FACTA UNIVERSITATIS (NIŠ)
SER.: ELEC. ENERG. vol. 24, no. 3, December 2011, 437-449

Rule-Based Optimization of AND-XOR Expressions

Danil Knysh and Elena Dubrova

Abstract: The problem of finding a minimum AND-XOR expression for a given Boolean function is known to be very hard. In this paper we investigate whether a rule-based approach can help minimizing AND-XOR expressions for functions which are too large to be handled by algorithmic-based approaches. We apply a simple greedy search based on a set of local transformations to the positive polarity Reed-Muller expression of Boolean functions. Our experiments on large functions show surprisingly good results. We achieve 23% reduction in the number of literals on average. We believe that much better results can be achieved if a more sophisticated non-greedy search is used. The purpose of this paper is to motivate more research in this direction.

Keywords: Reed-Muller expression; AND-XOR expression; local transformation; greedy search.

1 Introduction

Logic optimization approaches can be divided into algorithmic-based (or global transformation) methods and rule-based (or local transformation) methods [1].

Rule-based methods use a set of rules which are applied when certain patterns are found. A rule transforms a pattern for a local sub-expression, or a sub-circuit, into another equivalent one. Since rules need to be described, and hence the type available of operations/gates must be known, the rule-based approach usually requires that the description of the logic is confined to a limited number of operation/gate types such as AND, OR, XOR, NOT etc. In addition, the transformations

Manuscript received July 3, 2011. An earlier version of this paper was presented at the Reed Muller 2011 Workshop, May 25-26, 2011, Gustavelund Conference Centre, Tuusula, Finland.

D. Knysh is with Taganrog Institute of Technology, CAD/FACF, 347 928 Taganrog, Russia (e-mail: wiseman33@ya.ru). E. Dubrova is with Royal Institute of Technology, ES/ICT/KTH, 164 46 Kista, Sweden (e-mail: dubrova@kth.se)

Digital Object Identifier: 10.2298/FUEE1103437K have limited optimization capability since they are local in nature. Examples of rule-based systems include LSS [2] and LORES/EX [3].

Algorithmic methods use global transformations such as decomposition or factorization, and therefore they are much more powerful compared to the rule-based methods. However, general Boolean methods, including don't care optimization, do not scale well for large functions. Algebraic methods are fast and robust, but they are not complete and thus often give lower quality results. For this reasons, industrial logic synthesis systems normally use algebraic restructuring methods in a combination with rule-based methods.

For the case of AND-XOR optimization, many two- and multi-level algorithmic approaches have been proposed (see Chapter 2 of [4] for an excellent overview).

Most two-level minimization algorithms focus on finding the best polarity for input variables, e.g. [5, 6, 7]. Very good results have been achieved in multi-level minimization, especially for arithmetic functions [8].

We presented an approach to boolean function AND-XOR minimization based on binary trees expression of Reed-Muller form. We applied simple local transformations to the positive polarity Reed-Muller expression of Boolean functions. We tested our algorithm on large functions and we got results about 23% minimization.

For some functions, the improvement is over 50%.

We also made experiments to different NLFSR minimization and we achieved 25% reduction of area and power after synthesis in RTLCompiler (Cadence).

The paper is organized as follows. Section 2 describes main notions and definitions used in the sequel. Section 3 presents the main idea of the proposed rule-based approach. Section 4 describes the algorithm. Section 5 summarizes experimental results. Section 6 concludes the paper and discusses open problems.

2 Preliminaries

In 1954 Reed [9] and Muller [10] observed that any Boolean function can be expressed as an expansion using AND and XOR operations. Their work leads to better implementations of some practical Boolean functions using AND-XOR arrays rather than AND-OR arrays [11].

An n-variable Boolean function has 2^n canonical *fixed-polarity Reed-Muller expressions* of type:

$$f(x_1, x_2, \dots, x_n) = c_0 \oplus c_1 \dot{x}_1 \cdot \ldots \cdot c_n \dot{x}_n \oplus c_{2^n-1} \dot{x}_1 \dot{x}_2 \dots \dot{x}_n,$$

where $\dot{x}_i$ is either x_i or a complement of x_i, according to the polarity vector, and

$c_j \in \{0,1\}$, $j \in \{0,1,\ldots,2^n-1\}$ are constants.

