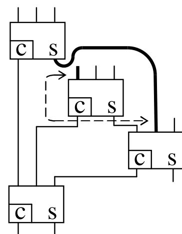

Fig. 2: Swapping two operands

Note that identifying such symmetries on the logic gate level by using conventional Boolean reasoning techniques is very expensive, often prohibitively so. However, if knowledge about the arithmetic nature of the circuit is preserved, only a simple analysis is required to identify all symmetries.

Therefore, we propose to represent all addition networks of a circuit as a netlist consisting of 1-bit-addition units (abbreviated by *AU* in the following). Each *AU* has either 2 (half adder) or 3 (full adder) inputs and calculates a sum and a carry output.

Definition 1 : An addition tree *AT* is a set of *n AUs* with the following properties:

1. The sum outputs of *n*-1 *AUs* are input signals of other *AUs* belonging to the same *AT*. The sum output of the *n*-th *AU* calculates the sum *S* of the *AT*.
2. For an *AT* with $n > 1$ the carry outputs of all *AUs* are inputs to another *AT*.

An addition tree consisting of *n AUs* generates one sum signal and *n* carry signals.

Obviously, two addition trees AT_i and AT_j may be merged into one single larger addition tree if the following conditions hold:

1. All carries of AT_i and all carries of AT_j are inputs to a third addition tree AT_k
2. S_i is an input of AT_j

Definition 2 : A *maximum AT* is an *AT* which cannot be enlarged by merging it with another *AT*.

In the following we will use the term *AT* as a synomym for maximum *AT*.

Definition 3 : An *addition network AN* is a set of *n ATs*, where *n*-1 $AT_i s$ have the property that all carry outputs generated by $AT_i \in AN$ are inputs to a subsequent $AT_j \in AN$, $i \neq j$.

An *arithmetic bit level description* of a circuit is a gate netlist where sub-circuits implementing addition networks are expressed exclusively in terms of *AU*-cells. All other parts of the circuit can be described using arbitrary gate types.

This description allows us to identify groups of arithmetically symmetric operands in linear time simply by traversing the addition networks.

Our approach is based on the assumption that the arithmetic bit level representation is preserved by the synthesis tool through all design stages. This can be done by specific annotations in the gate netlist that may have to be updated after certain design stages.

The arithmetic bit level representation allows us to perform arithmetic reasoning *beyond* the module generation phase, i.e. also during later design stages.

3 Operand swaps

A simple but very powerful method for timing optimization is swapping of symmetric operands. Operands are the inputs to the AU-cells.

This method has the attractive property that it does not insert new cells into the netlist. Consequently, it can be applied even on a fully placed netlist without affecting placement. Syntactically, operand swapping looks like the well-known pin swapping technique that is standard in many design flows. Note however, that the operand swapping considered here performs transformations that are much more global than conventional pin swapping.

In the following, we will show that certain pairs of operands in an *AN* are symmetric and can be swapped.

Lemma 1 : Swapping two operands being inputs to AU_i and AU_j, with AU_i, $AU_j \in AT_k$, will not change the sum S_k of AT_k provided that the operands do not lie in each other's fanin.

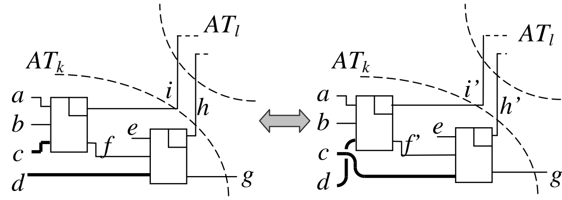

Fig. 3: Swapping operands *c* and *d*

Proof: Lets us consider Fig. 3 which shows the case that AU_i and AU_j are adjacent to each other. Adjacent in this context means that the sum of AU_i is an input of AU_j. S_k is represented by signal g and results to

$$g = (a \oplus b \oplus c) \oplus d \oplus e.$$

Now let us swap operand *c* and operand *d*. The new sum will be

$$g' = (a \oplus b \oplus d) \oplus c \oplus e.$$

Since the XOR function is commutative and associative it holds that $g' = g$. The proof is completed by noting that swapping two operands being inputs to non-adjacent *AUs* in an *AT* can always be implemented by a sequence of swaps for adjacent *AUs* provided that the swapped operands do not lie in the fanin cones of each other. ■

Lemma 2 : Swapping two operands being inputs to AU_i and AU_j, with AU_i, $AU_j \in AT_k$ will either swap the values of the carries generated by AU_i and AU_j or will not affect them provided that the operands do not lie in each other's fanin.

