Unate Decomposition of Boolean Functions

James Jacob
Intel Corporation
Strategic CAD Labs
Hillsboro, OR 97124
jjacob@ichips.intel.com

Alan Mishchenko*
Portland State University
Dept. of Electrical and Computer Engineering
Portland, OR 97207, USA
alanmi@ee.pdx.edu

Abstract

We propose a new way of decomposing completely or incompletely specified Boolean functions into a set of unate functional blocks to obtain a good initial structure for logic synthesis. The input to our algorithm is a flattened netlist. The output is a multi-level netlist that can be efficiently implemented using a logic synthesis tool. Experimental results on Espresso PLA benchmarks employing a state of the art commercial synthesis tool show that our technique leads to an average 11 % improvement in the area after technology mapping, without sacrificing on speed.

1 Introduction

Logic synthesis has a goal of implementing logic functions under timing, area, and power constraints. In many cases, the circuit quality can be improved by restructuring the netlist.

It is known that logic synthesis tools (example: SIS [1], or Synopsys Design Compiler [2]) perform better when the input netlists exhibit certain desirable properties. In particular, we have observed in our synthesis experiments that unate Boolean functions yield better results after technology mapping compared to binate Boolean functions. It can be explained by considering typical standard-cell libraries (for example, *mcnc.genlib* and *msu.genlib* included in SIS distribution). These libraries include mostly unate gates, that is, gates represented by unate functions, e.g. !((a*b) + (c*d)), with binate gates limited only to MUX, XOR, and XNOR. Another possible reason is that the underlying algorithms in technology independent optimization are mostly *algebraic*; they treat each variable and its complement as independent, unrelated variables. When a node is unate, each variable appears only in one polarity, leading to a simpler problem for algebraic decomposition, yielding better final results. Further, it is also known that logic minimization of unate functions can be efficiently done as their minimum sum-of-product (SOP) forms consist only of essential primes.

In this paper, we propose an algorithm to transform a given netlist into a multi-level network such that nearly all of its nodes are unate functions. We call this transformation *unate decomposition*. Earlier work in this area includes [12].

Essentially, unate decomposition can be viewed as a sequence of three steps: cover minimization, phase assignment, and selection of the largest unate component. In the process, the algorithm exploits the external don't-cares as well as the internal don't-cares generated during decomposition.

Software implementing our algorithm can efficiently process very large covers (more than 5000 cubes). The robustness is achieved by implementing cube manipulation needed for unate decomposition implicitly using Zero-suppressed binary Decision Diagrams (ZDDs) [6].

The rest of the paper is organized as follows. Section 2 introduces the basic terminology. Section 3 presents details of the algorithm. Section 4 discusses ZDD-based procedures for cube manipulation. Section 5 gives experimental results. Section 6 summarizes the paper.

2 Definitions

Let f: $B^n \rightarrow B$, $B \in \{0,1\}$, be a completely specified Boolean function and let F: $B^n \rightarrow \{0,1,-\}$ be an incompletely specified Boolean function represented by two completely specified functions: *on-set*, $F^{-1}(1)$, and *don't-care-set*, $F^{-1}(-)$. A *literal* is a propositional variable or its negation, e.g. a, $\overline{b}$. A *product*, or cube, is a Boolean AND of literals, e.g. $\overline{b}c$. A *cover* is a set of products. The *cardinality* of a cover is the number of cubes in the cover. A *complement*, or a negative phase, of cover S is a cover T such that the union of S and T is a *tautology* (constant 1 Boolean function).

A completely specified function can be represented by a set of cubes. This representation is known as a two-level *sum-of-product representation* (SOP). An SOP, or cover, is *irredundant*, if no cube can be removed without reducing the area of the covered Boolean space and no two cubes can be combined into one cube.

An SOP, or cover, is *minimum* if it contains the minimum number of cubes among all the SOPs covering the same area of the Boolean space. An SOP, or cover, is *minimal* if it is the local

* This work was supported by a research grant from Intel Corporation.

minimum in the number of cubes. Every irredundant cover is a minimal cover. The problem of *phase assignment* consists in finding a minimal representation of both the cover and its complement and selecting the representation with a smaller cardinality.

