Technology Mapping into General Programmable Cells

Alan Mishchenko Robert Brayton
Department of EECS, University of California, Berkeley
{alanmi, brayton}@berkeley.edu

Wenyi Feng Jonathan Greene
Microsemi Corporation SOC Products Group
{wenyi.feng, jonathan.greene}@microsemi.com

ABSTRACT

Field-Programmable Gate Arrays (FPGA) implement logic functions using programmable cells, such as K-input lookup-tables (K-LUTs). A K-LUT can implement any Boolean function with K inputs and one output. Methods for mapping into K-LUTs are extensively researched and widely used. Recently, cells other than K-LUTs have been explored, for example, those composed of several LUTs and those combining LUTs with several gates. Known methods for mapping into these cells are specialized and complicated, requiring a substantial effort to evaluate custom cell architectures. This paper presents a general approach to efficiently map into single-output K-input cells containing LUTs, MUXes, and other elementary gates. Cells with to 16 inputs can be handled. The mapper is fully automated and takes a logic network and a symbolic description of a programmable cell, and produces an optimized network composed of instances of the given cell. Past work on delay/area optimization during mapping is applicable and leads to good quality of results.

1. INTRODUCTION

Technology mapping for traditional FPGAs transforms a design into a network of K-input LUTs [7]. Since a K-LUT can implement any Boolean function of up to K inputs, mapping into LUTs can be structural, without any functional matching needed for standard cells and programmable cells.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids – Optimization;
B.7.1 [**Integrated Circuits**]: Types and Design Styles – Gate arrays

General Terms

Algorithms, Performance, Design, Experimentation, Modelling

Keywords

FPGA, technology mapping, programmable cells, Boolean matching, Boolean function

FPGA'15, February 22–24, 2015, Monterey, Califirnia, USA.

ACM 978-1-4503-3315-3/15/02…$15.00.
http://dx.doi.org/10.1145/2684746.2689082

Novel FPGA architectures based on programmable cells have been proposed. For example, LUT structures combine two or more LUTs in one cell [5][17]. Other extensions include cells based on LUTs and AND gates [1], LUTs and MUXes [4][16], and cones of elementary gates [20].

A common difficulty in evaluating any such architecture is the need to develop a dedicated mapper, or at least modify a traditional LUT mapper often using ad hoc and suboptimal methods. This task is difficult because matching into the given cell often requires sophisticated programming to fairly evaluate a proposed architecture. As a result, custom mappers are often inflexible (a change of the cell structure may lead to a non-trivial redesign of the mapper) and slow (in our experience, a 10x slow-down is common and expected when a modified LUT mapper performs on the fly matching against the cell).

To facilitate research in FPGA architecture evaluation, a general technology mapper ideally takes any design and programmable cell and produces a mapped network composed of multiple instances of the given cell, each with its own configuration parameters. These specify an assignment of variables of the original function to the cell inputs, and bits used to program LUTs present in the cell, so that the cell can realize a given Boolean function.

The present paper answers this need by proposing a general mapper into K-input programmable cells, where K can be up to 16, although computation is more efficient when K does not exceed 12, covering many of the practically interesting cell architectures.

The proposed mapper does not require manual tuning, other than providing a description of the cell. The time-consuming Boolean matching is replaced by a pre-computation, which can be carried out concurrently, reducing this one-time preparation from hours to minutes.

Finally, the quality of results produced by the mapper is on par with that of results produced by state-of-the-art LUT mappers, because the same mapping heuristics are used.

The paper is organized as follows. Section 2 reviews background. Section 3 shows a way to characterize a general programmable cell. Section 4 describes the process of pre-computing Boolean functions to be matched against the cell. Section 5 describes the Boolean matching. Section 6 describes modifications to a LUT mapper needed for mapping into programmable cells. Experimental results are described in Section 7, while Section 8 concludes the paper.

2. BACKGROUND

2.1 Boolean network

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

A node *n* has 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 fanin (fanout) cone of node *n* is a subset of nodes of the network, reachable through the fanin (fanout) edges of the node.

2.2 And-Inverter Graph

And-Inverter Graph (AIG) is a Boolean network whose nodes can be classified as follows:

- One constant 0 node.
- Combinational inputs (primary inputs, flop outputs).
- Internal two-input AND nodes.
- Combinational outputs (primary outputs, flop inputs).