Proof: Consider Fig. 3 again. The carries generated by AU_i and AU_j correspond to the signals *i*, *h*. Lemma 2 can be formulated as a Boolean expression

$$F = (\neg (i \oplus i') \cdot \neg (h \oplus h'))+(\neg (i' \oplus h) \cdot \neg (h' \oplus i))$$

With

$$i = (a{\cdot}b + a{\cdot}c + b{\cdot}c)$$

$$h = e{\cdot}f + e{\cdot}d + d{\cdot}f, \; f = a \oplus b \oplus c$$

and

$$i' = (a{\cdot}b + a{\cdot}d + b{\cdot}d)$$

$$h' = e{\cdot}f' + e{\cdot}c + c{\cdot}f', \; f' = a \oplus b \oplus d$$

is possible to show by applying Boolean transformation that *F* is always true.

The proof is completed by noting that swapping two operands being inputs to non-adjacent *AUs* in an *AT* can always be implemented by a sequence of swaps for adjacent *AUs* provided that the swapped operands do not lie in each other's fanout cone.■

Lemma 3 : Swapping two operands being inputs to AU_i and AU_j, with AU_i, $AU_j \in AT_k$ will not affect the sum S_l of the subsequent addition tree AT_l which adds the carries generated by AT_k.

Proof: Lemma 2 tells us that swapping the operands will create a permutation of the carry signals produced by AT_k. Lemma 1 tells us that this will not affect S_l .■

Lemma 3 tells us that the sum signals generated by an addition network will not be affected by swapping internal operands. The carry signals created by one particular addition tree may be permutated. The carry of the last addition tree will not be affected since this tree generates only one single carry signal.

Next we will show that it is possible to transform an addition tree of arbitrary topology into any other valid addition tree topology by performing operand swaps.

Definition 4 : The degree d of an AU is given by the number of its input signals being driven by a sum output of another AU belonging to the same AT.

Definition 5 : A *chain* of AUs is an addition tree consisting of n AUs, $n > 0$, where n-1 AUs have a degree of one and one AU has a degree of zero.

Fig. 4 shows an AU chain. The numbers denote the degree of each AU.

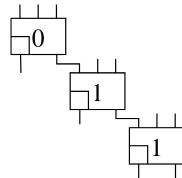

Fig. 4: AU chain

Lemma 4 : The degree of an AU with $d > 1$ can be decremented by one by swapping two operands.

Proof: Remember that an addition tree always contains at least one AU with $d = 0$ because of its acyclic structure. If there exits an AU_i with a $d_i > 1$, d_i can be decremented by one using the following procedure:

1. Find an AU_j with $d_j = 0$;
2. Swap an arbitrary input signal of AU_j and an arbitrary input signal of AU_i being driven by a sum output of an AU that is not located in the fanout cone of AU_j. (Note that such an input signal must exit; otherwise d_i would have been 0 or 1).■

This will result in $d_j = 1$ and $d_i = d_i$ - 1. The procedure is illustrated in Fig. 5.

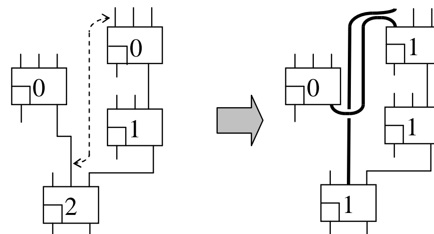

Fig. 5: Reducing the degree of an AU

Lemma 5 : Every addition tree can be transformed into a chain by performing operand swaps.

