SAT-Based Area Recovery in Technology Mapping

Bruno Schmitt

Ecole Polytechnique Federale de Lausanne (EPFL)

bruno@oschmitt.com

Alan Mishchenko Robert Brayton

Department of EECS, UC Berkeley

{alanmi, brayton}@berkeley.edu

Abstract

This paper proposes a new SAT-based algorithm for recovering area in technology mapping. The algorithm considers a sequence of relatively small overlapping regions of a mapped network and computes an improved mapping of each using a SAT solver. Delay constraints are taken into account by interfacing the SAT solver with a timer. Experimental results are given for mapping into 6-LUTs. An average reduction in area on top of a high-effort area-only synthesis/mapping flow was 3-4% while for some benchmarks the area reduction was more than 10%.

1. Introduction

Technology mapping expresses a Boolean network representing the functionality of a hardware design as a set of primitives from a technology library. In mapping into LUT-based FPGAs, the library is composed of *K*-input LUTs, where $K = 4$ or $K = 6$ for most commercial FPGAs.

LUT-based mappers measure area in terms of a *LUT count*, and delay in terms of *LUT level,* defined as the largest number of LUTs on any path from primary inputs to primary outputs. Reducing area and delay of a LUT mapping is an important goal that can be achieved using heuristic structural mappers, such as [11], followed by post-mapping Boolean re-synthesis, such as [12].

This paper proposes a SAT-based algorithm for reducing area of a given LUT mapping, with or without delay constraints. The proposed algorithm uses a SAT solver to find minimum-area *K*-LUT covers of a sequence of small multi-input and multi-output regions of an initial *K*-LUT mapping. The optimization performed is purely structural; it does not exploit the Boolean nature of the network and does not use don't-cares as previous SAT-based optimizers [12][13]. Currently it works only for LUT-based FPGAs, but the ideas extend to standard cells and other implementation technologies.

The method can optimize area under delay-constraints, but instead of encoding delay constraints in the CNF as in previous work [9], the SAT solver is interfaced with a dedicated timer. This is similar to an SMT solver composed of a solution enumerator and a solution evaluator. A similar approach based on decoupling of SAT-based enumeration and application-specific evaluation has been used in [8].

Experiments show that the proposed engine gives sizeable improvements for circuits already mapped by a high-effort area-oriented synthesis and mapping flow. These improvements indicate that area-recovery heuristics used in mainstream mappers do not perform well for deep AIGs with high internal fanout.

Currently, we use this SAT-based area recovery re-mapping as a post-processing step after a mainstream structural mapper. In the future, SAT-based mapping may become an integrated *part* of main-stream mappers, replacing complicated, error-prone, and often contradictory heuristics used to recover area under delay constraints. Such an integration may lead to mappers that are faster and produce better results. This can be similar to the effect obtained by the use of SAT-based detailed placement, which has improved runtime and quality of results, compared to a traditional detailed placer based on simulated annealing [9][10].

The rest of the paper is organized as follows. Section 2 contains relevant background. Section 3 shows the high-level view of the proposed SAT-based engine. Section 4 describes the components of the engine. Section 5 contains experimental results, and Section 6 concludes the paper.

2. Background

2.1 Boolean Function

In this paper, *function* refers to a completely specified Boolean function $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$. The *support* of function *f* is the set of variables *X*, which can influence the output value of *f*. The *support size* is denoted by $|X|$.

The *truth table* of an *n*-input Boolean function is a bit-string containing 2^n bits representing values of the function under all possible 2^n input assignments ordered by their integer representation. For example, the truth table of a 3-input majority gate, denoted MAJ3 in the paper, contains 8 bits, 11101000, as shown in the last column of Figure 1.

2.2 Boolean Network

A *Boolean network* (or *circuit*) is a directed acyclic graph (DAG) with nodes corresponding to Boolean functions and edges corresponding to wires connecting the nodes.

A node *n* may have zero or more *fanins*, i.e. nodes driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A *transitive fanin (fanout) cone* (TFI/TFO) of a node is a subset of nodes of the network, which are reachable through the fanin (fanout) edges of the node. The *TFO support* of a node is the set of primary outputs reachable through the fanouts of the node.

2.3 And-Inverter Graph (AIG)

And-Invertor Graph (AIG) is a combinational Boolean network composed of two-input AND gates and inverters. An AIG for a Boolean network can be derived by factoring the functions of the logic nodes found in the network. The AND/OR gates in the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule and added to the AIG in a topological order.

