Cirbo: A New Tool for Boolean Circuit Analysis and Synthesis

Daniil Averkov[1], Tatiana Belova[2, 3], Gregory Emdin[4], Mikhail Goncharov[5, 6], Viktoriia Krivogornitsyna[2], Alexander S. Kulikov[6], Fedor Kurmazov[2], Daniil Levtsov[5], Georgie Levtsov[5], Vsevolod Vaskin[5], Aleksey Vorobiev[3]

[1]St. Petersburg State University
[2]Steklov Mathematical Institute at St. Petersburg, Russian Academy of Sciences
[3]ITMO University
[4]École Polytechnique Fédérale de Lausanne
[5]Neapolis University Pafos
[6]JetBrains Research

Abstract

We present an open-source tool for manipulating Boolean circuits. It implements efficient algorithms, both existing and novel, for a rich variety of frequently used circuit tasks such as satisfiability, synthesis, and minimization. We tested the tool on a wide range of practically relevant circuits (computing, in particular, symmetric and arithmetic functions) that have been optimized intensively by the community for the last three years. The tool helped us to win the IWLS 2024 Programming Contest. In 2023, it was Google DeepMind who took the first place in the competition. We were able to reduce the size of the best circuits from 2023 by 12% on average, whereas for some individual circuits, our size reduction was as large as 83%.

Code — https://github.com/SPbSAT/cirbo

Introduction

Boolean circuits is a mathematical model with applications in various branches of Computer Science such as Complexity Theory, Computer Engineering, and Cryptography. The two most important related computational problems are circuit analysis and circuit synthesis.

Circuit analysis: given a circuit, check whether it possesses a certain property. A ubiquitous property is satisfiability (whether it is possible to assign 0/1 values to the inputs such that the circuit evaluates to 1). The corresponding problem is known as Circuit SAT. On the one hand, many circuit analysis problems (such as logical equivalence checking and verification) are equivalent to Circuit SAT. On the other hand, Circuit SAT is a generalization of SAT (satisfiability of formulas in conjunctive normal form) and many hard combinatorial problems are reduced to SAT via Circuit SAT. At the same time, Circuit SAT can be reduced to SAT in a natural way. This way, Circuit SAT shares a wide range of applications, both practical and theoretical, with SAT (Biere et al. 2021).

Circuit synthesis: given a specification of a Boolean function, synthesize a small circuit computing this function. This is the first step of integrated circuit design, a highly important problem in practice. In Theoretical Computer Science, this problem is known as the Minimum Circuit Size Problem (MCSP): given a truth table of a Boolean function and an integer parameter r, check whether the function can be computed by a circuit of size r. MCSP is arguably as important as SAT (Santhanam 2022).

Proving that a given Boolean function cannot be computed by a small circuit (that is, proving circuit lower bounds) is notoriously hard. The number of functions and the number of circuits (as functions of the number of inputs) grow too fast making it infeasible in practice to show that a given function on, say, 10 inputs cannot be computed by a circuit with 40 gates. Moreover, currently, there are no methods that allow one to exclude a possibility that every problem from NP has circuit size at most $4n$, where, as usual, n denotes the input size (Find et al. 2016; Li and Yang 2022; Find et al. 2023).

New Tool

The focus of the this paper is on practical aspects of the two problems mentioned above: we present a new tool, called `Cirbo`, for solving a wide range of problems on Boolean circuits. The tool implements a variety of algorithms, both known and novel ones. The tool allowed us to win the IWLS 2024 Programming Contest.[1] The goal of the competition is to synthesize efficient circuits for 100 Boolean functions (specified by their truth tables), in two bases, XAIG and AIG. For each of the two bases, for more than half of the functions, the circuits synthesized by our tool turned out to be the smallest among the circuits produced by all teams. Moreover, the datasets in 2024 contest were the same as in 2023, giving us a possibility to track the progress of reducing the circuit size for these datasets. Table 1 shows the corresponding circuit size for a selection of datasets and highlights that in some cases our size reduction was as large as 83%. Later in the text, we define all the functions from the table formally, provide more statistics as well as detailed steps that led to improved circuits.