Proof: Repeated application of the procedure described in the proof of Lemma 4 will reduce the degrees of all AUs with $d > 1$ to 1. According to Definition 5 the result will be a chain. ■

Theorem 1: An addition tree of arbitrary topology can be transformed into any other topology by applying a sequence of operands swaps.

Proof: Follows from Lemma 5. If every topology can be transformed into a chain, then the reversal can be done also by applying the performed swaps in reverse order. Consequently, every topology may be transformed into any other topology by transforming it into a chain first. ■

Theorem 1 tells us that any embedded addition tree being built manually by a designer or by a constructive algorithm can be adapted optimally to changing timing constraints by only performing operand swaps during the design flow. E.g., an array multiplier may be transformed into a Wallace tree architecture or anything in-between just by swapping operands.

4 Timing Optimization

In Section 3 we examined what operands can be swapped in an *AN* without changing its function. Further we showed that is possible to transform an *AT* from one particular topology into any other topology. In this section we will show how to exploit this for timing optimization. At first we want to investigate how the length of the longest path leading through an input of an *AU* is affected by an operand swap.

Let us ignore interconnect delay for the time being. Consider two input pins *A* and *B* of two different *AUs* as shown in Fig. 6. For the inputs of each *AU* in a circuit we can calculate a data arrival time t_{ar} and a data time-to-sink t_{2s} performing a static timing analysis. The delay t_{pd} of the longest path leading through a particular input pin may be calculated simply by adding t_{ar} and t_{2s} of that pin.

Now, let us consider what happens if we swap the two operands shown in the example. Exchanging the input signals of two pins means that the arrival times for the pins will be swapped also. The t_{2s} values, however, are not affected by the swap. Consequently, if we wish to calculate the delays of the longest paths after the swap we simply have to exchange the t_{ar} - values in the corresponding equations.

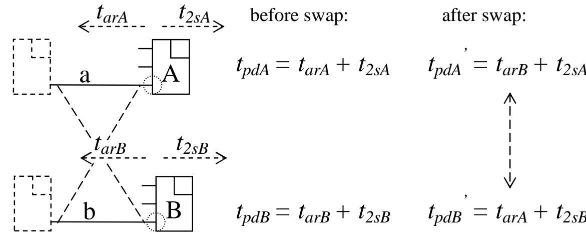

Fig. 6: Performance gain achieved by a swap (in absence of wire delay)

If we consider only the longest path trough *A* and the longest path through *B* the performance gain results to

$$gain = max(t_{pdA}, t_{pdB}) - max(t_{pdA}', t_{pdB}')$$

If one of the two pins affected by a swap lies on the most critical path of the circuit the swap allows us to shorten the critical path if it achieves a *gain* > 0. Note that the *gain* in cycle time for the whole circuit may be smaller than the gain of a swap. This can happen because some other path leading neither through pin *A* nor through pin *B* may become the most critical path in the circuit after the swap. However, it can be guaranteed that performing a swap with positive gain will at least not deteriorate cycle time.

A very important issue is that the gain of one particular operand swap can be calculated in constant time provided that t_{ar} and t_{2s} are known. Calculating t_{ar} and t_{2s} for all nodes in a circuit can be done in linear time applying a static timing analysis. Consequently, we are able to evaluate all possible operand swaps affecting the critical path very efficiently.

Now let us consider what happens if we take the influence of interconnect into account. In general, swapping operands will change the length of the corresponding wires, thereby affecting their delay. Consequently, the data arrival times at the inputs of the pins being swapped will change also. Note that for calculating these changes it is not necessary to perform a full timing analysis. Instead it is sufficient to determine how the delay of a wire being affected by a swap will change. Since

signals representing inputs to an AU typically have only a small number of pins, calculating the delay of a wire driving a particular pin can be done very efficiently in constant time (assuming a constant number of pins). If we denote the delay of a wire i driving a pin J by t_{w_iJ}, the path delays in the example shown above after the swap are as follows:

$$t_{pdA}' = t_{arB} + t_{2sA} - t_{w_bB} + t_{w_bA}$$

$$t_{pdB}' = t_{arA} + t_{2sB} - t_{w_aA} + t_{w_aB}$$

The wire delay factors t_w also include delays resulting from capacitive loads of the inputs of the AU.