If all variables appear in the above expression uncomplemented, it is called *positive polarity Reed-Muller expression*. The positive polarity Reed-Muller expression is also known as *Algebraic Normal Form (ANF)* in cryptographic community, where it is used for representing feedback functions of Linear and Non-Linear Feedback Shift Registers (FSRs) [12, 13]. By minimizing the number of literals in a positive polarity Reed-Muller expression, we can reduce the area of the combinational logic implementing the feedback functions of an FSR. This is important for FSR-based stream ciphers such as Achterbahn [14], Grain [15], Dragon [16], Trivium [17], VEST [18], and [19]. At present, FSR-based stream ciphers are the most promising candidates for cryptographic primitives for advanced contactless technologies like RFID because they have the smallest hardware footprint of all existing cryptographic systems [20]. Since low-cost RFID tags which cannot afford more than a few hundreds of gates for security functionality [21], minimizing the area of a stream cipher is very important. See [22] for examples of positive polarity Reed-Muller expressions used in existing stream ciphers.

3 Intuitive Idea

Our goal is to further reduce the number of literals in a given positive polarity Reed-Muller expression of a Boolean function. We do it by subsequently applying three transformation rules: $x \cdot y \oplus x \cdot z = x \cdot (y \oplus z)$. $x \oplus 1 = \bar{x}$, $x \cdot 1 = x$. In our implementation of the algorithm, we represent the Reed-Muller expression by a binary tree with internal nodes labelled by AND and XOR operations, and rest nodes are labelled by variables or constants. For example, the circuit representing the expression $f(x_1,x_2,x_3) = 1 \oplus x_1x_2 \oplus x_3x_1x_2$ is shown in Figure 3.

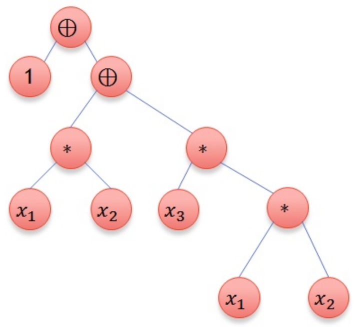
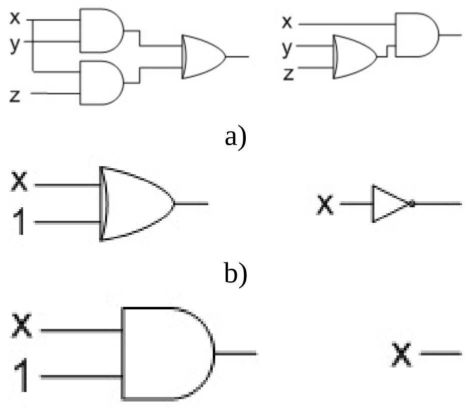


Fig. 1. Example binary tree representation of RM-expression

The first rule, $x \cdot y \oplus x \cdot z = x \cdot (y \oplus z)$, factors out a literal common for two product-terms.

If either y or z is a constant 1, then the rule reduces to $x \cdot y \oplus x = x \cdot (y \oplus 1)$. So, the transformation $x \cdot y \oplus x = x \cdot \bar{y}$ is done in two steps. Applying the first rule allows us reduce one gate (see Figure 2a). The second rule switches from fixed-polarity RM-form to mixed-polarity RM-form (see Figure 2b). And the third rule as the first one reduces one gate (Figure 2c).



c)

Fig. 2. Rules base: a) the first rule, b) the second rule, c) the third rule.