A cover is *positive* (*negative*) *unate in variable* x, if x appears only in its positive (negative) literal, x($\overline{x}$).

A completely specified function is *positive unate in variable* x, if $F_{x=1} \supseteq F_{x=0}$, where $F_{x=1}$ ($F_{x=0}$) is the positive (negative) cofactor of F with respect to (w.r.t.) variable x. Similarly, a function is *negative unate in* x, if $F_{x=0} \supseteq F_{x=1}$.

A cover (function) is called *unate* if it is unate in all its variables. A cover (function) is called *binate in variable* x if it is neither positive nor negative unate in variable x. A cover (function) is called *binate* if it is binate in at least one of its variables.

The following properties can be proved:

Property 1: A completely-specified function F is positive (negative) unate iff its irredundant SOP is positive (negative) unate.

Property 2: If a completely specified function or a cover is positive (negative) unate in variable x, then its complement is negative (positive) unate in variable x.

3 Unate Decomposition Algorithm

This section presents the main contribution of this paper: an algorithm to decompose an incompletely specified function into a set of unate functional blocks.

The Main Decomposition Loop

The pseudo-code of the procedure implementing the main unate decomposition loop is shown in Fig. 1. Procedure UnateDecomposition() takes the on-set and the dc-set of an incompletely-specified Boolean function. it calls GetMinCover2() to find a minimal cube cover of the function. GetMinCover2() performs phase assignment, that is, it minimizes both positive and negative phases and returns the cover with less cubes.

```
UnateDecomposition( func OnSet, func DcSet )
{
   cover C = GetMinCover2( OnSet, DcSet );

   while ( | C | > CubeLimit )
   {
      cover UnateCubes = GetUnateSubset( C );
      if ( |UnateCubes| < nCubesMin )
           break;

      cover Block = GetMinCover1( UnateCubes );
      AddComponent( Block );

      OnSet = OnSet - Area( UnateCubes );
      DcSet = DcSet + Area( UnateCubes );

      C = GetMinCover2( OnSet, DcSet );
   }

   CreateNetlistFromAccumulatedComponents();
}
```

Fig. 1. The pseudo-code of unate decomposition algorithm.

The main loop of the algorithms processes the cube cover as long as its cardinality exceeds the given cube limit (`CubeLimit`). This number is set to four by default but can be changed by the user by specifying it as a command line option. Inside the main loop, procedure GetUnateSubset() is called to heuristically select a subset of cubes that, taken separately, is a unate cover. In most cases, selection of such a subset is possible because the input cover, being binate, contains subsets of unate cubes.

If the cardinality of the unate subset returned is below the cube limit, the main loop is interrupted. Ending the loop at this point means that the cover cannot yield a unate component, whose cardinality is larger than or equal to the specified limit. Otherwise, the algorithm proceeds by minimizing and phase assigning the unate component just found. GetMinCover1() differs from GetMinCover2() only in that it takes one argument, the cover to minimize, instead of two Boolean functions representing the on-set and the dc-set. When the minimized cover is found, it is added to the accumulated blocks by procedure AddComponent().

Once a unate subset is successfully identified, the on-set and the dc-set are updated taking into account the most recently extracted unate component. Procedure Area() converts the unate component into a completely specified function. This part of the Boolean space is now subtracted from the on-set of the original function and added to the dc-set because it no longer needs to be covered. Finally, at the end of the loop, the minimal cover of the remainder is found by calling GetMinCover2().

When the control flow exits from the main loop, composing together the unate components found during decomposition yields the output netlist. Notice that the last component added to the list is not necessarily unate. Practically, the cardinality of the last binate component tends to be relatively small compared to the sum total of cubes in the extracted unate blocks.

The structure of a typical netlist resulting from running the unate decomposition algorithm is shown in Fig. 2. Blocks marked by “U” denote unate components found by the algorithm. The block marked by “B” is the only potentially binate block of the remainder. The rhombic boxes on the wires connecting the blocks with OR-gates stand for optional inverters. An inverter is introduced if during the phase assignment stage the complement of a cover was selected.