The fanins of internal AND nodes and combinational outputs can be complemented. The complemented attribute is represented as a bit mark on a fanin edge, rather than a single-input node. Due to their compactness and homogeneity, AIGs have become a de-facto standard for representing circuits in technology mappers.

2.3 Structural cut

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 is the node itself. A cut is *K-feasible* if the number of leaves does not exceed *K*. A local function of a node *n*, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in *n* and expressed in terms of the leaves, *x*, of a cut of *n*.

Cut enumeration [15] is a technique used by a cut-based technology mapper to compute cuts using dynamic programming, starting from PIs and ending at POs.

2.4 Boolean function

Let $f(X)$: $B^n \rightarrow B$, $B = \{0,1\}$, be a completely specified Boolean function, or *function* for short. The *support* of *f, supp*(*f*), is the set of variables *X* influencing the output value of *f*. The support size is denoted by $|X|$.

Two functions are NPN-equivalent if one of them can be obtained from the other by negation (N) and permutation (P) of the inputs and outputs. Consider the set of all Boolean functions derived from a given function *F* by a sequence of these transforms. These functions constitute the *NPN class* of function *F*. The *NPN canonical form* of function *F* is one function belonging to its NPN class, also called the *representative* of this class. Selection of the representative is algorithm-specific. For example, in some cases, the representation is the function whose truth table has minimum (or maximum) integer value among all the truth tables of functions belonging to the NPN class.

3. CELL DESCRIPTION

It is assumed that a programmable cell is composed of LUTs, MUXes and the elementary gates, AND and XOR. Other gates types can be expressed using these primitives.

The proposed matcher takes a character string expressed using the notation from [13][14]: parentheses represent an AND, square braces represent an XOR, angular braces represent a 2:1-MUX, curly braces represent a LUT, and an exclamation mark is NOT. For example, (*abc*) is AND(*a, b, c*), and <*abc*> is MUX(*a, b, c*) = *ab* + !*ac*. For example, a 7-input cell composed of a 6-LUT feeding a 2-input AND [1] is represented as: *h*={*abcdef*};*i*=(*gh*). The lower-case characters (*a, b, c*, etc) are reserved for primary inputs. Internal variables (in this case, *h* and *i*) follow without gaps in the alphabetical order. Spaces are disallowed in the description.

4. HARVESTING FUNCTIONS

During mapping into K-input cells, Boolean functions considered by an AIG-based mapper are those appearing as functions of K-input cuts in the AIG. Industrial designs often contain different types of logic and may vary greatly in terms of the functions found at their structural cuts. The matcher is applied to only those functions appearing at some cut in the design. Such functions can be collected and stored for future use.

An efficient method [19] to pre-compute Boolean functions of cuts in a design, or a suite of designs relies on fast algorithms to compute NPN-canonical forms [9] and compactly store them in a data-structure called the DSD manager [13][14]. The manager stores representatives of each NPN class as a shared tree, providing a convenient way of checking functional properties, such as symmetry, unateness, decomposability, etc.

5. BOOLEAN MATCHING

SAT-based evaluation of programmable cells was introduced in [11] and further developed in [6][8]. Our implementation uses a dedicated Quantified Boolean Formula (QBF) solver, which performs iterative counterexample-guided refinement [18][10].

In this case, the problem of matching a function $F(x)$ with a cell $C(x, p)$, is equivalent to checking satisfiability of the formula: $\exists p \forall x[C(x, p) == F(x)]$. In addition, if a satisfying solution exists, it shows how to configure the cell using parameters *p* to realize function $F(x)$.

The iterative approach finds one SAT assignment, (x_0, p_0), of the formula $C(x, p) == F(x)$. For the given values of p_0, if $\forall x[C_{p0}(x) == F(x)]$ holds, then a solution, p_0, is found. Otherwise, x_0 is substituted into $C(x, p)$, and the resulting function, $C_{x0}(p)$, is used as an additional constraint for $\exists p[C(x, p) == F(x)]$. If at some point the formula is UNSAT, the QBF instance has no solution and the match does not exist. An upper bound on iterations required is $2^{|x|}$, but in practice it converges much faster. Additional speedup can be achieved by adding symmetry-breaking clauses and solving multiple QBF instances concurrently.

6. TECHNOLOGY MAPPING

We modified the priority-cut-based technology mapper [12] to enable processing K-input cuts whose functions are pre-computed and pre-matched. The mapper has access to the DSD manager storing each NPN class of Boolean functions appearing in the design along with its matching status (matchable/unmatchable) as well as the configuration parameters for the matchable functions.