[1]https://www.iwls.org/iwls2024/

		AIG			XAIG		
function	IWLS 2024 code	2023 (best)	2024 (our)	improvement	2023 (best)	2024 (our)	improvement
modulo8	ex33	1182	250	78.85%	1158	190	83.59%
square12	ex96	1319	476	63.91%	1284	324	74.77%
div8	ex88	317	176	44.48%	306	142	53.59%
sqrt16	ex65	239	176	26.36%	226	136	39.82%
sort15	ex39	114	96	15.79%	114	73	35.96%
maj15	ex36	68	68	0%	68	48	29.41%
neuron	ex20	653	588	9.95%	644	587	8.85%
espresso	ex25	1381	1340	2.97%	1381	1339	3.04%

Table 1: Circuit size for a selection of benchmarks from IWLS 2024 Programming Contest from various categories. For each benchmark, we show, in the first two columns, its short description as well as the name of the corresponding benchmark. The next two columns show the smallest circuit size in the AIG basis found in 2023 as well as the size of a circuit synthesized by our tool in 2024. The next column shows size reduction in percent (compared to the best circuit size in 2023). Finally, the last three columns show the same data for the XAIG basis.

Related Work

There is a number of packages providing a similar functionality: `ABC` (Brayton and Mishchenko 2010) and `mockturtle` (Soeken et al. 2018) are general purpose tools for working with Boolean circuits implemented in `C++`, whereas `CLI` (Kulikov, Pechenev, and Slezkin 2022) and `CIOPS` (Reichl, Slivovsky, and Szeider 2023) are circuit minimization tools based on SAT/QBF solvers implemented in `Python`. As our experiments show, our tool is capable of solving various datasets better than the tools mentioned above. At the same time, for some of these datasets, the best results have been achieved by combining our tool with the existing ones.

General Setting

For a predicate P, $[P]$ is the Iverson bracket: $[P] = 1$ if P is true and $[P] = 0$ otherwise. For a non-negative integer q, $\mathrm{bin}(q)$ is the binary representation of q (padded with a number of leading zeroes if needed). Conversely, for a bit-string $b = (b_0, \ldots, b_k)$, $\mathrm{int}(b) = \sum_{i=0}^{k} 2^i b_i$ is the corresponding integer.

Boolean Functions

Let $B_{n,m} = \{f\colon \{0,1\}^n \to \{0,1\}^m\}$ be the set of all Boolean functions with n inputs and m outputs and let $B_n = B_{n,1}$ be the set of all n-input single-output functions (that is, predicates). A function of the form $f\colon \{0,1\}^n \to \{0,1,*\}^m$ is called *partially defined*: $*$ is known as *don't care* symbol and means an undefined Boolean value.

Below, we define a number of specific Boolean functions studied in this paper. By $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ we denote input n-bit strings and $\mathrm{sum}(x) = x_1 + \cdots + x_n$.

- $\mathrm{MAJ}_n \in B_n$ is the majority function, that is, it is equal to 1 if and only if more than half of the n input bits are 1's: $\mathrm{MAJ}_n(x) = [\mathrm{sum}(x) > n/2]$.
- $\mathrm{SUM}_n \in B_{n,\lceil \log_2(n+1)\rceil}$ computes the binary representation of the sum of n input bits: $\mathrm{SUM}_n(x) = \mathrm{bin}(\mathrm{sum}(x))$.
- $\mathrm{SORT}_n \in B_{n,n}$ sorts the given n bits: $\mathrm{SORT}_n(x) = (x'_1, \ldots, x'_n)$, where $x'_1 \le \cdots \le x'_n$ and $\mathrm{sum}(x) = \mathrm{sum}(x')$.
- $\mathrm{MULT}_n \in B_{2n,2n}$ computes the product of the given two n-bit integers: $\mathrm{MULT}_n(x, y) = \mathrm{bin}(\mathrm{int}(x) \cdot \mathrm{int}(y))$.
- $\mathrm{SQR}_n \in B_{n,2n}$ computes the square of the given n-bit integer: $\mathrm{SQR}_n(x) = \mathrm{MULT}_n(x, x)$.
- $\mathrm{SQRT}_n \in B_{n,n/2}$ computes the square root of the given n-bit integer: $\mathrm{SQRT}_n(x) = \mathrm{bin}(\lfloor \sqrt{\mathrm{int}(x)} \rfloor)$.
- $\mathrm{DIV}_n \in B_{2n,n}$ and $\mathrm{MOD}_n \in B_{2n,n}$ functions compute, respectively, the quotient and the remainder of the first input integer divided by the second input integer:

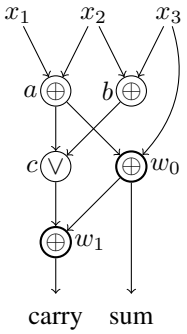
$$\mathrm{DIV}_n(x, y) = \mathrm{bin}(\lfloor \mathrm{int}(x) / \mathrm{int}(\mathrm{y}) \rfloor),$$
$$\mathrm{MOD}_n(x, y) = \mathrm{bin}(\mathrm{int}(x) \bmod \mathrm{int}(\mathrm{y})).$$

Boolean Circuits

A circuit is a natural way of computing Boolean functions. It is an acyclic directed graph of in-degree at most 2 whose n source nodes are labeled with input variables $x_1, \ldots, x_n$ and all other nodes (called *internal*) are labeled with Boolean operations from $B_1 \cup B_2$ (that is, unary and binary Boolean predicates). The nodes of the circuit are called *gates* and each gate computes a (single-output) Boolean function of $x_1, \ldots, x_n$. Thus, if m gates of the circuit are marked as outputs, it computes a function from $B_{n,m}$. The size of a circuit is its number of internal binary gates (it is common to assume that unary gates are given for free).