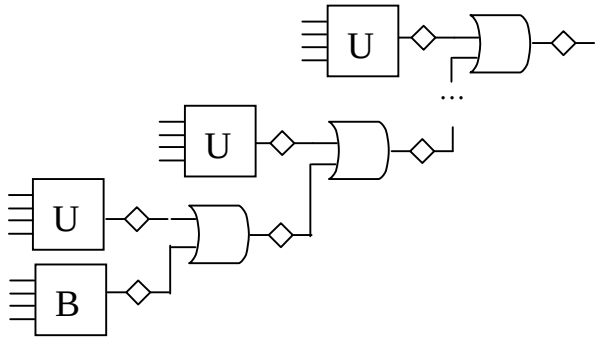

Fig. 2. Netlist structure after unate decomposition.

If the input function is multiple-output, in our current implementation, the algorithm decomposes each output independently and these netlists are merged together to create

the final netlist. In this way, the information about common subfunctions between the component functions is lost. This can have a negative effect in those cases where there is a significant sharing of logic between the logic cones of the outputs. Selective flattening and decomposition applied to collapsed internal nodes rather than to the entire output cones can alleviate this problem.

Functions Called from the Main Loop

This subsection discusses functions called from the main decomposition loop. GetMinCover1() and GetMinCover2() are implemented as wrappers around a call to a SOP minimizer. In the early version of our algorithm, these procedures called Espresso with option so_both, which minimizes the cover with phase assignment. Later we found that calls to Espresso may be replaced by Irredundant Sum-of-Product (ISOP) computation [5,6]. ISOP is called for both phases of the cover. Only the cover with a smaller cube count is returned. Using ISOP instead of Espresso resulted in several orders of magnitude speed-up without effecting the quality of results.

The implementation of procedure GetUnateSubset() is shown in Fig. 3. It greedily computes the largest unate subset by selecting cubes that are likely to form an unate subset with the largest number of cubes in the cover. It is reasonable to expect that the fewer literals the cube has, the more likely it is part of a large unate subset. To this effect, inside the loop, there is a call to LargestCubes(), which returns the set of cubes of the cover having the minimum number of literals. The implementation of LargestCubes() is discussed in Section 4 of the paper.

```
cover GetUnateSubset( cover C )
{
   cover Result = 0;

   while ( |C| > 0 )
   {
      cover SubSet = LargestCubes( C );
      cube  Cube = MaxCompCube( C, SubSet );
      cover Comp = CompatibleCubes( C, Cube );
      Result = Result + Cube;
      C = Comp – Cube;
   }

   return Result;
}
```

Fig. 3. Computation of the largest unate subset of cubes.

Procedure MaxCompCube() selects one cube from the subset. Among all cubes in the subset, this cube has the maximum number of cubes in the cover that are pair-wise unate with it. For simplicity, pair-wise unate cubes are called *compatible cubes*.

Procedure CompatibleCubes() selects cubes that compatible with the given cube. The cube is added to the result, while the cover to be searched for a unate subset in the next iteration is reduced to the compatible cubes, from which the given cube is subtracted. The reason of this reduction is that the cubes that are not pair-wise unate with the selected cube cannot belong a unate subset containing the selected cube.

The weakest point of GetUnateSubset(), as described above, is that LargestCubes() may potentially return one cube for covers that happen to have only one cube with the smallest literal count. A natural improvement is to modify LargestCubes() in such a way that it returns a required number of cubes. These cubes include all the cubes with the smallest number of literals (m) and, if their number is less than the required number, the cubes with (m-1) literals, and so on, until the required number of cubes in found.

Procedure CompatibleCubeCount() is not shown in the pseudo-code because it is called inside MaxCompCube(). Its implementation is similar to CompatibleCubes(). The difference is that CompatibleCubes() returns the cubes pair-wise unate with the given one, while CompatibleCubeCount() only counts their quantity. Implementation of CompatibleCubes() is discussed in Section 5.