In a typical LUT mapper, cuts are computed along with their truth tables using topological cut enumeration [15]. The cuts are then support-minimized by removing variables appearing in the structural support but not in the Boolean functions of the cut. NPN classes of the cut functions are computed [9]. These steps are performed as usual. Our modifications to the mapper concern handling of cuts whose functions have no match with the given cell. Such cuts are labeled and not allowed to be selected as best cuts. When prioritizing cuts, preference is given to matchable cuts, and if there is room left, some of the labeled cuts are stored and used to compute cuts for the fanouts. It is possible that, by merging two unmatchable cuts, a matchable cut is produced.

When the final mapping is derived, a subset of best cuts is selected and returned to the user as the final mapping. Since the best cuts are always matchable, the resulting mapping only contains the cuts that can be expressed using the target cell. At this point, the configuration parameters are retrieved and used to output the set of configured cells representing the given design. Optionally, a functional equivalence check can be performed to make sure that the network of configured programmable cells has the same functionality as the original Boolean network.

7. EXPERIMENTAL RESULTS

The proposed framework is implemented in ABC [2]. For a case-study, we a suite of 10 representative designs whose sizes are between 10K and 90K 6-LUTs.

7.1 Programmable cells

Figure 1 shows two programmable cells, which contain three 3-LUTs and two 2-ANDs. Our experiments indicate that the additional 2-LUT present in Cell B substantially increases its expressive power at the cost of only four additional configuration bits. Selecting these two cells is somewhat arbitrary but it allows us to illustrate the proposed method with Cell B having reconvergent paths, which is hard to handle using traditional Boolean methods.

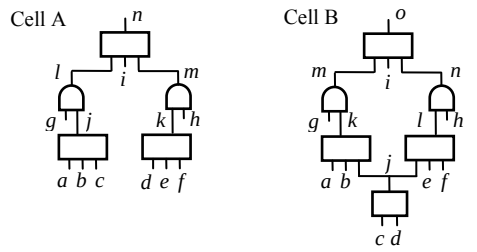

Figure 1. Programmable cells used in this paper.

Description of Cell A: *j={abc};k={def};l=(gj);m=(kh);n={lim};AB;BC;DE;EF;GH*. Description of Cell B: *j={cd};k={abj};l={jef};m=(gk);n=(lh);o={min};AB;CD;EF;GH.* In both cases, an optional list of primary input symmetries is appended, which makes the SAT solver 2-4x faster. The lower-case characters are inputs and internal nodes. The upper-case characters are used for symmetries. (With some extra effort, symmetries can be computed automatically.)

7.2 Harvesting Boolean functions

As the first step, NPN classes of 9-input functions appearing in the selected benchmarks are pre-computed. Any logic synthesis script can be used for this task. In our experiments, the script (*&synch2; &if -n –K 9*) was iterated three times for each benchmark. This script performs logic synthesis with choices (*&synch2*), computes 9-input cuts together with their Boolean functions and saves them in the DSD manager (switch '*-n*'). The computation took about 20 minutes and the resulting manager contained about one million NPN classes together with their occurrence counters. The manager is saved into a file as follows: *dsd_save funcs9.dsd*. The size of the resulting file is 41MB.

All functions that are *not* collected by the script but appear in a design during mapping will be treated as unmatchable by the mapper. This limitation was addressed, but the description is outside of the scope of this paper, because ignoring a small fraction of complex functions, in our experience, does not impact the quality of results. This is why the next optional step is to filter out rare NPN classes. Such classes rarely appear during mapping; moreover, it is unlikely that they can be matched with the cell. Filtering them out tends to preserve quality and reduce runtime.

In our experiments, NPN classes appearing in the designs less than 10 times are removed: *dsd_load funcs9.dsd; dsd_filter -L 10; dsd_save funcs9filter.dsd.* The resulting file contains 130K classes and occupies only 4MB. We tried using unfiltered NPN classes, leading to a negligible (less than 1%) degradation of area/delay.

7.3 Boolean matching

Matching was performed by the following command: *dsd_load funcs9filter.dsd; set progressbar; dsd_match –S "<cell_description>" -P 30; dsd_save funcs9match.dsd; dsd_ps.* The last column printed by *dsd_ps* shows the number/percentage of classes unmatchable with the cell.