Figure 1 shows an example of a circuit of size 5 computing SUM_3. It also highlights that a circuit corresponds to an extremely simple program (called a straight line program): every line of this program just applies a unary or binary Boolean operation to input bits or the results of the previous lines.

We assume that a gate of a circuit can compute any unary or binary Boolean function. It is not difficult to see that, for a given Boolean function f, the minimum size of a circuit




```
def sum3(x1, x2, x3):
  a = x1 ^ x2
  b = x2 ^ x3
  c = a | b
  w0 = a ^ x3
  w1 = c ^ w0
  return w0, w1
```

Figure 1: A circuit (known as Full Adder) and the corresponding straight line program (in `Python`) for SUM_3. The output gates are shown in bold.

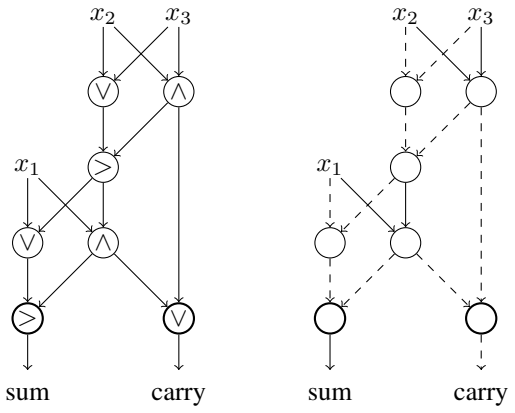


Figure 2: A circuit over the basis $B_2 \setminus \{\oplus, \equiv\}$ computing SUM_3 (left) and its AIG representation (right). The output gates are shown in bold, whereas the negated wires are shown dashed. The binary Boolean operation $>$ is defined in a natural way: $a > b = a \wedge \overline{b}$.

computing f is equal to the minimum size of a circuit for f when each gate computes either binary XOR ($\oplus$), binary AND ($\wedge$), or unary NOT ($\neg$): indeed, every binary Boolean operation is either a summation or a multiplication with possibly negated inputs and outputs. For this reason, when all unary and binary operations are allowed, we say that this is an XAIG circuit: X stands for XOR (summation), A stands for AND (multiplication), I stands for inverter (negation), and G stands for a graph.

It is well known that any Boolean function can also be computed by a circuit that only uses AND's and NOT's as operations in gates. They are called AIG circuits (Biere, Heljanko, and Wieringa 2011) and this is a convenient format for representing a circuit in practice: since every binary gate computes an AND, one just stores a graph of indegree 2 (without storing the operations computed in the gates) and a list of flags telling which of the edges are negated (or inverted). See an example in Figure 2.

Listing 1: Analyzing Boolean functions.

```
f = PyFunction(lambda xs: [sum(xs) % 2], 4)
print(f.is_monotone())

g = PyFunction.from_int_binary_func(
    lambda x, y: x + y, 2, 3)
print(g.is_symmetric())

e = TruthTable(['1101'])
print(e.is_constant())
```

Listing 2: Checking whether a circuit is satisfiable.

```
path = '../data/circuit.bench'
ckt = Circuit.from_bench_file(path)
result = is_circuit_satisfiable(ckt)
print(result.answer)
```

Tool Features

The tool has been implemented with a goal of being efficient and easy to use and extend: the code is open source[2] and written in `Python` (making the code compact and easy to read). Below, we describe the main features of the tool. For many of them, we complement their description with a short code snippet to highlight the ease of use. In the code samples below, we omit the preamble (that loads the necessary packages) to save space. The complete code snippets (runnable out of the box) can be found in the `tutorial` folder of the supplementary archive file.

Analysis

The tool allows to analyze Boolean functions and circuits.

Function Analysis Analyzing the properties of a given Boolean function is important for subsequent synthesis of an efficient circuit computing this function. Currently, the tool allows to check whether a Boolean function is monotone or symmetric (these checks are performed via enumerating all 2^n input assignments and hence are only practical when the number n of inputs is small enough). A function can be passed either as a truth table or as a `Python` function, see Listing 1.