The search is carried out starting from the root of the binary tree depth-first. We look for patterns corresponding to the transformation rules and apply them on first-found basis. For example, if we have an expression $1 \oplus x_1x_2 \oplus x_2x_3$, we identify that the literal x_2 is common for the product-terms x_1x_2 and x_2x_3 and factor it out as $1 \oplus x_2(x_1 \oplus x_3)$. If we have an expression $1 \oplus x_1 \oplus x_1x_2$, we identify that the literal x_1 is common for the product-terms x_1 and x_1x_2 and factor it out as $1 \oplus x_1(x_2 \oplus 1)$. By applying the transformation $x_2 \oplus 1 = \overline{x_2}$ we then further reduce this expression to $1 \oplus x_1\overline{x_2}$.

4 Rule-Based Algorithm

The pseudo-code of the algorithm is shown in Algorithm 1. The input is a positive polarity Reed-Muller expression of a Boolean function.

The output is an AND-XOR expression of this Boolean function with a smaller number of literals.

Algorithm 1 Minimizes the number of literals in a given positive polarity Reed-Muller expression.

```
1: $C := C_0$; /*$C_0$ is the binary tree representing the Reed-Muller canonical form*/
2: $cost := |C|$; /*$|C|$ is the number of vertices in $C$*/
3: while $|C| < cost$ do
4:     for every vertex $v$ in $C$ do
5:         $flag(v) := true$;
6:         $matchlist(p) :=$ MATCH($v$); /*MATCH($v$) returns the best match for $v$ and -1 if there is no
           match*/
7:     end for
8:     for every vertex $v$ in $C$ do
9:         if $flag(v) = true$ then
10:            if $matchlist(v) \neq -1$ then
11:                $C :=$ APPLY($v, matchlist(v)$);
12:                MARKAFFECTED($C$); /*Sets $flag(v) = false$ for all vertices $v$ affected by the applied
                   local transformation*/
13:            end if
14:        end if
15:    end for
16: end while
17: Return $C$;
```

The input Reed-Muller expression is represented by a binary tree with internal nodes labelled by AND and XOR operations, and leaf nodes labeled by variables. The tree is traversed depth-first. The procedure MATCH(v) tries to match the subgraph rooted at vertex v to any of the patterns for which a local transformation is defined. We use the following three transformations:

1. $x \cdot y \oplus x \cdot z = x \cdot (y \oplus z)$,
2. $x \oplus 1 = \bar{x}$,
3. $x \cdot 1 = x$.

The match which reduces the number of literals the most is returned. Only transformations which reduce the number of literals are considered. In case of multiple matches with equal cost, the first match is returned. In case of no match, -1 is returned.

The first rule is main in our rule base, let us consider how to apply it on binary trees:

1. Start from the root of tree,
2. Look for XOR node recursively,
3. If found then looking for common subgraph recursively (Figure 3a),
4. If found copy subgraph in temp (variable tmp), and replace it by 1 in tree (Figure 3b),

5. Add new AND node and connect it with common subgraph and XOR node (Figure 3c).

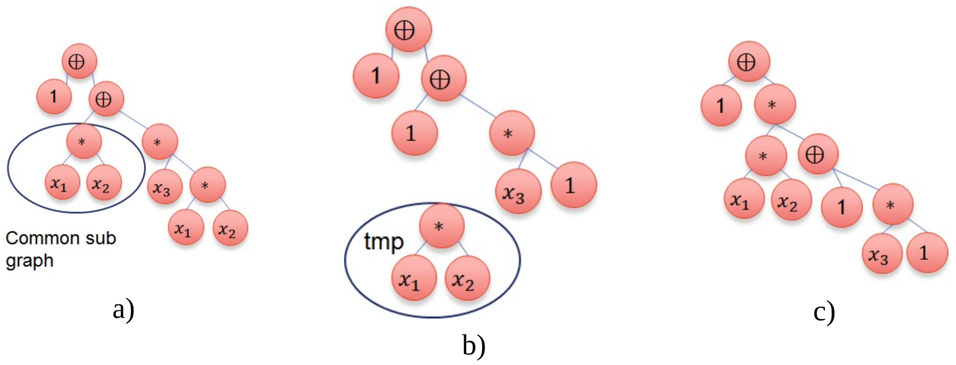


Fig. 3. An example of applying the first rule.