The runtime of concurrent matching (*dsd_match*) in our experiments was 450 sec for Cell A (1750 sec for Cell B) on a computer with 16 hyper-threaded cores. The argument "*-P 30*" limits the number of concurrent worker threads to 30, not counting the controller thread. The same computation for Cell A runs 8410 sec on one thread, which is 18.7x slower than the concurrent one.

7.4 Technology mapping

Given a pre-matched library of NPN classes of functions appearing in the sample designs, mapping can be performed using command *&if -k* after reading the library as follows: *dsd_load funcs9match.dsd.* Optionally, a custom LUT library (command *read_lut*) can be used to specify the LUT-size-specific area/delay.

In our experiment, we iterated the following script three times (*&synch2; &if –k –K 9*). Each node in the resulting netlists does not exceed 9 inputs and can be realized by the given programmable cell. Currently, the configuration parameters for the nodes are computed but not used. In general, a hierarchical mapped netlist can be produced where instances contain the logic of each cell defined by its configuration parameters.

The result of mapping into programmable cells is compared against mapping into traditional K-LUTs ($6 \le K \le 9$) produced by command *&if* in ABC [2]. Three iterations of the script (*&synch2; &if –m –K <num>*) were used with switch '*-m*' forcing truth table computation and cut minimization because these steps are required for mapping into programmable cells. In all cases, the results of mapping were verified using command *&cec*.

7.5 Summary of experiment

The experimental results are summarized in Table 1. Cell B outperforms Cell A in terms of both area and delay measured in terms of the number of cells and the number of cell levels. Area produced using Cell B is close to that for 9-LUTs, even though Cell B has only 28 configuration bits and two extra AND-gates, compared to the 512 bits needed for a 9-LUT! Delay produced by Cell B is between delays produced using 8-LUTs and 9-LUTs.

Table 1 shows that the synthesis flow based on Cell B is 48% slower than the flow based on the traditional 6-LUTs and only 20% slower than the flow based on 9-LUTs.

The good performance of Cell B motivates research into programmable cells containing reconvergent logic structure.

Another way to improve expressive power of the cells is to allow for constants and inverters at the free inputs of the AND-gates. For this, "*g*" and "*h*" in the cell description can be replaced by "{*g*}" and "{*h*}", respectively, where curly braces represent a 1-input LUT. Yet another way to boost the expressive power, is to replace 2-ANDs with 2-LUTs. In both cases, the matching and mapping stages of the flow can be repeated, resulting in more matches and improved quality of mapping, while the runtime of matching may degrade due to the increased cell complexity.

8. CONCLUSIONS

The paper presents an integrated approach to map logic netlists into arbitrary single-output programmable cells specified by the user. The approach is based on pre-computing typical Boolean functions appearing in a set of sample netlists and pre-matching them against the given cell. An available technology mapper is minimally modified to make use of the matching information. Because of the pre-computation, the runtime of the mapper is

reasonable. The quality of results is good because the same efficient heuristics are used as during mapping into K-LUTs.

9. ACKNOWLEDGEMENTS

This work is supported in part by SRC contract 1875.001 and NSA grant ”Enhanced equivalence checking in crypto-analytic applications”. We also thank industrial sponsors of BVSRC: Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Mentor Graphics, Microsemi, Real Intent, Synopsys, Tabula, and Verific for their continued support.

10. REFERENCES

[1] J. H. Anderson, Q. Wang, "Area-efficient FPGA logic elements: architecture and synthesis," *Proc. ASP-DAC’11*, pp. 369-375. http://janders.eecg.toronto.edu/pdfs/aspdac_2011.pdf

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

[3] V. Bertacco and M. Damiani, "Disjunctive decomposition of logic functions," *Proc. ICCAD ‘97*, pp. 78-82.

[4] S. Chin, J.H. Anderson, “A case for hardened multiplexers in FPGAs,” *Proc. ICFPT’13, pp. 42-49*. http://janders.eecg.toronto.edu/ pdfs/xan.pdf

[5] J. Cong and Y. Hwang, “Boolean matching for LUT-based logic blocks with applications to architecture evaluation and technology mapping,” *IEEE TCAD’01*, Vol. 20(9), pp. 1077-1090.