Circuit Analysis A circuit is a particular way of representing a Boolean function. For this reason, the `Circuit` class implements `Function` interface allowing one to use all the checks described in the previous section for circuits also. Additionally, one can check whether a circuit is satisfiable. This is done by transforming a circuit into a CNF (via Tseitin transformation (Tseitin 1968)) and invoking a SAT solver via the `pysat` module (Ignatiev, Morgado, and Marques-Silva 2018), see Listing 2.

Circuit satisfiability is a ubiquitous problem as many other hard problems can be reduced to it naturally. For example, to verify whether two circuits $C_1, C_2 \colon \{0,1\}^n \to \{0,1\}^m$ compute the same function, one combines them into

[2]The code is publicly available at https://github.com/SPbSAT/cirbo.

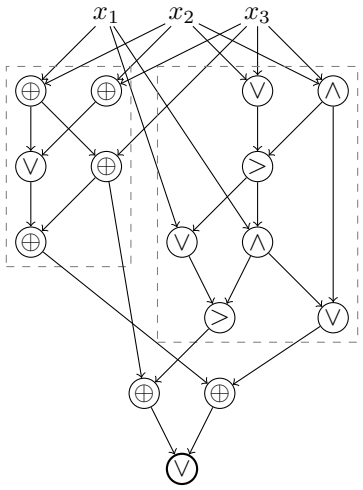


Figure 3: A miter composed out of circuits from Figures 1 and 2. Since these two circuits compute the same function, the miter is unsatisfiable.

Listing 3: Verifying that two circuits compute the same function.

```
aig = generate_sum_n_bits(3, basis='AIG')
xaig = generate_sum_n_bits(3, basis='XAIG')

mtr = Circuit().add_circuit(aig, name='aig')
mtr.connect_inputs(xaig, name='xaig')
mtr.extend_circuit(
    generate_pairwise_xor(aig.output_size),
    name='pairwise_xor',
)

outs = mtr.get_block('pairwise_xor').outputs
mtr.emplace_gate('or', OR, tuple(outs))
mtr.set_outputs(['or'])

mtr.view_graph(autorename_labels=True)
```

a miter circuit

$$M(x) = \bigvee_{i=1}^{m} (y_i \oplus z_i)$$

where $(y_1, \ldots, y_m) = C_1(x)$ and $(z_1, \ldots, z_m) = C_2(x)$). One then checks whether M is satisfiable (it is if and only if C_1 and C_2 do not compute the same function). Figure 3 gives an example. A miter can also be constructed easily in the tool, see Listing 3. The listing also illustrates two other important features of the tool. First, one can compose a circuit out of blocks. While designing a circuit, reasoning in terms of blocks, rather than in terms of gates, is more convenient. Second, one can draw a circuit. The drawing method also shows the block structure of the circuit and produces a picture similar to the one showing in Figure 3.

One can also reduce many other hard combinatorial problems to circuit satisfiability. We illustrate this for the factorization problem where one is given a positive integer k and is asked to find its nontrivial divisor (or to report that k is prime). Listing 4 shows such a reduction. Using generator for multiplying and equality, the reduction constructs a circuit that is satisfiable if and only if there exists $(n-1)$-bit non-negative integers p and q (where n is the bit length of k) such that $pq = k$. If the resulting circuit is satisfiable, the values of p and q can be read off from its satisfying assignment.

Listing 4: Reducing the factorization problem to circuit satisfiability.

```
def factorization(number: int) -> Circuit:
    n = number.bit_length()
    ckt = Circuit.bare_circuit(input_size=2
        * (n - 1))
    p, q = ckt.inputs[:n - 1], ckt.inputs[n
        - 1:]
    outs = add_mul(ckt, q, p)
    g3 = add_equal(ckt, outs, number)
    ckt.mark_as_output(g3)
    return ckt
```

Listing 5: Synthesizing a circuit for the majority function.

```
ckt = Circuit.bare_circuit(input_size=6)
b0, b1, b2 = add_sum_n_bits(ckt, ckt.inputs)
ckt.add_gate(Gate('a0', AND, (b0, b1)))
ckt.add_gate(Gate('a1', OR, ('a0', b2)))
ckt.mark_as_output('a1')
ckt.view_graph()
```

Synthesis

Using the tool, one can synthesize Boolean circuits using the following three regimes.

Manual Synthesis out of Presynthesized Blocks The tool contains generators of various circuits that are frequently used in circuit synthesis: comparators, summators, multipliers, etc. One can use them as building blocks to synthesize efficient circuits for various functions. We give an example for the majority function of six input bits. This is a symmetric function, so one can first compute the binary representation (b_0, b_1, b_2) of the sum of the input bits and then output $(b_0 \wedge b_1) \vee b_2$. Listing 5 shows how to achieve this in the tool.