The optimization algorithm iteratively traverses the binary tree marking possible sub-graph transformations vertices and then, after the marking, replaces as many of them as possible. The procedure APPLY(v) applies a given transformation to a vertex v. After the tree has been locally transformed by APPLY, the flags assigned to vertices affected by the transformation are set to *false*. The procedure

Apply is only called for a vertex product, v, if its flag is *true*.

The algorithm stops iterating when no further minimization can be made. Full process of optimization for the function: $f(x_1,x_2,x_3) = 1 \oplus x_1x_2 \oplus x_3x_1x_2$ is shown in Figure 4.

5 Experimental Results

To evaluate the proposed approach, we applied it to more then 30 benchmark circuits with 15 and more inputs (we choosed it because it have a big number of inputs, it is important for NLFSR). The original benchmarks were in *espresso* (two-level AND-OR) format. We transformed them to the positive polarity Reed-Muller expression using our implementation of the Functional Decision Diagrams (FDD)-based algorithm [23] in CUDD package [24].

The results are summarized in Table 1. Columns 1, 2 and 3 show the name of the benchmark, the number of primary inputs and primary outputs, respectively. Columns 4 and 5 show the number of literals in the original positive polarity Reed-Muller expression and in the AND-XOR expression computed using the presented rule-based approach.

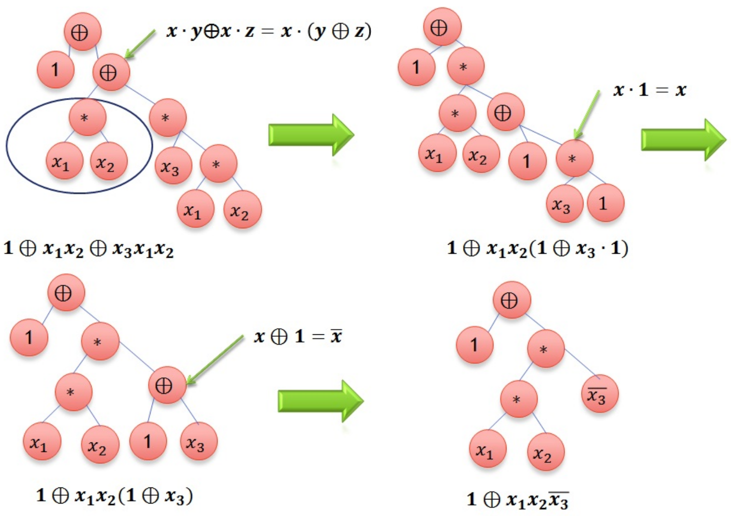

$x \cdot y \oplus x \cdot z = x \cdot (y \oplus z)$
$1 \oplus x_1x_2 \oplus x_3x_1x_2$
$x \cdot 1 = x$
$1 \oplus x_1x_2(1 \oplus x_3 \cdot 1)$
$x \oplus 1 = \overline{x}$
$1 \oplus x_1x_2(1 \oplus x_3)$
$1 \oplus x_1x_2\overline{x_3}$

Fig. 4. An example of full process of optimization.

Our current implementation supports single output functions only. For multiple-output functions, we calculate the number of literals separately for each output. The numbers shown in columns 4 and 5 are the sums of literals for all outputs, without taking sharing into account.

In the last column, we show the run time of the presented approach (a total sum of run times for each individual output).

The difference between number of cells in PPRM-expression and the result of our algorithm for this benchamrk is shown on Figure 5

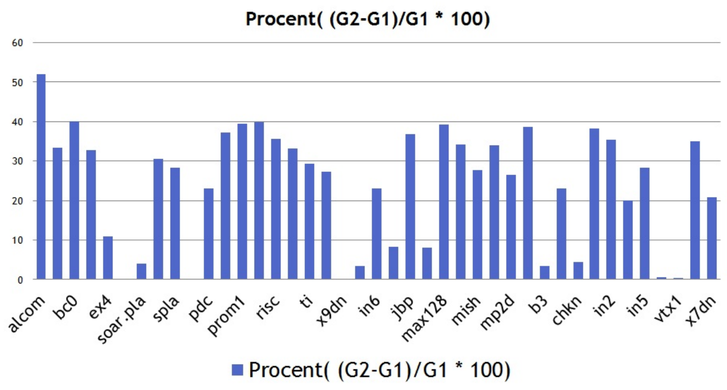


Fig. 5. Experimental results for some benchmark functions with 15 or more inputs

As we can see, on average, we achieve 23% reduction in the number of literals using 30.6 min for all outputs (which is 1 min per output on average).