A *cut C* of a node *n* is a set of nodes of the network, called *leaves* of the cut, such that each path from a PI to *n* passes through at least one leaf. Node *n* is called the *root* of cut *C*. The cut *size* is the number of its leaves. A *trivial cut* of a node is the cut composed of the node itself. A cut is *K-feasible* if the number of nodes in the cut does not exceed *K*. *Cut enumeration* is used by a cut-based technology mapper, such as [11], to compute cuts using dynamic programming, starting from PIs and ending at POs.

2.4 LUT Mapping

A *K*-input lookup table (*K*-LUT) is a hardware device, which can implement any Boolean function up to *K* inputs. A Boolean network can be *mapped* into *K*-LUTs by a software package called a *technology mapper*. The mapper takes a *subject graph*, which is a technology-independent representation of the network, such as an AIG, and returns a *mapping*, which is a set of LUTs covering the subject graph. Each internal node of the subject graph is either *used in the mapping* (if the mapping includes a LUT rooted in this node) or not used in the mapping (otherwise). A mapping is *valid* if the internal nodes driving the POs of the network are used in the mapping and, for each LUT of the mapping, its fanins are used in the mapping, or are PIs.

In this work, we assume that the network is already mapped using *K*-LUTs and the resulting mapping is valid. Furthermore, we do not consider any technology-dependent information associated with the LUTs, except their connectivity.

2.5 Boolean Satisfiability

A *satisfiability problem* (SAT) takes a propositional formula representing a Boolean function and decides if the formula is satisfiable or not. The formula is *satisfiable* (SAT) if there is an assignment of variables that evaluates the formula to 1. Otherwise, the formula is *unsatisfiable* (UNSAT). A software program that solves SAT problems is called a *SAT solver*. SAT solvers provide a satisfying assignment when the problem is satisfiable.

Modern SAT solvers can accept assumptions, which are single-literal clauses holding for one call to the SAT solver. The process of determining the satisfiability of a problem under given assumptions is called *incremental SAT solving*.

2.6 Conjunctive Normal Form (CNF)

To represent a propositional formula in the SAT solver, important aspects of the problem are encoded using Boolean *variables*. The presence or absence of a given aspect is represented by a positive or negative *literal* of a variable. A disjunction of literals is called a *clause*. A conjunction of clauses is called a *CNF*. CNFs are processed by CNF-based SAT solvers, such as MiniSAT [4].

CNF is composed of clauses encoding different aspects of the SAT problem. For example, some CNF clauses may encode *covering constraints*, that is, requirements for each node that, if a node is mapped, its fanins are mapped. Another important constraint type present in many SAT problems, is introduced in the next section.

2.7 Cardinality Constraints

A *cardinality constraint* states that, among N ($N > 1$) Boolean variables $X=\{x_1, x_2, \ldots x_N\}$, less than K ($K \leq N$) have value 1. A naïve way to represent this constraint is to generate $N!/(K!*(N-K)!)$ clauses, each ruling out the case when a specific set of K out of N given variables are 1 at the same time. However, this representation leads to a prohibitive number of CNF clauses for those values of N and K that often appear in practice. For example, for $N = 64$ and $K = 16$, the naïve approach gives $\approx 4.8*10^{14}$ clauses.

Efficient CNF representation has been a topic of active research for more than a decade. It has been shown [5] that *pseudo-Boolean constraints* (a generalization of cardinality constraints when the values of Boolean variables are multiplied by integer coefficients) can be efficiently expressed by *sorting networks*. The resulting CNFs lead to faster SAT because sorting networks contain only ANDs and Ors, and do not contain XORs and MUXes appearing in adders, BDDs, and other proposed representations.

Moreover, is been shown [1] that exactly one half of the clauses used to encode a cardinality constraint represented as a sorting network do not have to be added to the SAT solver. This new representation of cardinality constraints is called *cardinality networks*. Finally, it has been shown [3] that, among the two equal-cost implementations of sorting networks, the one known as *pair-wise sorting network* [14] has better implicativity (creates more implications) and therefore leads to substantially faster SAT solving. Our representation of cardinality constraints uses *cardinality networks* derived from *pair-wise sorting networks*.

