# Introducing K-cuts and KL-cuts in Circuit Re-Mapping

**[1]Lucas Machado, [1]Osvaldo Martinello, [1]Renato Perez Ribas, [1]André Reis**
{lmachado, omjunior, rpribas, andreis}@inf.ufrgs.br,

**[1]PGMICRO, UFRGS, Porto Alegre, Brazil.**

## Abstract

*This paper introduces the concept of k-cuts and kl-cuts on top of a mapped circuit. These concepts are useful for local optimization of integrated circuit designs. Also, this paper compares the introduced cuts to the former And Inverter Graph (AIG) k-cuts and kl-cuts, pointing out the differences and advantagesthat motivate the use of kl-cuts on top of an already mapped netlist. An algorithm for computing the k-cuts and the kl-cuts on top of a mapped circuit is presented and applied to a combinational circuit example. The use of kl-cuts in a local optimization flow is shown.*

## 1. Introduction

Logic synthesis is an important area of study in the field of microelectronics, regarding the transformation of a design written in HDL (Hardware Description Language) to a netlist that instantiates logic gates of a given technology. Recent advances in logic synthesis are based on And-Inverter-Graphs – AIGs, for scalability reasons [1]-[2]. Part of these advances is based on the concept of *k-feasible cuts* [3]-[4], including algorithms for re-synthesis based on AIG rewriting [5]. Algorithms for efficient cut computation are well known for single output cuts. Particularly, algorithms for exhaustive computation of *k*-cuts were introduced by Cong [3] and Pan [4].

The concept of *kl-feasible cuts*on top of AIGs was introduced in[6]-[7], demonstrating an algorithm that is able to calculate *kl*-cuts on top of AIGs and discussing its potential use in local optimization [8], regularity extraction [9] and technology mapping. The use of *kl*-cuts in local optimization is justified as a cut of the circuit, potentially with multiple outputs, can be exchanged by another, taking into account all signals which it affects.

Martinello[6]-[7] demonstrated improvements in the number of multiple outputLUTs in FPGA mapped circuits. Multiple output LUTs are able to implement any function of up to *k* inputs and *l* outputs, as determined by the specific FPGA technology.

However, when it comes to local optimization over an ASIC flow, the use of AIGs does not address a very good correlation with the previous combinational circuit, since it keeps only the logical information. Information about buffering, drive strengths, area, delay, inverter chains, clock gating and arithmetic cells are simply lost in the AIG translation. This level of information opens possibilities to perform different approaches of local optimization, after a conventional logic synthesis flow, and the *kl*-cuts over a netlist can partition the circuit keeping this information.

This paper introduces the calculation of *kl*-cuts on top of mapped circuits and its use as a method for local optimization after the logic synthesis process, considering the standard cell IC flow. It focuses on the combinational part of the design. This method can take into account different methods of remapping, different remapping objectives and it tends to results in significant circuit improvement. Since *kl*-cuts are usually small combinational circuits, it is possible to apply a higher computational effort to improve the *kl*-cuts locally. Notice that applying computationally expensive algorithms to the entire design is not feasible, but it is feasible for *kl*-cuts.

This paper is organized as follows. Section 2 presents background knowledge and motivation for *kl*-cuts over netlist approach. The *k*-cuts and *kl*-cuts computation over mapped circuits are described in section 3. Section 4 will introduce how a local optimization flow can be done. Some comparison results with AIG are shown in Section 5and the final section concludes the paper.

## 2. Background

This section provides a review concerning AIGs, *k-feasible cuts*[3]-[4] and *kl-feasible cuts*[6]-[7] on top of AIG representation. Also, the netlist representation and the AIG representation are compared, and the motivation to use *kl*-cuts on top of a mapped circuit is discussed.

### 2.1. AIGs

An And-Inverter-Graph (AIG), $\mathcal{G}$, is a specific type of a Directed Acyclic Graph (DAG), where each node has either 0 incoming edges – the*primary inputs* (PI) – or 2 incoming edges – the AND nodes. Each edge can be negated or not. Some nodes are marked as *primary outputs* (PO).