Table 1. Experimental results for 30 benchmark functions with 15 or more inputs.

			Number of Literals		$\frac{L_1-L_2}{L_1}$	Time,
Name	Inputs	Outputs	L_1, Original PPRM	L_2, Optimized		sec
alcom	15	38	590	283	0.52	151
bc0	26	11	102807	61719	0.4	1209
cps	24	109	1254725	842617	0.33	8224
ex4	128	28	492009	438561	0.11	1057
soar	83	94	1851018	1775561	0.04	4381
spla	16	46	1516116	1087487	0.28	4754
pdc	16	40	1306776	1006523	0.23	4681
t1	21	23	4544	3040	0.33	906
ti	47	72	140494	99330	0.29	4084
ts10	22	16	5616	4080	0.27	2114
x9dn	27	7	21775743	21730592	0.002	1084
in4	32	20	3069036	2965936	0.03	2417
in6	33	23	49866	38399	0.23	758
in7	26	10	1269250	1164289	0.08	529
jbp	36	57	84808	53610	0.37	607
mark1	20	31	7315436	6726301	0.08	4091
misg	56	23	823	542	0.34	302
mish	94	43	245	177	0.28	151
misj	35	14	115	76	0.34	2
opa	17	69	55489	34037	0.39	2114
b3	32	20	2753233	2660698	0.03	2267
b4	33	23	49831	38389	0.23	758
chkn	29	7	769518	735422	0.04	917
ibm	48	17	20128	12422	0.38	604
in2	19	10	116031	74972	0.35	1060
in3	35	29	612307	489596	0.2	1371
in5	24	14	32676	23403	0.28	1057
vg2	25	8	6996204	6948513	0.01	482
x6dn	39	5	124964	81280	0.35	755
x7dn	66	15	442103	350452	0.21	2265
average	37.47	30.73	1740416.7	1648276.9	0.23	1838.4

Also we evaluated our Algorithm on random NLFSRs as well as on NLFSRs of existing stream ciphers (just boolean functions without flip-flops). We compared numbers of cells and area of scheme of NLFSR. All characteristics are achieved by RTLCompiler. We tested 40 different NLFSRs.

The results of comapring between RM-exprerssion of NLFSR and expression in our algoriyhm before syntesis in RTLComlier are showed in Table 2. We also compared area and number of cells in PPRM-expression and our algorithm expression after synthesis in RTLCompiler. The results showed in Table 3. This experiment showes ability of our algorithm to improve sinthesis work of RTLComplier about 10%. The results of our algorithm work dependen on number of common literals in RM-from of Boolean function.

Let us look at one of the most popular NLFSR - grain128 and compare its expression in RM-form and our algorithm form:

$x_{62} \oplus x_{60} \oplus x_{52} \oplus x_{45} \oplus x_{37} \oplus x_{33} \oplus x_{28} \oplus x_{21} \oplus x_{14} \oplus x_9 \oplus x_0 \oplus x_{63}x_{60} \oplus x_{37}x_{33} \oplus x_{15}x_9 \oplus x_{60}x_{52}x_{45} \oplus x_{33}x_{28}x_{21} \oplus x_{63}x_{45}x_{28}x_9 \oplus x_{60}x_{52}x_{37}x_{33} \oplus x_{63}x_{60}x_{21}x_{15} \oplus x_{63}x_{60}x_{52}x_{45}x_{37} \oplus x_{33}x_{28}x_{21}x_{15}x_9 \oplus x_{52}x_{45}x_{37}x_{33}x_{28}x_{21}$ - 50 nodes