Automated SAT-based Synthesis When the number n of inputs is small (say, $n \leq 10$), one can synthesize an efficient circuit for a given function using a SAT-based approach (Kojevnikov, Kulikov, and Yaroslavtsev 2009): one transforms the statement "there exists a circuit of the given size computing the given function" to CNF and checks its validity using a SAT solver. Listing 6 shows how one can automatically synthesize the Full Adder circuit (shown in Figure 1) using this approach.

Hybrid Synthesis One can combine the two strategies above by synthesizing a circuit out of built-in blocks and blocks synthesized via SAT-based approach. For example, instead of manually constructing the final block $(b_0 \wedge b_1) \vee b_2$ for MAJ_6 in the example above, one can synthesize it with

Listing 6: Synthesizing Full Adder using SAT-based approach.

```
def sum_3(x1, x2, x3):
    s = x1 + x2 + x3
    return [(s >> i) & 1 for i in range(2)]

func = PyFunction.from_positional(sum_3)
cf = CircuitFinderSat(func, 5, basis='XAIG')
ckt = cf.find_circuit()
ckt.view_graph()
```

Listing 7: Synthesizing a circuit for MAJ_6 using a hybrid approach.

```
def geq3(x: bool, y: bool, z: bool):
    s = x + 2 * y + 4 * z
    return [DontCare] if s > 6 else [True]
        if s >= 3 else [False]

ckt = generate_sum_n_bits(n=6)
pfm = PyFunctionModel.from_positional(geq3)
cfs = CircuitFinderSat(pfm, 2, basis='XAIG')
geq3_ckt = cfs.find_circuit()
ckt.extend_circuit(geq3_ckt, name='geq3')
ckt.view_graph()
```

the SAT-based approach, see Listing 7. The listing illustrates another important feature of the SAT-based synthesis: it allows to synthesize partially defined functions: in this particular case, the value of the final block on the input $(b_0 = 1, b_1 = 1, b_2 = 1)$ may be arbitrary as the sum of the input six bits can never be equal to seven.

Minimization

In the circuit minimization problem, one is given a circuit and is asked to come up with a smaller circuit computing the same function.

Low Effort Minimization The `cleanup` method performs straightforward cleaning of a circuit: for example, removes dangling and duplicate gates. To give an example, consider a task of synthesizing a circuit for MAJ_7. Clearly, this is the same as computing the most significant bit of the sum of the input bits. Hence, it makes sense to compute (the binary representation of) the sum and to output this bit. However, since we only need one bit (out of all bits of the sum), some computations can be dropped. This is indeed what happens in practice: as the result of invoking the code from Listing 8, two unnecessary gates are removed from the circuit. Also, one can run various `ABC` commands right from

Listing 8: Cleaning an XAIG circuit for MAJ_7.

```
ckt = Circuit.bare_circuit(input_size=7)
*_, b2 = add_sum_n_bits(ckt, ckt.inputs)
ckt.mark_as_output(b2)
print(ckt.gates_number())
ckt = cleanup(ckt)
print(ckt.gates_number())
```

Listing 9: Cleaning an AIG circuit for MAJ_7 using `ABC`.

```
ckt = Circuit.bare_circuit(input_size=7)
*_, lst = add_sum_n_bits(ckt, ckt.inputs,
    basis='AIG')
ckt.mark_as_output(lst)
ckt = abc_transform(ckt, 'strash; dc2')
```

Listing 10: SAT-based minimization of a circuit for SUM_5.

```
ckt = Circuit.bare_circuit(input_size=5)
x1, x2, x3, x4, x5 = ckt.inputs
a0, a1 = add_sum3(ckt, [x1, x2, x3])
b0, b1 = add_sum3(ckt, [a0, x4, x5])
w1, w2 = add_sum2(ckt, [a1, b1])
ckt.set_outputs([b0, w1, w2])
print(ckt.gates_number())
ckt = minimize_subcircuits(ckt, 'XAIG')
print(ckt.gates_number())
```

the tool. Listing 9 shows how to achieve the same task (removing redundant gates from a circuit for SUM_7 to get a circuit for MAJ_7) for the basis AIG using `ABC` cleaning.

High Effort Minimization A more powerful, but significantly less efficient minimizing method is the following: try to minimize small subcircuits of a given circuit (Kulikov, Pechenev, and Slezkin 2022). For each subcircuit, we compute a partial function computed by it and try to synthesize a more efficient circuit computing the same function using the SAT-based approach. If a more efficient circuit is found, we replace the corresponding subcircuit in the original subcircuit and iterate. For example, the code in Listing 10 synthesizes a circuit of size 12 out of two Full Adders and one Half Adder. The subsequent call to `minimize_subcircuits` improves the resulting circuit by one gate.