### 2.2. K-cuts on AIGs

A cut of a node *n* is a set of nodes *c* such that every path between a PI and *n* contains a node in *c*. A cut of *n* is irredundant if no subset of it is a cut. A *k*-cut is an irredundant cut containing *k* or fewer nodes. Let A and B to be two sets of cuts. Let the auxiliary operation ⋈ to be:

$$A \bowtie B \equiv \{a \cup b \mid a \in A, b \in B, \mid a \cup b \mid < k\}$$

Let $\Phi_{\mathcal{K}}(n)$ to be the set of *k-feasible cuts* of $n \in \mathcal{G}$and if *n* is an AND node, let $n_1$ and $n_2$ to be its inputs. Then, $\Phi_{\mathcal{K}}(n)$ is defined recursively as follows:

$$\Phi_{\mathcal{K}}(n) \equiv \begin{cases} \{n\}, & : n \text{ is a PI} \\ \{n\} \cup \{\Phi_{\mathcal{K}}(n_1) \bowtie \Phi_{\mathcal{K}}(n_2)\} & : otherwise \end{cases}$$

The ⋈ operation can also easily remove the redundant cuts, by comparing the cuts with one another, or by making use of signatures [10].

### 2.3. KL-cuts on AIGs

The *k*-cuts are an efficient way of representing a region of an AIG regarding one output generation. However, when it comes to multiple output regions, multiple *k*-cuts would be needed. The *kl*-cuts introduced in [6,7] make use of multiple outputs to use a single cut for a given region of the AIG.

A *kl*-cut defines a sub-graph $\mathcal{G}_{kl}$ of $\mathcal{G}$which has no more than *k* inputs and no more than *l*outputs. It is represented as two sets of nodes $\{\mathcal{G}_k, \mathcal{G}_l\}$: being $\mathcal{G}_k$ the inputs set and $\mathcal{G}_l$ the outputs set.

If a node *n* belongs to a path between $n_k \in \mathcal{G}_k$and$n_l \in \mathcal{G}_l$, and$n \notin \mathcal{G}_k$, then *n* is contained in $\mathcal{G}_{kl}$. Notice that all nodes in $\mathcal{G}_l$ are contained in $\mathcal{G}_{kl}$. However, $\mathcal{G}_{kl}$does not contain any node of $\mathcal{G}_k$.

A KL-cut is said to be complete when all the following conditions are met:

- *Condition 1*: Every path between a PI and a node $n_l \in \mathcal{G}_l$ contains a node in$\mathcal{G}_k$;
- *Condition 2*: Every path between a node contained in $\mathcal{G}_{kl}$ and a PO contains a node in $\mathcal{G}_l$;
- *Condition 3*: No *kl*-cut defined by a subset of $\mathcal{G}_k$ and the same $\mathcal{G}_l$ is complete;
- *Condition 4*: No *kl*-cut defined by the same $\mathcal{G}_k$ and a subset of $\mathcal{G}_l$ is complete.

### 2.4. Netlists vs. AIGs

A mapped circuit,$\mathcal{C}$, is a specific type of a DAG, where each node has either 0 incoming edges – the *primary inputs* (PI) – or up to *m* incoming edges, where *m* is an integer value bigger or equal to 1, , defined by the gate with the largest number of inputs in the design – the logic gate nodes. Some nodes are marked as *primary outputs* (PO). The main differences between the AIG and netlist descriptions are: (1) the number of incoming edges, which is not limited by 2 in the netlist, and (2) the existence of inverters and buffers instead of simple negated or direct edges. Furthermore, it is possible to have inverter chains and buffering information in the mapped circuit. This information is suppressed in the AIG description.

Notice that it is possible to perform *k*-cuts and *kl*-cuts directly on top of netlists, given the similarities between an AIG and a netlist. This can be done extending the amount of inputs and handling with inverters and buffers. The algorithm and its particularities are shown in Section 3.

## 3. K-cuts and KL-cuts on Netlists

This section demonstrates how the computation of *k*-cuts and *kl*-cuts can be done in netlists, introducing the algorithms and demonstrating it through an example of a combinational circuit

### 3.1. K-cuts on Netlists

The difference of *k*-cuts in netlists and AIGs addresses the higher amount of inputs in the logic gate nodes and handles with single input logic gates (inverters and buffers).