Note that the gain calculation is not bound to a certain timing model. Delay can be calculated with the accuracy of the data being available at a particular design level. This allows us to take all technology dependent parameters influencing delay like transistor sizes, wire capacitances, etc., into consideration during optimization. This is a fundamental advantage over the simple timing models being used in constructive algorithms.

In the following we present a simple greedy algorithm for optimizing a circuit in terms of timing. It improves performance by swapping arithmetically symmetric operands repeatedly and stops when no swap with positive gain is found. Two signals are swap candidates for each other if they belong to the same AT and if they do not lie in the fanin cone of each other.

Fig. 7 shows the optimization algorithm *optimize*.

```
repeat {
   perform timing analysis;
   determine most critical path p_critical
   maxgain=0;
   Signal a,b;
   foreach Signal i ∈ p_critical being an input of an AU {
      foreach Signal j being a swap candidate for i {
         if (gain(i,j)>maxgain) {
            a=i;
            b=j;
            maxgain=gain(i,j);
         }
      }
   }
   if (maxgain>0) {
      swap signals a and b;
   }
} until (maxgain=0);
```

Fig. 7: Algorithm *optimize*

5 Experiments

As a platform for our experiments we used the design environment of Magma Design Automation Inc. This tool offers a Tcl-based command line interface granting access to most of its internal data. Therefore, it is well suited for integrating prototype algorithms. Our optimizer is implemented in a set of Tcl scripts. For all the other tasks including module generation, optimization, timing analysis and physical synthesis the corresponding Magma functions have been used.

Our new design flow consists of the following steps:

1. Module generation
2. Global placement
3. Optimize by operand swaps using the layout data gained from step 2.
4. Incremental placement with integrated optimization and global routing,
5. Detailed routing

The wire length values used for gain calculation in step 3 have been estimated using the half perimeter bounding box method. The conventional design flow we used for comparison performed the same steps except step 3.

For benchmarking we generated circuits for the following arithmetic expressions:

1. $y_{[16]} = a{\cdot}b + c{\cdot}d + e{\cdot}f + g{\cdot}h$
2. $y_{[32]} = a{\cdot}b + c{\cdot}d + e{\cdot}f + g{\cdot}h$
3. $y_{[16]} = (a+b) \cdot (c+d) + (e+f) \cdot (g+h)$
4. $y_{[32]} = (a+b) \cdot (c+d) + (e+f) \cdot (g+h)$
5. $y_{[32]} = a{\cdot}b$
6. a 16-Bit microprocessor with multiplication

All circuits have been mapped onto a 0.13µm standard cell library.

In our first experiments our goal was to minimize cycle time. The results are shown in Table 1. Column 1 contains the number of the expression. Column 2 and column 3 contain the improvements in cycle time and area that have been achieved compared to the results of the conventional flow. Both cycle time and area are measured at the end of the flow, which means after detailed routing. Hence, they are very accurate and realistic. The area can be different as steps 4 and 5 of the flow above still perform a lot of optimizations, such as restructuring, buffering and gate sizing.

Benchmark	Cycle time	Area
1	-31%	+2%
2	-32%	-10%
3	-2%	-9%
4	-3%	-15%
5	-20%	+5%
6	-15%	+4%

Table 1: Cycle time improvement

The results show improvements in cycle time of up to 32% while the area roughly stays the same, or even decreases. Note that the benchmarks with the largest improvements (1,2) are typical for multiply/accumulate-expressions that occur frequently in signal processing.