4 ZDD-Based Procedures

This section presents two specialized ZDD-based procedures for the manipulation of cube covers in the unate decomposition algorithm. Both procedures use cache to avoid redundant computation, but the cache lookup is omitted in the pseudo-code for the sake of clarity.

Selection of the Largest Cubes

This procedure takes the cover and returns the cubes that have the largest size, that is, the minimum literal count. For example, given the cover { abd, $\bar{b}c$, $\bar{a}de$ }, this procedure returns one cube { $\bar{b}c$ }; given the cover { bd, $\bar{b}c$, $\bar{a}de$ }, it returns two cubes { bd, $\bar{b}c$ }. The pseudo-code is in Fig. 4.

The terminal cases of LargestCubes() are simple. If the cover is empty, there are no largest cubes. If the cover contains only the tautology cube, it is returned as the one having the smallest number of literals. If it is not a terminal case, the cover is cofactored into three subcovers C0, C1, and C2 w.r.t. the top-most variable. Subcover C0 (C1) contains cubes with the top-most variable as a negative (positive) literal, while C2 contains cubes without this variable. Next, the problem is solved recursively for the subcovers. The resulting covers are considered to determine the one(s) with the minimum number of literals. (Function Lit() returns the number of literals in a random cube belonging to the cover). Only the cover(s) with the minimum number of literals contribute to the result, while the other ones are discarded (set to zero). Finally, the resulting cover is composed and returned.

```
cover LargestCubes( cover C )
{
   if ( C == 0 ) return 0;
   if ( C == 1 ) return 1;

   var x = TopVariable( C );
   (C0, C1, C2)= DecomposeCover( C, x );

   R0 = LargestCubes( C0 );
   R1 = LargestCubes( C1 );
   R2 = LargestCubes( C2 );

   int MinLit = min( Lit(R0),Lit(R1),Lit(R2) );
   if ( Lit(R0) > MinLit )  R0 = 0;
   if ( Lit(R1) > MinLit )  R1 = 0;
   if ( Lit(R2) > MinLit )  R2 = 0;

   return ComposeCover( R0, R1, R2, x );
}
```

Fig. 4. Computation of the largest cubes in the cover.

Selection of Pair-Wise Unate Cubes

This procedure performs computation of the cubes belonging to the cover that are pair-wise unate with the given cube. The pseudo-code of the procedure is given in Fig. 5.

The terminal cases of CompatibleCubes() are similar to the previous procedure. Next, the top-most variable in the cover and the cube are determined and the arguments are cofactored w.r.t. this variable. Notice that if a cube does not depend on the given variable, then its first two cofactors are empty, while the third cofactor is equal to the original cube.

If the top-most variable is absent in the cube, it means that all cubes in the cover can potentially contribute to the set of compatible cubes, if they are compatible with other variables in the cube. Therefore, there are three recursive calls to compute the components of the resulting cover. If the top-most variable in the cube is in the positive polarity, then the subcover with the negative literal is discarded because its cubes are not compatible with the cube. Consideration is similar if the top-most variable in the cube is in the negative polarity. Finally, the resulting cover is composed and returned.

```
cover CompatibleCubes( cover C, cube Q )
{
   if ( C == 0 ) return 0;
   if ( C == 1 ) return 1;
   assert( Q != 0 );

   var x = TopVariable( C, Q );
   (C0, C1, C2) = DecomposeCover( C, x );
   (Q0, Q1, Q2) = DecomposeCover( Q, x );

   if ( Q0 == 0 and Q1 == 0 )/*x is not in Q*/ {
        R0 = CompatibleCubes( C0, Q );
        R1 = CompatibleCubes( C1, Q );
        R2 = CompatibleCubes( C2, Q );
   }
   else if ( Q0 == 0 ) /*x is positive in Q*/ {
        R0 = 0;
        R1 = CompatibleCubes( C1, Q1 );
        R2 = CompatibleCubes( C2, Q1 );
   }
   else if ( Q1 == 0 ) /*x is negative in Q*/ {
        R0 = CompatibleCubes( C0, Q0 );
        R1 = 0;
        R2 = CompatibleCubes( C2, Q0 );
   }

   return ComposeCover( R0, R1, R2, x );
}
```

Fig. 5. Computation of cubes belonging to the cover that are pair-wise unate with the given cube.