Let $\Phi_{\mathcal{K}}(n)$ to be the set of *k*-cuts of$n \in \mathcal{C}$ and if *n* is a logic gate node, let$n_1, \dots, n_g$, with $g$ an integer value representing the number of inputs of the logic gate, $1 \leq g \leq m$, to be its inputs. Using the same operation ⋈ described in section 2, which is commutative, $\Phi_{\mathcal{K}}(n)$ is defined recursively as follows:

$$\Phi_{\mathcal{K}}(n) \equiv \begin{cases} \{n\}, & : n \text{ is a PI} \\ \Phi_{\mathcal{K}}(n_1), & : g = 1 \\ \{n\} \cup \{\Phi_{\mathcal{K}}(n_1) \bowtie \Phi_{\mathcal{K}}(n_2) \bowtie \cdots \bowtie \Phi_{\mathcal{K}}(n_g)\} & : otherwise \end{cases}$$

Fig. 1 shows a combinational circuit example. Calculating the *k*-cuts with *k* equals to 6 for this example, we obtain the values in Tab. 1.

Tab. 1 – The *k*-cuts for all nodes in the combinational circuit example

| **Node** | ***k*-cuts** | **Node** | ***k*-cuts** |
|---|---|---|---|
| a | {a} | wire0 | {a} |
| b | {b} | wire1 | {wire1}, {d, a} |
| c | {c} | wire2 | {a, b, c, d}, {a, b, c, wire1} |
| d | {d} | wire3 | {wire3}, {e, f, g, h} |
| e | {e} | o0 | {a, b, c, d}, {a, b, c, wire1} |
| f | {f} | o1 | {o1}, {a, wire3}, {a, e, f, g, h} |
| g | {g} | o2 | {o2}, {a, e, f, h} |
| h | {h} | | |

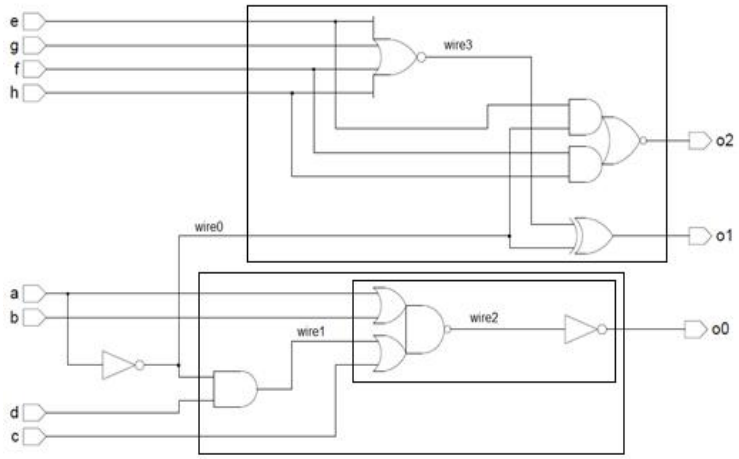



Fig. 1 – A combinational circuit example to demonstrate the *k*-cuts and *kl*-cuts computation.

## 3.2. KL-cuts on Netlists

The *kl*-cuts computation on netlists has different approaches compared AIGs. The *l* is defined as unbounded to discover all possible shared logic, keeping track of all outputs that depend on the same set of variables. It is possible to have several intermediate outputs in a *k*-cut that need to be covered in the *kl*-cut. Also, since the main focus is the local optimization, *kl*-cuts consisted of a single cell and *kl*-cuts with *k* equals to one (inverter and buffer chains) are discarded.

Fig. 2 shows a pseudo-code for *kl*-cuts enumeration on top of a mapped circuit. The algorithm receives as input the maximum *k*inputs to the *kl*-cuts and a combinational circuit. If the design has sequential elements, it is necessary to split the design into combinational and sequential parts.

```
01. compute_klcuts(k, circuit) {
02.   kcuts = compute_kcuts(circuit, k)
03.   for all kcut in kcuts {
04.     insts<- ø
05.     outputs<- ø
06.     for all node in kcut {
07.       addInsts(node, insts, outputs)
08.     }
09.     klcut = createKLcut(kcut, insts, outputs)
10.     klcuts.add(klcut)
11.   }
12.   returnklcuts
13. }
14. addInsts(node, insts, outputs) {
15.   ifKCutsOK(node) and node is not PO then {
16.     insts.add(node.inst)
17.     for all out in node.inst.outputs {
18.       addInsts(out, insts, outputs)
19.     }
20.   } else {
21.     outputs.add(node)
22.   }
```

Fig.2.Pseudo-code for *kl*-cuts calculation on top of a mapped netlist.