3. SAT-Based Engine

This section outlines the proposed SAT-based structural mapping engine shown in Figure 1. The *input* to the engine is an AIG mapped into *K*-input LUTs. The *output* is the same AIG with a different *K*-input LUT mapping assigned. The new mapping is expected the same or better area and delay, compared to the original mapping.

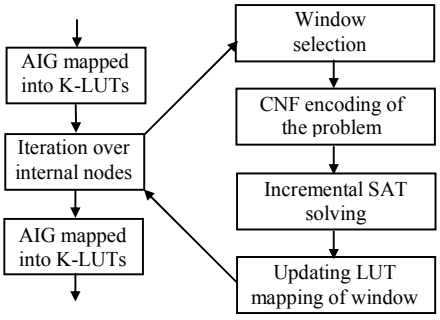

Figure 1: Overview of the proposed engine.

LUTs of the current mapping are considered in a topological order, although a different order could be used.

For each LUT, a window is computed (Section 4.1). The window contains the given LUT together with other LUTs in its TFI and TFO. The complete set of structural *K*-LUT covers of the window is represented in CNF (Section 4.2). The SAT solver takes this CNF and looks for an improved mapping (Section 4.3). If an improved mapping is found, the current mapping is updated (Section 4.4). The delay constraints are handled as shown in Section 4.5.

When computation for a given LUT is finished, the mapper moves to the next LUT in the order. When all LUTs have been considered or when a resource limit, such as a global timeout, has been reached, the mapper outputs an improved LUT mapping.

4. SAT-Based Engine

This section describes the components of the SAT-based remapping engine in detail, starting with window selection.

4.1 Window Selection

A *window* is a small region of the subject graph processed by the SAT-based engine at one time. A window is computed for an AIG mapped into LUTs and is centered at a LUT represented by the root node sand one of its structural cut. Initially, the window contains only this LUT. The windowing algorithm extends the window by including adjacent LUTs whose root nodes are fanins or fanouts of the LUTs currently included in the window. When deciding what next LUT should be added to the window under construction, priority is given to the LUT that increases the size of the window as little as possible. Such windows have a higher optimization potential, allowing the engine to remap them into fewer LUTs.

We do not limit the number of LUTs contained in the window but limit the number of AIG nodes. This is because each AIG node adds one SAT solver variable and one input to the cardinality constraint. Therefore, controlling the number of AIG nodes in the window is important for ensuring the scalability of the mapper.

Although our current implementation scales up to 128 AIG nodes, it was found experimentally that a good tradeoff between runtime and quality is achieved for windows contains up to 32 nodes. Smaller windows often do not improve much, while larger windows can improve more but they are more likely to time out.

A computed window is represented as an ordered set of internal AIG nodes included in the window. Once this set is known, it is easy to compute *window leaves* (the nodes not included, having at least one fanout that is included) and *window roots* (the nodes included, having at least one fanout that is not included). These two sets of nodes are used for CNF construction described in Section 4.2.

The windowing algorithm, when applied to different nodes, can result in identical windows. To avoid the same window from being processed repeatedly, window selection employs a hash table, which caches windows that have been tried and did not lead to improvement.

4.2 CNF Encoding

The CNF used to present the set of all possible structural mappings of the window contains two types of variables: (1) each internal AIG node i is represented by variable n_i having value 1 iff the node is used in the mapping; (2) each n-input cut k ($2 \le n \le K$ where K is the LUT size) of node i is represented by variable c_{ik} having value 1 iff node i is used in the mapping and cut k is used to map node i.

The resulting CNF contains four types of constraints:

- If node i is used, one of its cuts is used: $n_i \rightarrow \mathbf{V}_k\, c_{ik}$.
- If cut k is used, all cut leaves are used: $c_{ik} \rightarrow \mathbf{\Lambda}_f\, n_f$.
- The nodes driving the outputs are used: $\mathbf{\Lambda}_o\, n_o$.
- The cardinality constraint: $\mathbf{\Sigma}_i\, n_i < L$ where L is the LUT count of the current mapping in the window.

The window leaves do not have to be represented by SAT variables because they are always used in the mapping. Thus, if a cut leaf is a window leaf, its SAT variable is assumed to have value 1. Similarly, the window roots do not have to be represented by SAT variables because they are always used in the mapping. In this case, the third constraint ($\mathbf{\Lambda}_o n_o$) is omitted and the respective n_i variables in other constraints can be assumed to have value 1.

The CNF generated for a typical window with 32 AIG nodes contains roughly a thousand of CNF clauses and about half of them are due to the cardinality constraint.