5 Experimental results

The algorithm has been implemented in the program UN-DEC written in platform-independent C++ using the BDD/ZDD package “CUDD v. 2.3.0” [11]. The program has been tested on an HP J-6000 workstation under Unix operating system.

Table 1 shows the results of running UN-DEC on Espresso PLA benchmarks [3]. We considered all PLAs from the benchmark set with at least one output function having 10 or more cubes after Espresso minimization. The smaller PLAs were not considered because they are relatively trivial functions from a synthesis perspective.

The left part of Table 1 describes the PLA statistics followed by the results of unate decomposition, for the 63 (out of 88) PLAs, for which synthesis results improved. Columns “in” and “out” denote the number of inputs and outputs in each PLA. Columns “SOC” and “MOC” denote the number of cubes after running Espresso using the single-output and the multiple-output minimization modes respectively (in the single output mode, each function is minimized independently, and there is no sharing of product terms across different outputs). Column “S/M” gives the cube count ratio after single-output and multi-output minimization.

Column “UDC” denotes the number of cubes in the multi-level network (the sum total of cubes in each node) after unate decomposition is applied to the single-output minimized PLAs. Column “CL” lists the minimum unate cluster size (CubeLimit parameter) used to control the unate decomposition procedure. The algorithm stops when no unate cluster of size equal to or greater than CubeLimit is found. This limit was chosen in such a way that no function is decomposed into more than 7 blocks, because generating a large number of unate clusters can be counter-productive from the delay perspective as they increase the logic depth of the network.

Column “U/M” gives the cube count ratio after unate decomposition and after multi-output PLA minimization. A value less than 1 in the “U/M” column indicates that unate decomposition gave reduction in the number of cubes. A ratio greater than 1 indicates that there is significant product term sharing in the multiple output minimized PLA, which is not exploited by our algorithm, as it decomposes each output of the benchmark separately. Finally, column “T” gives the CPU time (in seconds) for unate decomposition.

The right part of Table 1 compares synthesis area, delay and CPU time for synthesis before and after unate decomposition. Synopsys Design Compiler was used in our experiments to structure and map each PLA (after Espresso multi-output minimization) as well as the unate decomposed network of these PLAs into a proprietary static CMOS cell library of Intel.

Columns “A” and “UDA give the area after technology mapping on the PLAs without and with unate decomposition respectively. Column “UDA/A”, gives the value of UDA/A, which ranges from 0.45 to 1.0 for various PLAs. Column “DelRatio” gives the ratio of the delay numbers of the mapped networks with and without unate decomposition. A value less than 1.0 indicates speed up after unate decomposition.

The CPU time for synthesis of the original PLAs as well as unate-decomposed PLAs (the latter time includes time taken for unate decomposition) is reported in columns “T” and UDT. The ratio UDT/T is reported in the last column. In the majority of cases, this ratio is less than 1, implying that the synthesis employing unate decomposition leads to savings in synthesis time as well.

Consider, for example, PLA ex1010 (row 29). This PLA has 10 inputs, 10 outputs, and 452 cubes after single output minimization using Espresso. The number of cubes after multi-output minimization (using Espresso) is 284 giving the ratio “S/M” equal to 1.59 indicating that a significant number of cubes are shared across multiple outputs. After unate decomposition, there are 453 cubes. This increase is possible because the algorithm uses the single-output minimized PLA as

the starting point and the heuristic SOP minimizer Espresso is invoked each time to minimize the unate components as well as the remaining binate component for each output function. The CPU time for the unate decomposition step was 2.5 seconds.

In the right part of Table 1, the area after synthesis and mapping of the original multi-output minimized PLA was 84198 while the unate decomposed network of this PLA was synthesized with area of 49736, giving the UDA/A ratio of 0.59 equivalent to 41 % savings in area. The delay of the original synthesized network and the unate-decomposed network was the same as implied by value 1 in "DelRatio" column. The CPU time needed to synthesize the original PLA and the unate-decomposed network was 1614 and 996.5 seconds respectively (the latter includes the unate decomposition time, which was only 2.5 seconds) yielding a 38% reduction in CPU time.