Database of (Nearly) Optimal Circuits

In the tool, we employ a database of circuits of all Boolean functions with at most three inputs and at most three outputs. The vast majority of them are provably optimal. Thus, each time when we need to synthesize an efficient circuit, we first check whether it is stored in the database. The number of functions is

$$\sum_{n=2}^{3}\sum_{m=1}^{3}\binom{2^{2^n}}{m} = 2\,797\,112.$$

To reduce the search space while populating the database, we used a classification approach similar to NPN-classification (Haaswijk et al. 2017). As in NPN-classification, we say that two functions are equivalent if one can be transformed into the other by permuting and negating some of the inputs and outputs. Additionally, our classification considers the permutation of outputs, as we are working with multiple outputs. Clearly, two equivalent functions have the same circuit size. This allows us to partition the set of all functions into equivalent classes and to synthesize an efficient circuit for a single representative from each class.

	XAIG		AIG	
size	classes	functions	classes	functions
1	8	60	4	48
2	74	2,160	42	1,461
3	324	36,672	142	17,720
4	1,153	266,500	373	81,996
5	2,967	892,312	949	241,428
6	3,690	1,242,704	1,759	515,611
7	1,030	354,528	2,462	773,088
8	9	2,176	2,207	730,576
≤ 9			1,087	364,400
≤ 10			224	69,664
≤ 11			6	1,120

Table 2: Distribution of classes and functions by circuit size, for all functions with three inputs and three distinct outputs, for XAIG and AIG bases.

In the XAIG basis, each function from $B_{3,3}$ has a relatively small circuit size (at most 8) and it is possible to find a provably optimal circuit via a reduction to SAT. We used a tool (Kulikov, Pechenev, and Slezkin 2022) for this task. In the AIG basis, circuits for functions from $B_{3,3}$ can have size as large as 11. For some of these functions, proving that there is no smaller circuit is already a difficult task for the state-of-the-art SAT solvers. For this reason, for several classes of functions we have an efficient circuit, but no proof that this circuit has the smallest size. We provide a detailed statistics in Tables 2.

Experimental Evaluation

Synthesizing Efficient Circuits

Recall that a Boolean function $f(x_1, \ldots, x_n) \in B_{n,m}$ is called *symmetric* if its value depends on $(x_1 + \cdots + x_n)$ only (equivalently, the function value never changes when one permutes the input). On the one hand, many interesting functions are symmetric (for example, SUM, MAJ, and SORT). On the other hand, even if a function is not symmetric, a circuit for it can rely on symmetric functions: for example, to compute the product of two n-bit integers, one first computes pairwise products of input bits and then computes the sums of these products.

SUM Since the value of a symmetric function depends on $(x_1 + \cdots + x_n)$, SUM is a fundamental symmetric function: below, we demonstrate that to get an efficient circuit computing a symmetric function, it makes sense to first compute $(b_0, \ldots, b_k) = \mathrm{SUM}_n(x_1, \ldots, x_n)$ (that is, to compress n input bits into about $k + 1 = \lceil \log_2(n + 1) \rceil$ bits) and then to compute the result out of $(b_0, \ldots, b_k)$. For this reason, it is important to have efficient circuits for SUM. A well known way to compute SUM_n is to apply blocks computing SUM_3 and SUM_2 iteratively: in particular, this leads to upper bounds

$$\mathrm{size}_{\mathrm{XAIG}}(\mathrm{SUM}_n) \leq 5n \text{ and } \mathrm{size}_{\mathrm{AIG}}(\mathrm{SUM}_n) \leq 7n\,.$$

n	3	5	7	9	11	15
XAIG	5	11	19	27	34	51
AIG	7	17	28	41	52	77

Table 3: Size of currently best known circuits computing SUM_n.

n	7	9	11	13	15
IWLS 2023	18	29	40	52	68
Our tool, 2024	17	24	31	45	48
Improvement	5%	17%	22%	13%	29%

Table 4: Size of XAIG circuits for MAJ_n: the best circuits submitted to IWLS 2023 Programming Contest and circuits synthesized by our tool.

No better construction is known for AIG, whereas for XAIG, a better upper bound is known (Demenkov et al. 2010):

$$\mathrm{size}_{\mathrm{XAIG}}(\mathrm{SUM}_n) \leq 4.5n + o(n)\,.$$

Our tool allows to generate best known circuits computing SUM_n for all n. In Table 3, we show the size of the corresponding circuits.