It starts enumerating all *k*-feasible cuts for all nodes in the circuit. All *k*-cuts of the circuit are grouped and used to find the *kl*-cuts. The grouped *k*-cuts for the Fig. 1 example would be: {d, a}, {a, b, c, d}, {a, b, c, wire1}, {e, f, g, h}, {a, wire3}, {a, e, f, g, h} and{a, e, f, h}.The function *addInsts()* traverses the circuit from the *k*-cut nodes to the outputs direction, saving the logic gate instances, the outputs and logic equation for each logic gate found. Each node is checked by the function *KCutsOK()*, which returns true if the node has at least one *k*-cut formed only by nodes in *kcut* (the current *k*-cut evaluated in the pseudo-code).

Further rework must be done to inverters in the inputs of the *kl*-cuts. If this kind of inverter is used only inside the *kl*-cut, there is no need to rework, since the *kl*-cut encapsulates the inverter. If the inverter is used somewhere else in the circuit, it is important to save this information in the *kl*-cut, in a way the re-mapper is able to reuse the already negated input (the negated variable has no extra cost in the remapping). Fig. 1 shows the three *kl*-cuts found in the combinational circuit example, with rectangles around the instances contained in each *kl*-cut.

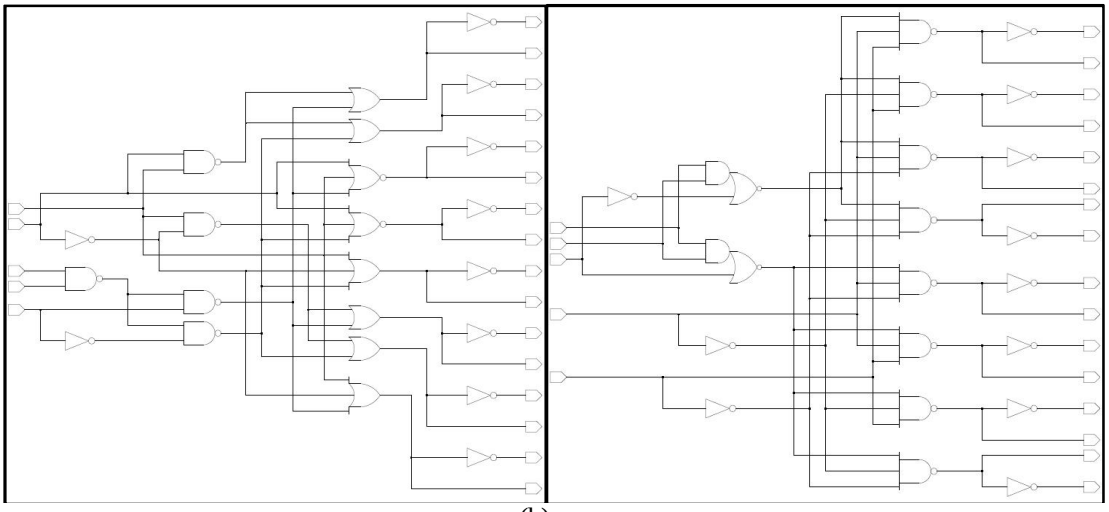
# 4. Local Optimization with KL-cuts

The local optimization flow using *kl*-cuts is based on a basic standard cell flow. After the conventional logic synthesis process, the output is a netlist containing all logic gates to implement the hardware described, using the cells of a given library. On top of this netlist, all *kl*-cuts are found, given a *k* maximum number of inputs.The amount of *kl*-cuts can be very high. One way to overcome this issue is to group the *kl*-cuts into P classes (inputs and outputs),decreasing greatly the quantity of different *kl*-cuts. This can be done through an extension of the work presented in [11].