This benchmark gives a good example illustrating the benefits of unate decomposition, though in general, PLAs with high logic sharing across outputs (large "U/M" value) do not always yield such impressive results.

In summary, 88 PLAs from the Berkeley Espresso PLA benchmarks were considered. Unate decomposition improved the synthesis results for 63 PLAs (72 % of test cases). The improvement in area ranged from 0 to 55% with an average of **11% area reduction** with no significant delay degradation. The delay of the synthesized network after unate decomposition was better or within 2% of the original delay, with the average reduction in delay being 1.3%, considering the 63 PLAs reported in the table. Unate decomposition itself takes only a small fraction of the time compared to Synopsys DC.

It may be noted that most of these PLAs in Table 1 have significant sharing of product terms across multiple outputs as evidenced by the "S/M" ratio being greater than 1. Since unate decomposition was applied to each output independently, we did not take advantage of this logic sharing. We are currently working to extend our algorithm to take advantage of the product term sharing, which is likely to improve the synthesis results even further.

6 Conclusions

We presented a new approach to decomposition of incompletely specified multi-output functions into a netlist composed of unate functional blocks. The decomposition is based on implicit manipulation of cube covers using Zero-suppressed binary Decision Diagrams (ZDDs).

Our experimental results on PLA benchmarks show that the netlists decomposed into unate blocks lead to improved implementations using state-of-the-art logic synthesis tools.

The future work includes extending the algorithm in the following directions: (1) taking advantage of the shared logic in the multi-output benchmarks; (2) improving the quality of unate cube set selection; (3) implementing heuristics to automatically choose the largest size of the unate component.

References

[1] E. Sentovich, et al. "SIS: A System for Sequential Circuit Synthesis", *Tech. Rep. UCB/ERI, M92/41, ERL*, Dept. of EECS, Univ. of California, Berkeley, 1992.

[2] Synopsys Design Compiler. http://www.synopsys.com/products/logic/logic.html

[3] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Dordrecht, 1984.

[4] S. Minato. "Fast Generation of prime-irredundant covers from binary decision diagrams". *IEICE Trans. Fundamentals*, Vol. E76-A, #10, pp. 1721-1729, Oct., 1993.

[5] O. Coudert, J. C. Madre, H. Fraisse, H. Touati, "Implicit Prime Cover Computation: An Overview", in *Proc. of SASIMI '93*, Nara, Japan.

[6] S. Minato. "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems". *Proc. DAC '93*, pp. 272-277.

[7] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Comp*, Vol. C-35, No. 8 (August, 1986), pp. 677-691.

[8] O. Coudert. "Two-Level Logic Minimization: An Overview". *Integration*. Vol. 17, #2, Oct. 1994, pp. 97-140.

[9] S. Minato. "Fast Factorization Method for Implicit Cube Cover Representation". *IEEE Trans. CAD*, Vol. 15, No 4, April 1996, pp. 377-384.

[10] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer 1995.

[11] F. Somenzi. BDD package "CUDD v. 2.3.0.". http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[12] K. Nakamura, N. Tokura, T. Kasami, "Minimal Negative Gate Networks," *IEEETC C-21*, No.1, Jan. 1972, pp.5-11.

Table 1. PLA statistics, unate decomposition data and synthesis results for original and unate-decomposed PLAs.