4.3 Incremental SAT Solving

The CNF computed as shown above is loaded into the SAT solver. The cardinality constraint is set to a value that is smaller by one than the number of LUTs in the window. If such solution is found, the cardinality constraint is tightened again. The sequence of incremental SAT calls is continued until the solver returns UNSAT, or until a resource limit is reached. The last feasible solution whose quality is strictly better than that of the initial mapping is used to update the current mapping of the window.

The above procedure is made faster through the use of the initial solution. This improvement requires that, before the SAT solver begins its search for an improved mapping, the SAT variables were initialized to values they have in the current LUT mapping of the window. This way, if the SAT solver runs with the cardinality constraint disabled, it produces the current mapping without search. Now, when the cardinality constraint *is* asserted, the solver tends to look for the improved solution in the vicinity of the known solution. This typically reduces the time it takes the SAT solver to find the next feasible solution.

4.4 Updating LUT Mapping

To enable efficient window selection and updating of the LUT mapping, the internal nodes of the AIG representing the subject graph are annotated with their status in the current LUT mapping. For this, each AIG node used in the mapping is put in correspondence with a K-feasible cut used to represent this cut in the current mapping.

When a mapping of the window is updated by the SAT-based mapper, the old annotation of the internal nodes of the window is invalidated and the new annotation is created to reflect the update. As a result, the mapping of the subject graph is valid after each update. Thus, if the computation exceeds a resource limit, the current mapping can be returned, resulting in a valid mapping of the original AIG.

4.5 Handling Delay Constraints

In the case of the LUT count optimization, the mapper focuses on reducing area without considering delay. If delay constraints are given, they can be converted into CNF and solved as part of the SAT problem [9]. However, in our experience, the delay constraints substantially increase CNF size and slow down the SAT solver. One way to avoid this problem is to design a dedicated SMT solver capable of handling both Boolean constraints representing a valid cover and linear constraints representing timing [10].

A simpler approach to handle delay constraints as part of the SAT based enumeration is to follow another approach inspired by the design of an SMT solver [8].

The main idea there is to interface the SAT solver with a dedicated timer applicable to mappings founds by the SAT solver. If the timer determines a mapping to be acceptable from the delay point of view, a resulting solution is found. Otherwise, the timer produces a critical path, which is a subset of LUTs used in the mapping that do not meet the timing. These LUTs are represented by specific SAT variables having value 1 in the current solution.

The current critical path can be ruled out by generating a blocking clause, which does not allow it to appear in a resulting mapping. When this clause is added to the SAT solver, it will never produce a mapping with the same critical path. The number of delay-violating critical paths in a small structural window is usually quite limited. The proposed integration of the SAT solver and the timer quickly finds a timing-feasible solution, or enumerates all critical paths and finds that no such feasible solution exists.

5. Experimental Results

The presented SAT-based remapping is implemented as command *&satlut* in ABC [2]. The implementation has been tested using a suite of EPFL benchmarks [6] mapped into 6-LUTs. The best area results available for these benchmarks [7] was used as input to *&satlut.*

Command *&satlut* was run twice: first with the default settings (-N 32 -C 100) and second with the high-effort settings (-N 64 -C 10000), where switches N and C specify the limits on AIG node count in the window and on SAT conflicts, respectively. The results produced by *&satlut* have been checked for correctness using the combinational equivalence checker *&cec*.

The detailed results are reported in Table 1. The first section of the table shows the parameters for the area-optimized versions of the designs available from [7]. The parameters include the LUT count (column "LUT") and the LUT level (column "Level"). The column "AIG" shows the number of nodes in the AIG representation when the LUT network in BLIF format is entered into ABC and converted into a mapped AIG using command *&get -m*.

The second section of Table 1 shows the results after applying *&satlut* with default settings, including the LUT count (column "LUT"), the LUT level (column "Level"), and the runtime of *&satlut* in seconds (column "Time, s"). The third section shows similar results when *&satlut* is applied with the high-effort settings.

The table shows that *&satlut* reduces both area and delay. It is remarkable that an average area reduction of 3.5% is achieved with the default settings in a very short time. For several arithmetic benchmarks, the area reduction is very substantial: *div* (9.5%), *log2* (7.5%), *mult* (11.7%), *square* (11.3%). We speculate that the large improvements are possible because these benchmarks have regular circuit structures, which can mislead area-recovery heuristics used in the structural LUT mapper.