MAJ and SORT For MAJ and SORT, our tool improved greatly the best known circuits following the hybrid approach outlined in Listing 7:

1. first, compute $(b_0, \ldots, b_k) = \mathrm{SUM}(x_1, \ldots, x_n)$ (here, $k = \lceil \log_2(n + 1) \rceil - 1$);
2. then, using a SAT-based approach, find a circuit that computes the required function out of $(b_0, \ldots, b_k)$;
3. finally, minimize the composition of the two circuits.

Tables 4 and 5 compare the circuits constructed this way with the best circuits submitted to the IWLS 2023 Programming Contest.

Multipliers As mentioned above, efficient circuits for SUM allow to synthesize circuits for various arithmetic functions as many of them use bit summation, one way or another. A prominent example is MULT_n. With respect to circuit size, efficient long integer multiplication algorithms (like the ones by (Karatsuba and Ofman 1963) and (Schönhage and Strassen 1971)) start to outperform the

n	12	13	14	15	16
IWLS 2023	64	90	77	114	109
Our tool, 2024	58	62	68	73	82
Improvement	9%	31%	11%	35%	24%

Table 5: Size of XAIG circuits for SORT_n: the best circuits submitted to IWLS 2023 Programming Contest and circuits synthesized by our tool.

straightforward grade-school algorithm only for large values of n. Even when using the grade-school algorithm with small n, it is still unclear what is the best way of computing the sum of the pairwise products of input bits. The tool contains a number of generators of efficient multiplier circuits (which, in turn, are based on efficient SUM circuits described above).

Arithmetic Functions In the IWLS 2024 Programming Contest, it is the arithmetic functions category where we were able to achieve the most dramatic improvement in circuit size compared to the best results of 2023 (see the first four rows of Table 1). For each of the arithmetic benchmarks, we followed the same two-step approach:

1. Convert a known algorithm for computing the corresponding function to a circuit. When doing this, optimize individual parts of the circuit whenever possible.
2. Minimize the resulting circuit.

We illustrate the first step of this approach for the DIV function. Assume that

$$\mathrm{DIV}_n(x_0,\ldots,x_{n-1},y_0,\ldots,y_{n-1}) = (z_0,\ldots,z_{n-1}),$$

where x_0, y_0, z_0 are the least significant bits of the corresponding integers. Let also

$$\begin{aligned} X &= \mathrm{int}(x_0,\ldots,x_{n-1}),\\ Y &= \mathrm{int}(y_0,\ldots,y_{n-1}),\\ Z &= \mathrm{int}(z_0,\ldots,z_{n-1}) \end{aligned}$$

be the corresponding integers. Thus, $0 \le X, Y, Z < 2^n$ and $Z = \lfloor X/Y \rfloor$. We assume also that $Y > 0$. We follow the grade-school division algorithm: determine the bits of Z one by one starting from the most significant bit, each time try subtracting $2^i z_i Y$ from X. Since z_i is either 0 or 1, one does not even need to multiply by z_i: $2^i z_i Y$ is either 0 or Y shifted by i positions. Moreover, all iterations of the algorithm perform similar checks and one can precompute the required checks in advance using the dynamic programming technique. We provide the details below.

The algorithm recovers the bits z_i for $i = n-1,\ldots,0$. For the i-th iteration, by $X' = X - \sum_{j=i+1}^{n-1} 2^j z_j Y$ we denote the "remaining part" of X that still needs to be divided by Y. Then, $z_i = 1$ if and only if $X' \ge 2^i Y$. This is, in turn, equivalent to

$$Y < 2^{n-i} \wedge \mathrm{int}(x'_i,\ldots,x'_{n-1}) \ge \mathrm{int}(y_0,\ldots,y_{n-i-1})\,.$$

Finally, the first of these two conditions is $\bigvee_{j=n-i}^{n-1} \overline{y_j}$.

We check both these conditions as follows. To avoid recomputing $p_i = \bigvee_{j=n-i}^{n-1} \overline{y_j}$ at every iteration from scratch, we precompute $p_{n-1},\ldots,p_0$ using the dynamic programming approach: $p_{n-1} = \overline{y_{n-1}}$ and $p_i = p_{i+1} \vee \overline{y_i}$, for $i = n-2,\ldots,0$.