For each *kl*-cut found (or grouped), it is possible to remap its output equations using more advanced techniques found in the literature, targeting improvement in area. For instance, the functional composition algorithm in [12] is able to identify shared logic between equations, providing the best logic trees for a *kl*-cut logic. Logic tree mapping techniques, as presented in [13], can result in better mapping for the given logic trees.

Besides the remapping methods, libraries with larger number of complex logic gates can be better explored using this local optimization approach.After the remapping, each *kl*-cut, that has a significant improvement in area compared to previous mapping, must be checked if timing constraints are still attained. It is necessary to match the input and output capacitances and compare the delays between the mapped and remapped *kl*-cut. It is possible to do sizing and buffering to keep the timing constraints met. If the *kl*-cut does not meet timing, it is discarded.

Fig.3a shows a *kl*-cut example found on top of a commercial benchmark mapped using a commercial library. Fig. 3b shows the same *kl*-cut after a remapping in which the timing constraints are still met, obtaining an area decrease of 20%.



(a) (b)

Fig.3.A *kl*-cut remapping example with the timing constraints still met and the area decreases in 20%

After all *kl*-cuts are remapped and checked timing, it is necessary to select the best *kl*-cuts subset to replace in the design, in a way they do not overlap. Two *kl*-cuts do not overlap if (1) their set of instances covered are disjoint and (2) their intermediate wires covered and input/output sets are disjoint. An intermediate wire of a *kl*-cut cannot be reused by another *kl*-cut after changing its internal logic.

The choice of the best subset of *kl*-cuts that do not overlap can be a very time-consuming task. The selection of a *kl*-cuts subset can be done by using a greedy algorithm, selecting the *kl*-cuts that give a higher improvement at first. This kind of choice algorithm does not give the best result, but it is much faster and feasible.

# 5. Results

Some initial results, with naive remapping implementations, already show up to 5% improvement in combinational area for commercial benchmarks. It is worth to notice that this decreasing in area occurs after all possible improvements given by a commercial logic synthesis tool.

Tab. 2 shows a comparison of the number of *kl*-cuts found on top of AIG and on top of the mapped circuit. All the experiments were done using the ISCAS benchmarks in the IWLS 2005 benchmarks set [14], with *k* equals to 6 and unbounded *l*.

In the results of Tab. 2, notice that there are approximately 70% more *kl*-cuts in the AIG than in the netlist. This can be easily explained by the purpose of each algorithm and the different data structures.

The *kl*-cuts on top of an AIG are created to cover the AIG functionality afterwards. Thus, there are *kl*-cuts with a single AND node and *kl*-cuts with more than one node that end up as a single cell in the netlist. The *kl*-cuts are created in the AIG version in a way the covering has more options to give a better AIG implementation, using those *kl*-cuts.

The *kl*-cuts on top of a netlist are created to be improved and then replaced in the original netlist. Hence, there are not *kl*-cuts with a single cell, for example. A single cell can have lots of AND nodes in an AIG representation, creating extra *kl*-cuts in the AIG version.

Tab. 2. Comparison of the number of *kl*-cuts found:AIG vs. Netlist.

| Benchmark | Netlist | AIG | Difference | Benchmark | Netlist | AIG | Difference |
|---|---|---|---|---|---|---|---|
| s27 | 36 | 38 | 5.26% | s838 | 1031 | 4097 | 74.84% |
| s208 | 104 | 444 | 76.58% | s832 | 1049 | 3498 | 70.01% |
| s420 | 348 | 1608 | 78.36% | s510 | 1094 | 2779 | 60.63% |
| s382 | 425 | 1217 | 65.08% | s15850 | 1726 | 4363 | 60.44% |
| s400 | 473 | 1047 | 54.82% | s1196 | 2373 | 7560 | 68.61% |
| s386 | 482 | 1328 | 63.70% | s13207 | 2406 | 7453 | 67.72% |
| s444 | 485 | 1150 | 57.83% | s1238 | 2592 | 8915 | 70.93% |
| s298 | 489 | 1187 | 58.80% | s1488 | 2698 | 8049 | 66.48% |
| s349 | 552 | 1516 | 63.59% | s1423 | 2880 | 7760 | 62.89% |
| s344 | 604 | 1855 | 67.44% | s1494 | 3129 | 8100 | 61.37% |
| s526 | 714 | 1930 | 63.01% | s9234 | 5920 | 16469 | 64.05% |
| s713 | 783 | 1823 | 57.05% | s5378 | 7688 | 22861 | 66.37% |
| s526n | 812 | 2028 | 59.96% | s35932 | 20113 | 63197 | 68.17% |
| s641 | 826 | 1840 | 55.11% | s38584 | 20265 | 96172 | 78.93% |
| s820 | 992 | 3323 | 70.15% | s38417 | 43499 | 195031 | 77.70% |