No	Name	in	out	SOC	MOC	S/M	UDC	CL	U/M	T, s	A	UDA	UDA/A	DelRatio	T,s	UDT,s	UDT/ T
1	9sym	9	1	86	86	1.00	72	4	0.84	0.4	11788	5329	0.45	0.89	270	139.4	0.51
2	z5xp1	7	10	74	65	1.13	74	4	1.14	0.2	3670	3660	1.0	1.02	13	13.2	1.02
3	alu4	14	8	631	575	1.10	494	7	0.86	6.5	30762	19222	0.62	1.0	433	93.5	0.22
4	apex1	45	45	902	206	4.38	898	7	4.36	3.6	66024	64676	0.98	0.95	825	440.6	0.53
5	apex2	39	3	1065	1035	1.03	370	7	0.36	0.7	15500	9155	0.59	0.91	117	121.9	1.04
6	apex3	54	50	628	280	2.24	601	4	2.15	4.6	64977	61825	0.95	1.01	708	813.6	1.15
7	apex4	9	19	1004	436	2.30	988	5	2.27	2.0	71302	71022	1.0	1.0	641	599	0.93
8	b12	15	9	53	43	1.23	31	4	0.72	0.2	3245	2758	0.85	1.0	31	29.2	0.94
9	bw	5	28	110	22	5.0	103	4	4.68	0.5	6449	4292	0.67	1.0	18	13.5	0.75
10	clip	9	5	148	120	1.23	144	9	1.20	0.2	7413	7392	1.0	0.99	168	34.2	0.20
11	cordic	23	2	914	914	1.0	314	4	0.34	0.3	2354	1918	0.81	0.96	17	9.3	0.78
12	duke2	22	29	200	86	2.33	200	4	2.33	0.9	18424	16112	0.87	1.0	231	255.9	1.11
13	misex3	14	14	1232	690	1.79	482	9	0.70	3.6	54225	28222	0.52	0.95	818	528.6	0.65
14	rd53	5	3	31	31	1.0	31	4	1.00	0.1	1420	1410	0.99	1.02	6	6.1	1.02
15	rd73	7	3	141	127	1.11	141	4	1.11	0.76	6501	4749	0.73	0.94	75	57.2	0.76
16	sao2	10	4	73	58	1.26	54	4	0.93	0.1	7327	5785	0.80	0.99	126	122.1	0.97
17	seq	41	35	1399	336	4.16	1377	9	4.10	9.2	81752	80508	0.98	1.01	906	814.2	0.90
18	table5	17	15	550	158	3.48	550	5	3.48	10.1	57428	53468	0.93	0.99	611	756.1	0.81
19	alu2	10	8	73	68	1.07	53	4	0.78	0.2	2872	2468	0.86	0.94	9	22.2	2.47
20	alu3	10	8	68	66	1.03	57	7	0.86	0.2	2623	2499	0.95	0.99	9	8.2	0.91
21	amd	14	24	158	66	2.39	158	4	2.39	0.5	12742	12265	0.96	1.01	77	57.5	0.75
22	b10	15	11	171	100	1.71	166	6	1.66	0.5	16402	15271	0.93	0.96	246	305.5	1.2
23	bc0	26	11	487	179	2.72	454	4	2.54	5.2	36578	35541	0.97	0.99	400	225.5	0.56
24	bcb	26	39	542	155	3.50	545	5	3.52	10.1	54183	50170	0.93	1.02	874	613.1	0.70
25	chkn	29	7	140	140	1.0	141	7	1.0	1.5	16226	15967	0.98	1.0	206	329.5	1.57
26	cps	24	109	596	163	3.66	578	4	3.55	4.1	56143	53686	0.96	0.94	664	553.1	0.83
27	dc2	8	7	49	39	1.26	49	4	1.26	0.1	3784	3650	0.96	1.01	24	10.1	0.42
28	dist	8	5	152	123	1.24	147	4	1.20	0.3	12307	12016	0.98	1.01	177	189.3	1.07
29	ex1010	10	10	452	284	1.59	453	4	1.60	2.5	84198	49736	0.59	1.00	1614	996.5	0.62
30	ex5	8	63	304	74	4.11	165	4	2.23	1.0	10990	10554	0.96	1.01	45	46	1.02
31	ex7	16	5	119	119	1.0	85	4	0.71	0.3	5080	4479	0.88	0.99	174	13.3	0.08