In the following experiments we tried to minimize area for a given cycle time. This was done by using the cycle time that could be achieved by the conventional flow as a target during the arithmetic level pin swaps. This optimization does not cost area, and the optimizations in steps 4 and 5 above do not need to add (as much) area in order to achieve the performance goal. The area improvements are shown in Table 2. The improvements in cell area have been nearly the same since the same cell row utilization has been used for both experiments.

Benchmark	Area reduction
1	-41%
2	-37%
3	-11%
4	-13%
5	-18%
6	-3

Table 2: Area improvement

For benchmark 6 the improvement is relatively small due to large non-arithmetic parts of the design.

Finally, we investigated the ability of our algorithm to adapt a circuit to non-uniform input data arrival times. For this purpose we generated a few sets of non-uniform data arrival times for circuit no. 2 randomly. The performance improvements now ranged up to 45% depending on how the data arrival times had been chosen.

The runtime for our optimizer ranges from a few minutes for small circuits to half an hour for the complex arithmetic expressions. However, it must be mentioned that most of the time is consumed by the Tcl-Interpreter. A more refined integration of the proposed approach into the Magma environment could further reduce CPU-times substantially.

6 Conclusion

In this paper we exploit the advantages of an arithmetic bit level representation when synthesizing datapaths. This level preserves information about arithmetic symmetries in addition networks that are very expensive to identify using Boolean techniques. We presented an optimization algorithm on the arithmetic bit level which optimizes addition networks in terms of timing. It is an attractive property of this algorithm that it can produce arbitrary topologies of the addition network while leaving the placement of a netlist untouched.

In our experiments we obtained performance improvements for complex arithmetic expressions of up to 32% for uniform data arrival times and of up to 45% for non-uniform data arrival times.

References

[1] "Logic Synthesis and Verification", ed. by Hassoun S. and Sasao T., Kluwer Academic Publishers, 2002 Boston/Dordrecht/London

[2] Oklobzija V., Villeger D., Ravi R., "A Method for Speed Optimizing Partial Product Reduction and Generation of Fast Parallel Multipliers using an Algorithmic Approach", IEEE Trans. on Comp. Vol. 5, No. 3, 1996

[3] Stelling P., Martel C., Oklobzija V., Ravi R., „Optimal Circuits for Parallel Multipliers, IEEE Trans. on Comp., Vol. 47, No. 3, 1998

[4] Um J., Kim T., "An Optimal Allocation of Carry-Save-Adders in Arithmetic Circuits", IEEE Trans on Comp., Vol. 50, No. 3, pp. 215-233, 2001

[5] Kutzschebauch T., Stok L., "Congestion Aware Layout Driven Logic Synthesis", Proc. ICCAD-2001, pp. 216-223, 2001

[6] Kutzschebauch T., Stok L, "Regularity Driven Logic Synthesis", Proc. ICCAD-2000, pp. 439-446, 2000

[7] Hartje H., Neumann I., Stoffel D., Kunz W.,Cycle Time Optimization by Timing Driven Placement with Simultaneous Netlist Transformations", Proc. ISCAS-2001, pp. 359-362, 2001

[8] Lou J., Chen W., Pedram M., “Concurrent logic restructuring and placement for timing closure”, Proc. ICCAD-1999, pp. 31-35, 1999

[9] Stoffel D., Kunz W., “Verification of Integer Multipliers on the Arithmetic Bit Level”, Proc. ICCAD-2001, pp. 183-189, 2001

[10] Kunz W., Menon P., “Multi-Level Logic Optimization by Implication Analysis”, ICCAD-94, pp. 6-13, 1994

[11] Chang C, Marek-Sadowska M., “Perturb and Simplify: Multi-Level Boolean Network Optimizer, ICCAD-94, pp. 2-5, 1995

[12] Sinha S., Brayton R., “Implementation and Use of SPFDs in Optimizing Boolean Networks”, ICCAD-98, pp. 103-110, 1998

[13] Chang C., Cheng C., Suaris P., Marek-Sadowska M., “Fast Post-placement Rewiring Using Easily Detectable Functional Symmetries”, Proc. DAC-2000, pp. 286-289, 2000