After perfoming the algorithm we obtained

$x_{62} \oplus x_{60} \oplus x_{52} \oplus x_{45} \oplus x_{37} \oplus x_{33} \oplus x_{28} \oplus x_{21} \oplus x_{14} \oplus x_9 \oplus x_0 \oplus x_{63}x_{60} \oplus x_{37}x_{33} \oplus x_{15}x_9 \oplus x_{60}x_{52}x_{45} \oplus x_{28}(x_{33}x_{21} \oplus x_{63}x_{45}x_9) \oplus x_{60}(x_{52}x_{37}x_{33} \oplus x_{63}(x_{21}x_{15} \oplus x_{52}x_{45}x_{37})) \oplus x_{33}x_{28}x_{21}(x_{15}x_9 \oplus x_{52}x_{45}x_{37})$ 43 nodes

So we reduced 7 nodes.

6 Conclusion

This presents a greedy local transformation-based method for optimizing AND-XOR expressions using literal count as objective. Our experimental results show that we achieve 23% reduction in the number of literals on average compared to the positive polarity Reed-Muller expression.

The main drawback of the discussed technique is that the optimization algorithm is greedy.

A greedy algorithm might get stuck quite fast in a local minimum instead of exploring all possibilities.

Further work remains to explore more sophisticated search approaches.

Another drawback of the presented algorithm is a simplistic optimization objective.

The use of literals as an objective function often results in poor circuit structures since the number of literals is only loosely correlated with the delay of the mapped circuit.

In our future work we will consider more complex objective functions, e.g. providing the choice of minimizing the number of literals or number of levels.

Table 2. Experimental results for NLFSRs.

		NLFSR in RM-form		NLFSR our Algorithm		difference	difference
		Cells	Area	Cells	Area	Number of Cells	Area
1	2	3	4	5	6	7	8
1	VEST-1	31	194	19	110	38,71	43,30
2	VEST-2	33	207	23	138	30,30	33,33
3	VEST-3	30	188	22	128	26,67	31,91
4	VEST-4	30	188	21	125	30	33,51
5	VEST-5	32	207	23	135	28,13	34,78
6	VEST-6	35	219	22	128	37,14	41,55
7	VEST-7	32	200	24	144	25	28
8	VEST-8	26	169	21	119	19,23	29,59
9	VEST-9	27	169	20	113	25,93	33,14
10	VEST-10	33	207	25	147	24,24	28,99
11	VEST-11	31	194	23	135	25,81	30,41
12	VEST-12	30	188	23	135	23,33	28,19
13	VEST-13	35	219	26	150	25,71	31,51
14	VEST-14	32	207	23	132	28,13	36,23
15	VEST-15	32	200	25	147	21,875	26,5
16	VEST-16	34	213	24	144	29,41	32,39
17	VEST-17	32	200	22	132	31,25	34
18	VEST-18	28	182	20	119	28,57	34,62
19	VEST-19	27	169	20	116	25,93	31,36
20	VEST-20	34	213	22	132	35,29	38,03
21	VEST-21	33	207	23	138	30,30	33,33
22	VEST-22	36	225	22	135	38,89	40
23	VEST-23	32	200	22	128	31,25	36
24	VEST-24	35	219	26	153	25,71	30,14
25	VEST-25	32	207	22	132	31,25	36,23
26	VEST-26	32	200	22	132	31,25	34
27	VEST-27	34	213	26	153	23,53	28,17
28	VEST-28	32	200	23	138	28,13	31
29	VEST-29	32	200	23	132	28,13	34
30	VEST-30	28	182	24	135	14,29	25,82
31	VEST-31	25	157	20	122	20	22,29
32	VEST-32	31	194	23	132	25,81	31,96
33	Achterbahn-1	51	326	44	282	13,73	13,50
34	Achterbahn-2	37	250	34	225	8,11	10
35	Achterbahn-3	37	244	34	225	8,11	7,79
36	Achterbahn-4	67	426	56	351	16,42	17,61
37	Achterbahn-5	66	413	53	332	19,70	19,61
38	Achterbahn-6	67	432	54	338	19,40	21,76
39	grain128	18	133	18	113	0	15,04
40	grain80	39	263	38	244	2,56	7,22
average						24,43	28,92

Acknowledgment

The first author would like to thank Swedish Institute for the funding for his 1-year research visit to the Royal Institute of Technology.