32	exep	30	63	109	110	0.99	109	4	0.99	1.3	15427	14318	0.93	1.00	86	160.3	1.86
33	exps	8	38	512	136	3.76	509	4	0.99	1.2	33271	32960	0.99	1.00	213	236.2	1.11
34	gary	15	11	191	107	1.79	187	4	1.75	0.8	18134	17293	0.95	1.03	250	206.8	0.83
35	in3	35	29	214	74	2.89	207	4	2.80	0.9	12804	12120	0.95	0.98	120	41.9	0.35
36	in4	32	20	350	212	1.65	325	9	1.53	3.0	18113	16672	0.92	0.95	338	137.0	0.41
37	in5	24	14	175	62	2.82	175	4	2.82	0.8	13592	12400	0.91	0.99	139	83.8	0.60
38	in7	26	10	79	54	1.46	68	4	1.26	0.2	4935	4323	0.88	1.0	76	68.2	0.90
39	inc	7	9	44	32	1.38	44	4	1.38	0.2	3473	3214	0.93	1.0	13	12.2	1.0
40	intb	15	7	631	631	1.0	445	11	0.71	0.61	45049	26874	0.60	0.99	686	418.5	0.61
41	lin.rom	7	36	456	128	3.56	438	4	3.42	0.8	34878	34681	0.99	0.99	438	259.8	0.59
42	luc	8	27	158	27	5.85	156	6	5.78	0.5	7102	6729	0.95	1.02	26	24.5	0.94
43	m2	8	16	106	47	2.26	100	4	2.13	0.3	8294	7579	0.91	0.99	28	28.3	1.00
44	m3	8	16	133	66	2.02	127	4	1.92	0.3	12763	11498	0.90	0.99	231	178.3	0.77
45	m4	8	16	211	105	2.01	187	4	1.78	0.4	19513	18455	0.95	0.99	267	301.4	1.13
46	mainpla	27	54	2922	172	16.99	2901	7	16.87	28.9	92628	89093	0.96	1.0	674	756.9	1.12
47	max1024	10	6	328	274	1.20	301	4	1.10	1.8	36070	35168	0.97	0.94	455	519.8	1.14
48	max128	7	24	194	83	2.34	173	4	2.08	0.5	12919	11757	0.91	0.98	57	225.5	3.96
49	max512	9	6	167	145	1.15	155	5	1.07	0.3	15531	15013	0.97	0.98	396	79.3	0.20
50	mlp4	8	8	144	128	1.13	143	5	1.12	0.3	13686	13706	1.00	0.94	264	211.3	0.80
51	mp2d	14	14	76	31	2.45	49	4	1.58	0.3	2831	2779	0.98	1.02	13	13.3	1.02
52	newcond	11	2	31	31	1.0	31	4	1.0	0.1	2281	2208	0.97	1.0	39	49.1	0.79
53	opa	17	69	298	79	3.77	300	4	3.80	1.4	21482	19295	0.90	1.0	135	123.4	0.91
54	pdc	16	40	138	145	0.95	138	4	0.95	0.7	19824	9269	0.47	0.98	171	28.7	0.17
55	pope.rom	6	48	293	62	4.73	216	4	3.48	0.8	14805	14515	0.98	1.01	61	76.8	1.26
56	prom1	9	40	2067	472	4.38	1961	5	4.15	3.5	198839	192195	0.97	1.0	2946	3520.5	1.20
57	prom2	9	21	960	287	3.34	961	4	3.35	2.6	107257	90606	0.84	1.01	1953	1031.6	0.53
58	root	8	5	71	57	1.25	71	4	1.25	0.2	7361	5381	0.73	0.99	111	106.2	0.96
59	sqn	7	3	43	38	1.13	42	4	1.11	0.1	4147	3442	0.83	1.0	17	13.1	0.77
60	ti	47	72	512	213	2.40	510	4	2.39	2.2	29881	26781	0.90	1.01	293	137.2	0.47
61	X1dn	27	6	110	110	1.0	101	4	0.92	0.2	4997	4800	0.96	0.98	18	13.2	0.73
62	xparc	41	73	1531	254	6.03	1530	7	6.02	79.1	110016	108149	0.98	0.99	1175	1722.1	1.47
63	misg	56	23	75	69	1.09	35	4	0.51	0.4	2426	1462	0.61	0.79	20	7.4	0.37