A delay limit imposed for each example in the experiments was the delay of the original LUT mapping. Even so, a delay improvement was achieved in most examples because the original mapping was done just for area. However, after area was substantially reduced, delay was reduced also.

After removing 10 smaller examples whose area did not change or changed by less than 5 LUTs (because they were likely near minimum already), we are left with 9 examples marked with an asterisk in the table. For this subset, the default settings improve area and delay by 7.6% and 13%, respectively, while the high-effort settings improve by 8.2% and 13.5%, respectively. These ratios are listed on the "Geomean*" row in Table 1.

Similarly, Table 2 lists results for delay-optimized version of the EPFL benchmarks. In this case, *&satlut* performed only area-recovery without trying to further reduce the delay. More area would be saved if the engine considered the slack on near-critical paths. However, the engine is currently integrated in such a way that it does not increase delay on any path without using the slack.

6. Conclusions and Future Work

The paper describes a SAT-based engine for recovering area after AIGs have been mapped into *K*-input LUTs. The optimization is purely structural and does not exploit functional properties of the design as in previous SAT-based methods [12][13].

The engine, *&satlut*, gives substantial improvements in area even after a high-effort synthesis and mapping flow had been applied to those circuits. It can optimize area under delay-constraints. Delay constraints are not encoded in the CNF, as in past work [9]. Instead, the SAT solver is interfaced with a dedicated timing engine, making it look like an SMT solver composed of two component solvers: a structural cover enumerator and a timer.

To our knowledge, the proposed engine is the first that can find exact minimum-area solutions for non-trivial multi-input and multi-output AIGs. Given the current state-of-the-art in SAT solvers and CNF encoding, exact solutions are feasible for relatively small netlists composed of 10-20 LUTs.

Future work will proceed in the following directions:

- Further improving CNF representation of cardinality constraints. The motivation is that these constraints take about 50% of the clauses in a typical SAT instance; so any reduction in their number should translate into increased scalability of the mapper.
- Extending SAT-based mapping to work for standard cells. A preliminary implementation confirmed that the approach is practical and leads to area savings.
- Creating a hybrid structural/functional SAT-based optimization engine, which exploits both the efficient structural solution of the covering problem and the functional nature of the underlying Boolean network.

7. Acknowledgements

This work was partly supported by NSF/NSA grant ”Enhanced equivalence checking in cryptoanalytic applications” at University of California, Berkeley.

We also would like to thank Jie-Hong Roland Jiang for discussions in Taiwan where this paper was written.

8. References

[1] R. Asin, R. Nieuwenhuis, A. Oliveras, and E. Rodriguez-Carbonell, “Cardinality networks and their applications”, *Proc. SAT’09*, Springer, LNCS 5584, pp. 167-180.

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[3] M. Codish and M. Zazon-Ivry, “Pairwise cardinality networks”, *Proc. LPAR’10*, Springer, LNCS 6355, pp. 154-172. http://www.cs.bgu.ac.il/~mcodish/Papers/Sources/lpar16.pdf

[4] N. Een and N. Sörensson, “An extensible SAT-solver”, *Proc. SAT’03*, LNCS 2919, pp. 502-518. http://minisat.se/downloads/MiniSat.pdf

[5] N. Een and N. Sörensson, “Translating pseudo-Boolean constraints into SAT “, *Journal of SAT*, Vol. 2, 2006, pp. 1-26. http://minisat.se/downloads/MiniSat+.pdf

[6] EPFL Benchmarks. http://lsi.epfl.ch/benchmarks

[7] EPFL Benchmarks. The best area results for mapping into 6-LUTs: https://documents.epfl.ch/groups/b/be/benchmarks/www/benchmark_best_results.zip

[8] V. Ganesh, C. W. O’Donnell, M. Soos, S. Devadas, M. C. Rinard, and A. Solar-Lezama “Lynx: A programmatic SAT solver for the RNA-folding problem”, *Proc. SAT’12*, LNCS 7317, pp. 143-156. https://people.csail.mit.edu/devadas/pubs/programmaticSAT.pdf

[9] A. Mihal and S. Teig, “A constraint satisfaction approach for programmable logic detailed placement”, *Proc. SAT’13*, LNCS 7962, pp. 208–223.

[10] A. Mihal, “A difference logic formulation and SMT solver for timing-driven placement”, *Proc. SMT’13*.