Table 3. Experimental results for NLFSRs after synthesis in RTLCompiler.

		NLFSR in RM-form		NLFSR our Algorithm		difference	difference
		Cells	Area	Cells	Area	Number of Cells	Area
1	2	3	4	5	6	7	8
1	VEST-1	15	60	12	44	20	26,67
2	VEST-2	20	64	14	51	30	20,31
3	VEST-3	11	51	11	43	0	15,69
4	VEST-4	24	101	11	48	54,17	52,48
5	VEST-5	27	113	17	67	37,03	40,71
6	VEST-6	17	62	13	60	23,53	3,23
7	VEST-7	21	75	15	48	28,57	36
8	VEST-8	12	50	13	51	-8,33	-2
9	VEST-9	14	53	10	40	28,57	24,53
10	VEST-10	20	70	14	52	30	25,71
11	VEST-11	21	63	13	49	38,10	22,22
12	VEST-12	18	61	14	46	22,22	24,59
13	VEST-13	27	86	17	65	37,04	24,42
14	VEST-14	20	74	14	50	30	32,43
15	VEST-15	16	55	19	69	-18,75	-25,45
16	VEST-16	21	70	15	56	28,57	20
17	VEST-17	16	66	15	60	6,25	9,09
18	VEST-18	17	62	16	58	5,88	6,45
19	VEST-19	16	56	12	44	25	21,43
20	VEST-20	16	57	17	58	-6,25	-1,75
21	VEST-21	19	81	17	57	10,53	29,63
22	VEST-22	21	69	16	60	23,81	13,04
23	VEST-23	22	93	18	65	18,18	30,12
24	VEST-24	22	73	12	48	45,45	34,25
25	VEST-25	22	77	16	65	27,27	15,58
26	VEST-26	18	62	12	51	33,33	17,74
27	VEST-27	14	48	14	48	0	0
28	VEST-28	15	54	15	57	0	-5,56
29	VEST-29	20	77	15	58	25	24,68
30	VEST-30	21	78	10	44	52,38	43,59
31	VEST-31	15	60	10	44	33,33	26,67
32	VEST-32	13	48	9	45	30,77	6,25
33	Achterbahn-1	22	85	20	69	9,09	18,82
34	Achterbahn-2	27	142	27	135	0	4,93
35	Achterbahn-3	27	99	30	127	-11,11	-28,28
36	Achterbahn-4	51	199	44	185	13,73	7,04
37	Achterbahn-5	19	86	22	93	-15,79	-8,14
38	Achterbahn-6	19	89	20	95	-5,26	-6,74
39	grain128	15	80	15	80	0	0
40	grain80	27	165	30	154	-11,11	6,67
average						17,28	15,18

References

[1] R. K. Brayton, C. McMullen, G. Hatchel, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms For VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[2] J. A. Darringer, W. H. Joyner, C. L. Berman, and L. Trevillyan, “Logic synthesis through local transformations,” *IBM J. Res. Dev.*, vol. 25, pp. 272–280, July 1981.

[3] J. Ishikawa, H. Sato, M. Hiramine, K. Ishida, S. Oguri, Y. Kazuma, and S. Murai, “A rule based logic reorganization system lores/ex,” in *Computer Design: VLSI in Computers and Processors, 1988. ICCD ’88., Proceedings of the 1988 IEEE International Conference on*, Oct. 1988, pp. 262 –266.

[4] T. Sasao and M. Fujita, *Representations of discrete functions*. Kluwer Academic Publishers, 1996.