For the second part, we need to compare two $(n-i)$-bit integers. Note that if $z_i = 1$, then one also needs to update X'. It turns out that one can combine the comparison and the subtraction in the same subcircuit of size $5(n-i)$. To achieve this, we process the two integers going from the least significant bit to the most significant one. For each position, we apply the Full Adder block to the three bits: two bits of the numbers to be compared and the carry bit from the previous position. This way, the final carry bit c_i is equal to zero if and only if $\mathrm{int}(x'_i,\ldots,x'_{n-1}) \ge \mathrm{int}(y_0,\ldots,y_{n-i-1})$. Then, $z_i = \overline{c_i} \wedge \overline{p_i}$. To update X', one uses $n-i$ if-then-else blocks, each having circuit size equal to 3. Overall, the size of the resulting circuit is about $4n^2$.

benchmark	ex29	ex58	ex77	ex28	ex54
IWLS 2023	644	283	167	398	426
Our tool, 2024	587	280	165	394	421
Improvement	8%	1%	1%	1%	1%

Table 6: Size of XAIG circuits for neuron benchmarks: the best circuits submitted to IWLS 2023 Programming Contest and circuits synthesized by our tool.

Minimizing Existing Circuits

We have been able to synthesize efficient circuits for various symmetric and arithmetic functions, partly due to the fact that we knew the exact structure of these functions. At the same time, in the IWLS 2024 Programming Competition, there were many benchmarks whose structure was unclear: for example, 32 benchmarks correspond to three-output neurons from the LogicNets project (Umuroglu et al. 2020).

In such cases, we were trying to optimize either a known circuit or some inefficient circuit for the corresponding function. To do this, we combined our SAT-based minimization tool with other efficient tools like `ABC` (Brayton and Mishchenko 2010) and and `CIOPS` (Reichl, Slivovsky, and Szeider 2023): for a given circuit, we applied the three tools repeatedly. This strategy allowed us to further improve about half of the circuits synthesized for the neuron benchmarks, see Table 6.

Acknowledgments

Research of the authors 1, 2, 3, 5, 7, 10, and 11 is supported by Huawei (grant TC20211214628). Research of the author 2 is additionally supported by the grant 075-15-2022-289 for creation and development of Euler International Mathematical Institute, and by the Foundation for National Technology Initiative's Projects Support (agreement 70-2021-00187).

References

Biere, A.; Heljanko, K.; and Wieringa, S. 2011. AIGER 1.9 And Beyond. Technical report, FMV Reports Series, JKU Linz, Austria.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2021. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Brayton, R. K.; and Mishchenko, A. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, 24–40. Springer.

Demenkov, E.; Kojevnikov, A.; Kulikov, A. S.; and Yaroslavtsev, G. 2010. New upper bounds on the Boolean circuit complexity of symmetric functions. *Inf. Process. Lett.*, 110(7): 264–267.

Find, M. G.; Golovnev, A.; Hirsch, E. A.; and Kulikov, A. S. 2016. A Better-Than-3n Lower Bound for the Circuit Complexity of an Explicit Function. In *FOCS*, 89–98. IEEE Computer Society.

Find, M. G.; Golovnev, A.; Hirsch, E. A.; and Kulikov, A. S. 2023. Improving 3N Circuit Complexity Lower Bounds. *Comput. Complex.*, 32(2): 13.

Haaswijk, W.; Testa, E.; Soeken, M.; and Micheli, G. D. 2017. Classifying Functions with Exact Synthesis. In *ISMVL*, 272–277. IEEE Computer Society.

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2018. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *SAT*, volume 10929 of *Lecture Notes in Computer Science*, 428–437. Springer.

Karatsuba, A. A.; and Ofman, Y. 1963. Multiplication of many-digital numbers by automatic computers. *Dokl. Akad. Nauk SSSR*, 145(2): 293–294.

Kojevnikov, A.; Kulikov, A. S.; and Yaroslavtsev, G. 2009. Finding Efficient Circuits Using SAT-Solvers. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, 32–44. Springer.

Kulikov, A. S.; Pechenev, D.; and Slezkin, N. 2022. SAT-Based Circuit Local Improvement. In *MFCS*, volume 241 of *LIPIcs*, 67:1–67:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Li, J.; and Yang, T. 2022. $3.1n - o(n)$ circuit lower bounds for explicit functions. In *STOC*, 1180–1193. ACM.

Reichl, F.; Slivovsky, F.; and Szeider, S. 2023. Circuit Minimization with QBF-Based Exact Synthesis. In *AAAI*, 4087–4094. AAAI Press.

Santhanam, R. 2022. Why MCSP Is a More Important Problem Than SAT (Invited Talk). In *FSTTCS*, volume 250 of *LIPIcs*, 2:1–2:1. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Schönhage, A.; and Strassen, V. 1971. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4): 281–292.

Soeken, M.; Riener, H.; Haaswijk, W.; and Micheli, G. D. 2018. The EPFL Logic Synthesis Libraries. *CoRR*, abs/1805.05121.

Tseitin, G. 1968. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 115—-125.

Umuroglu, Y.; Akhauri, Y.; Fraser, N. J.; and Blott, M. 2020. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. In *FPL*, 291–297. IEEE.