# 6. Conclusion

This paper presented a comparison of *k*-cuts and *kl*-cuts performed on top of mapped circuits as opposed to computing *k*-cuts and *kl*-cuts on top of AIG representations. The main differences lie on (1) the number of inputs for the 2-input AND nodes used on AIGs and the nodes of a gate netlist which may have several inputs, and (2) the existence of explicit inverters and buffers, appearing as nodes, in the netlist compared to the use of negated or direct edges used in the AIG. Also, we presented a method to perform *k*-cuts and *kl*-cuts on top of a netlist representation. We have also demonstrated that *kl*-cuts on top of mapped circuits can be very useful for local optimization of integrated circuit designs.

# 7. Acknowledgements

Research partially funded by Nangate Inc. under a Nangate/UFRGS research agreement, by CAPES and CNPq Brazilian funding agencies, by FAPERGS under grant 11/2053-9 (Pronem), and by the European Community's Seventh Framework Programme under grant 248538 – Synaptic.

# 8. References

[1] Ling, A. C., Zhu, J., “Scalable Synthesis and Clustering Techniques Using Decision Diagrams”, *IEEE Trans. on CAD*, 2008.

[2] Mishchenko, A., Brayton, R., “Scalable Logic Synthesis using a Simple Circuit Structure”, *Int’l Workshop on Logic & Synthesis*, 2006.

[3] Cong, J., Wu, C., Ding, Y., “Cut Ranking and Pruning: Enabling A General And Efficient FPGA Mapping Solution”, *Int’l Symp. on FPGA*, 1999.

[4] Pan, P., Lin C., “A New Retiming-based Technology Mapping Algorithm for LUT-based FPGAs”, *Int’l Symp. on FPGA*, 1998.

[5] Mishchenko, A., Chatterjee, S., Brayton, R., “DAG-aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis”, *Design Automation Conference*, 2006.

[6] MartinelloJr, O. and Marques, F.S. and Ribas, R.P. and Reis, A.I., “KL-cuts: a new approach for logic synthesis targeting multiple output blocks”, *Design Automation & Test in Europe*, 2010.

[7] MartinelloJr, O. and Marques, F.S. and Ribas, R.P. and Reis, A.I., “KL-cuts”, *Int’l Workshop on Logic & Synthesis*, 2009.

[8] Werber, J., Rautenbach, D., Szegedy, C., “Timing Optimization by Restructuring Long Combinatorial Paths”, *Int’l Conf. on CAD*, 2007.

[9] Rosiello, A. P. E., Ferrandi, F., Pandini, D., Sciuto, D., “A Hash-based Approach for Functional Regularity Extraction During Logic Synthesis”, *IEEE Comp. Soc. Annual Symp. on VLSI*, 2007.

[10] Mishchenko, A., Chatterjee, S., Brayton, R., “Improvements to Technology Mapping for LUT-Based FPGAs”, *Int’l Symp. on FPGA*, 2006.

[11] U. Hinsberger and R. Kolla, “Boolean matching for large libraries”, *Design Automation Conference*, 1998.

[12] Martins, M. G. A.; Rosa JR, L. S.; Rasmussen, A.B.; Ribas, R.P.; Reis, A. I., “Boolean Factoring with Multi-Objective Goals”, *Int’l Conference on Computer Design*, 2010.

[13] Correia, V.; Reis, A., "Advanced technology mapping for standard-cell generators", *Symposium on Integrated Circuits and Systems Design*, 2004.

[14] http://iwls.org/iwls2005/benchmarks.html.