[11] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*, pp. 354-361. https://people.eecs.berkeley.edu/~alanmi/publications/2007/iccad07_map.pdf

[12] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang. "Scalable don't-care-based logic optimization and resynthesis", *ACM TRETS*, Vol. 4(4), April 2011, Article 34.

[13] A. Mishchenko, R. Brayton, T. Besson, S. Govindarajan, H. Arts, and P. van Besouw, "Versatile SAT-based remapping for standard cells", *Proc. IWLS'16*.

[14] I. Parberry, "The pairwise sorting network", *Parallel Processing Letters*, 2 (2, 3), 1992, pp. 205–211. http://larc.unt.edu/ian/pubs/pairwise.pdf

Table 1: The results of applying &satlut to EPFL benchmarks mapped into 6-LUTs for area.

	Area-optimized statistics			&satlut –N 32 –C 100			&satlut –N 64 –C 10000		
Design	AIG	LUT	Level	LUT	Level	Time, s	LUT	Level	Time, s
adder	1981	201	73	201	73	0.05	201	73	0.11
arbiter *	1542	429	24	418	24	0.12	413	23	21.97
barrel	3840	512	4	512	4	0.08	512	4	0.25
cavlc	951	107	6	106	6	0.03	106	6	1.10
ctrl	169	28	2	28	2	0.01	28	2	0.01
dec	1072	272	2	272	2	0.02	272	2	0.02
div *	37378	3813	1542	3454	1211	1.03	3435	1217	6.07
i2c	1114	215	7	213	6	0.04	213	6	0.51
int2float	255	34	4	34	4	0.02	34	4	0.19
log2 *	46748	7344	142	6796	126	2.87	6633	121	284.48
max	3108	532	192	528	190	0.38	528	190	103.81
mem_ctrl *	10680	2125	23	2106	22	0.62	2103	23	87.43
mult *	54684	5681	120	5019	80	1.43	4923	90	11.32
priority	492	118	27	114	26	0.12	114	26	31.15
router	111	26	6	26	6	0.02	26	6	2.83
sin *	10209	1347	62	1285	55	0.42	1242	53	59.98
sqrt *	38306	3286	1180	3209	1116	0.75	3202	1109	3.41
square *	36468	3798	116	3371	88	1.16	3270	86	6.80
voter *	25699	1521	18	1395	17	0.39	1354	17	1.70
Geomean		1.000	1.000	0.965	0.923	1.000	0.957	0.924	17.854
Geomean *		1.000	1.000	0.934	0.864	1.000	0.918	0.865	23.244

Table 2: The results of applying &satlut to EPFL benchmarks mapped into 6-LUTs for delay.

	Area-optimized statistics			&satlut –d –N 32 –C 100			&satlut –d –N 64 –C 10000		
Design	AIG	LUT	Level	LUT	Level	Time, s	LUT	Level	Time, s
adder	2839	419	6	410	6	0.14	415	6	1.86
arbiter *	1834	542	6	533	6	0.13	533	6	2.67
barrel	3840	512	4	512	4	0.09	512	4	0.29
cavlc	956	120	4	119	4	0.03	115	4	2.87
ctrl	169	28	2	28	2	0.01	28	2	0.01
dec	1072	272	2	272	2	0.01	272	2	0.02
div *	74605	14576	238	14530	238	4.42	14553	238	18.55
i2c	1137	234	3	230	3	0.07	229	3	1.78
int2float	261	44	3	41	3	0.03	41	3	1.89
log2 *	57309	9275	55	9140	55	4.25	9221	55	33.27
max	4504	899	10	893	10	0.31	882	10	116.39
mem_ctrl *	10874	2234	6	2215	6	0.59	2216	6	16.13
mult *	54813	7095	29	6951	29	4.87	6944	29	8.57
priority	568	158	4	156	4	0.16	157	4	34.53
router	110	30	4	30	4	0.05	30	4	1.74
sin *	11087	1835	30	1801	30	0.65	1803	30	4.36
sqrt *	62832	11745	254	11711	254	3.36	11687	254	8.63
square *	41891	4201	11	4094	11	2.38	4036	11	8.09
voter *	26500	1515	12	1501	12	0.63	1469	12	2.23
Geomean		1.000	1.000	0.987	1.000	1.000	0.983	1.000	11.844
Geomean *		1.000	1.000	0.987	1.000	1.000	0.984	1.000	5.737