[5] H. Wu, M. Perkowski, X. Zeng, and N. Zhuang, “Generalized partially-mixed-polarity reed-muller expansion and its fast computation,” *Computers, IEEE Transactions on*, vol. 45, no. 9, pp. 1084 –1088, Sept. 1996.

[6] R. Drechsler, M. Theobald, and B. Becker, “Fast ofdd-based minimization of fixed polarity reed-muller expressions,” *Computers, IEEE Transactions on*, vol. 45, no. 11, pp. 1294 –1299, Nov. 1996.

[7] G. W. Dueck, D. Maslov, J. T. Butler, and a. S. N. Y. V. P. Shmerko, “A method to find the best mixed polarity reed-muller expression using transeunt triangle,” in *5th Int. Reed-Muller Workshop (RM’2001)*, May 2001, pp. 82–92.

[8] J. Saul, “Logic synthesis for arithmetic circuits using the reed-muller representation,” in *Design Automation, 1992. Proceedings., [3rd] European Conference on*, Mar. 1992, pp. 109 –113.

[9] I. S. Reed, “A class of multiple-error-correcting codes and the decoding scheme,” *Transactions of the IRE Professional Group on Information Theory*, vol. 4, pp. 38–49, 1954.

[10] D. E. Muller, “Application of Boolean algebra to switching circuit design and to error detection,” *IRE Transactions on Electronic Computers*, vol. 3, pp. 6–12, 1954.

[11] T. Sasao and P. Besslich, “On the complexity of mod-2l sum pla’s,” *Computers, IEEE Transactions on*, vol. 39, no. 2, pp. 262 –266, Feb. 1990.

[12] S. Golomb, *Shift Register Sequences*. Aegean Park Press, 1982.

[13] E. Dubrova, “A transformation from the Fibonacci to the Galois NLFSRs,” *IEEE Transactions on Information Theory*, vol. 55, no. 11, pp. 5263–5271, November 2009.

[14] B. Gammel, R. Göttfert, and O. Kniffler, “Achterbahn-128/80: Design and analysis,” in *SASC’2007: Workshop Record of The State of the Art of Stream Ciphers*, 2007, pp. 152–165.

[15] M. Hell, T. Johansson, and W. Meier, “Grain - a stream cipher for constrained environments.” citeseer.ist.psu.edu/732342.html.

[16] K. Chen, M. Henricken, W. Millan, J. Fuller, L. Simpson, E. Dawson, H. Lee, and S. Moon, “Dragon: A fast word based stream cipher,” in *eSTREM, ECRYPT Stream Cipher Project*, 2005, report 2005/006.

[17] C. D. Canniere and B. Preneel, “TRIVIUM specifications.” citeseer.ist.psu.edu/734144.html.

[18] B. Gittins, H. A. Landman, S. O’Neil, and R. Kelson, “A presentation on VEST hardware performance, chip area measurements, power consumption estimates and benchmarking in relation to the aes, sha-256 and sha-512,” Cryptology ePrint Archive, Report 2005/415, 2005, http://eprint.iacr.org/.

[19] B. M. Gammel, R. Göttfert, and O. Kniffler, “An NLFSR-based stream cipher,” in *ISCAS*, 2006.

[20] M. Robshaw, “The estream project,” *New Stream Cipher Designs: The eSTREAM Finalists, LNCS 4986*, pp. 1–6, 2008.

[21] A. Juels, “RFID security and privacy: a research survey,” *Selected Areas in Communications, IEEE Journal on*, vol. 24, no. 2, pp. 381–394, Feb. 2006.

[22] E. Dubrova, “F2G transforaation of NLFSRs,” 2011, http://web.it.kth.se/ dubrova/fib2gal.html.

[23] U. Kebschull and W. Rosenstiel, “Efficient graph-based computation and manipulation of functional decision diagrams,” in *Design Automation, 1993, with the European Event in ASIC Design. Proceedings. [4th] European Conference on*, Feb. 1993, pp. 278 –282.

[24] F. Somenzi, *CUDD: CU Decision Diagram Package, Release 2.3.1*. University of Colorado at Boulder, 2001.