[6] J. Cong and K. Minkovich, “Improved SAT-based Boolean matching using implicants for LUT-based FPGAs”, *Proc. FPGA’07*. http://cadlab.cs.ucla.edu/~kirill/fpga07.pdf

[7] R. J. Francis, J. Rose, and K. Chung, ”Chortle: A technology mapping program for lookup table-based field programmable gate arrays”, *Proc. DAC ’90*, pp. 613-619.

[8] Y. Hu, V. Shih, R. Majumdar, and L. He. “Efficient SAT-based Boolean matching for heterogeneous FPGA technology mapping”, *Proc. ICCAD’07*.

[9] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification", *Proc. ICFPT'13*. http://www.eecs.berkeley.edu/~alanmi/publications/2013/icfpt13_npn.pdf

[10] M. Janota, W. Klieber, J. Marques-Silva, and E. Clarke, “Solving QBF with counterexample-guided refinement”, *Proc. SAT’12*. https://www.cs.cmu.edu/~wklieber/papers/qbf-cegar-sat-2012.pdf

[11] A. Ling, D. Singh, and S. Brown. “FPGA PLB evaluation using Quantified Boolean Satisfiability”, *Proc. FPGA’05*. http://www.eecg.toronto.edu/~brown/papers/fpl05-ling.pdf

[12] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*, pp. 354-361.

[13] A. Mishchenko and R. Brayton, "Faster logic manipulation for large designs", *Proc. IWLS'13*. http://www.eecs.berkeley.edu/ ~alanmi/publications/2013/iwls13_dsd.pdf

[14] A. Mishchenko, "Enumeration of irredundant circuit structures", *Proc. IWLS'14*. http://www.eecs.berkeley.edu/~alanmi/publications/2014/iwls14_dsd.pdf

[15] P. Pan and C.-C. Lin, “A new retiming-based technology mapping algorithm for LUT-based FPGAs”, *Proc. FPGA ’98*, pp. 35-42.

[16] M. Purnaprajna and P. Ienne, “A case for heterogeneous technology-mapping: soft vs hard multiplexers”. *Proc. FCCM’13*, pp. 53-56.

[17] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures", *Proc. DATE'12*.

[18] A. Solar-Lezama, Ch. G. Jones, and R. Bodik, "Sketching concurrent datastructures", *Proc. PLDI '08*. http://people.csail.mit. edu/asolar/papers/pldi207_SketchingConcurrency.pdf

[19] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis", *Proc. ICCAD'12*, pp. 597-604. http://www.eecs.berkeley.edu/~alanmi/publications/2012/iccad12_lms.pdf

[20] G. Zgheib, L. Yang, Z. Huang, D. Novo, H. Parandeh-Afshar, H. Yang, and P. Ienne, “Revisiting and-inverter cones”. *Proc. FPGA’14*, pp. 45-54. http://lap.epfl.ch/files/content/ sites/lap/files/shared/publications/ZgheibFeb14_RevisitingAndInverterCones_FPGA14.pdf

Table 1. Comparing traditional K-LUT mapping with mapping using the K-input programmable cells

Design	Area (number of instances)						Delay (logic depth in terms of instances)						Runtime (seconds)			
	6-LUT	7-LUT	8-LUT	9-LUT	Cell A	Cell B	6-LUT	7-LUT	8-LUT	9-LUT	Cell A	Cell B	6-LUT	9-LUT	Cell A	Cell B
01	31246	28197	25734	24101	32408	25284	19	16	14	14	15	14	216	243	262	271
02	19808	19428	18998	18439	21694	18503	5	5	5	4	5	4	11	14	14	15
03	25528	23042	21314	20310	29071	20766	12	11	10	9	14	10	59	80	97	97
04	39366	36414	37315	35384	39548	33357	9	8	7	6	8	7	75	93	99	103
05	44426	41609	38232	36036	47810	36679	9	8	7	7	8	7	102	134	145	159
06	88964	83504	76317	72346	94669	72835	9	8	7	7	8	7	205	272	286	310
07	31048	27950	25042	24257	31769	24893	8	8	7	6	7	7	60	78	86	94
08	33154	29074	25268	26008	37626	26634	19	15	15	13	18	14	58	86	82	96
09	32684	32091	31485	31164	34709	31355	5	4	4	3	4	3	24	28	28	30
10	12909	12286	11688	11554	13165	11916	7	6	6	4	7	5	11	14	18	20
Geomean	1.000	0.929	0.870	0.841	1.065	0.852	1.000	0.885	0.820	0.707	0.919	0.759	1.000	1.282	1.381	1.487