Generic Logic Synthesis meets RTL Synthesis

Heinz Riener[1] Mathias Soeken[2] Eleonora Testa[1] Giovanni De Micheli[1]
[1]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
[2]Microsoft, Switzerland

***Abstract*—We present an integration of Generic Logic Synthesis, a recent methodology for developing logic synthesis algorithms that are independent of a specific technology-independent logic representation, in an RTL synthesis flow. This integration allows us to choose different multi-level logic representations during the synthesis process and judge the impact of this choice on the overall synthesis result. We propose a prototypical implementation that combines the open-source RTL synthesis framework *Yosys* with the EPFL logic synthesis library *mockturtle*. In an experimental evaluation, we show the synthesis results for different arithmetic and cryptographic benchmarks from OpenCores.org and for a hand-crafted modular multiplier using four different multi-level logic representations in technology-independent logic optimization.**

I. Introduction

Generic Logic Synthesis [1] is a recent methodology in technology-independent logic synthesis for developing algorithms that can be generically applied to different (multi-level) logic representations. As a consequence, this methodology enables a fast realization of complex synthesis flows for tailored logic representations such as *majority-inverter graphs* (MIG) or *XOR-AND graphs* (XAG) without the need for replicating and adapting large portions of source code.

Each logic synthesis algorithm is parameterized with a logic representation and generically implemented using a set of naming conventions with agreed semantics defined by a common *network interface API*. Logic representations then have to implement (a subset of) the network interface API. When a generic logic synthesis algorithm is instantiated with a concrete logic representation, static checking ensures at compile-time that the logic representation implements all methods of the network interface API required by the logic synthesis algorithm. If one (or more) methods of the network interface API are missing, or if the interface is not correctly implemented, a compile-time error is reported. Otherwise, if compilation succeeds, a highly-tailored (and optimized) logic synthesis algorithm for the concrete logic representation at hand is generated.

A prototypical implementation of Generic Logic Synthesis [1] has been presented for the scalable peephole synthesis framework introduced by Mishchenko and Brayton [2]. The implementation is publicly available in the EPFL logic synthesis library *mockturtle* [3].

In this paper, we demonstrate how the *mockturtle* library can be integrated into an RTL synthesis flow and show the impact of the choice of a multi-level logic representation in RTL synthesis after LUT mapping.

II. Logic synthesis integrated in RTL Synthesis

A. Synthesis and verification flow

In this section, we illustrate the overall RTL synthesis flow by means of a running example. In the example, we synthesize a multiplier module provided in the hardware description language Verilog.

```
module top(input clk, input[7:0] a,b, output reg[15:0] c);
  always @(posedge clk) c <= a * b;
endmodule // top
```

Synthesis. As synthesis suite, we use the open-source synthesis framework Yosys [4]. Yosys comes with a shell interface which allows sessions like the following:

```
yosys> read_verilog file.v
yosys> prep
yosys> techmap
yosys> cirkit -script optimize.cs
yosys> flatten
yosys> write_verilog file_optimized.v
```

In the session above, a conservative RTL synthesis flow (‘prep’) followed by a technology mapping step (‘techmap’) and a flattening step (‘flatten’) are carried out on the Verilog file.

We have implemented a new command ‘cirkit’ into Yosys that enables us to run logic optimization scripts composed of optimizing transformations provided by the mockturtle library.[1] Each combinational part of the technology-mapped implementation is extracted and logically optimized using the script ‘optimize.cs’ (‘cirkit’). Input and output of the ‘cirkit’ command are provided in form of LUT networks. The logically optimized combinational parts are finally re-composed, flattened into one gate level netlist, and the optimized gate level netlist is written into a new Verilog file.

Fig. 1 depicts the overall interaction of the RTL synthesis engine with the logic synthesis framework.

Verification. The correctness of the optimizing transformations carried out by the proposed flow can be verified in two stages:

1) *Combinational equivalence checking (CEC)*: We prove that LUT networks provided as input and output to CirKit are functionally equivalent using combinational equivalence checking implemented in ABC [5].
2) *Weak sequential equivalence checking (SEC)*: We prove that the outputs of the initial RTL design and the obtained gate level netlist do not diverge for a fixed

[1]The modified Yosys synthesis suite can be found online: *https://github.com/hriener/yosys/*

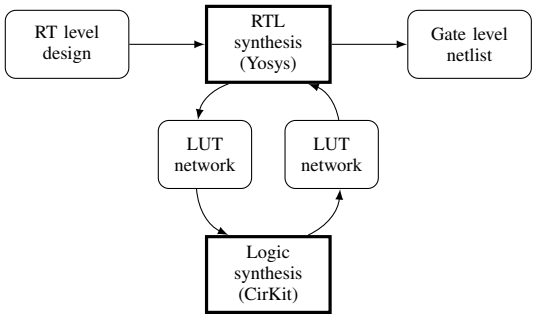

Fig. 1. RTL synthesis with integrated logic optimization

number of time steps when fed with same inputs using sequential equivalence checking implemented in Yosys.

B. Optimization scripts

CirKit[2] is a front-end for the mockturtle library that instantiates the implemented algorithms and provides a shell interface for executing sequences of optimizing transformations. In the following, we focus on the subset of the supported commands used to interface with the RTL synthesis suite Yosys. A typical CirKit optimization script for this purpose looks as follows:

```
cirkit> read_blif <TMP_DIR>/input.blif
cirkit> lut_resynthesis --mig
cirkit> cut_rewrite --mig
cirkit> lut_mapping
cirkit> collapse_mapping
cirkit> write_blif <TMP_DIR>/output.blif
```

First, the LUT network is read (‘read_blif’). This LUT network is composed of LUTs with potentially large number of fan-ins. The goal of LUT resynthesis (‘lut_resynthesis’) is to decompose these LUTs into primitive gates of a homogeneous gate library to express the LUT network in form of a multi-level logic representation. The additional parameter ‘--mig’ specifies that the gate library of MIGs should be used, which consists only of majority-of-three (MAJ-3) gates and inverters. Multi-level logic optimization algorithms, such as Boolean rewriting [6] (‘cut_rewrite’) are then carried out to optimize the logic representation with respect to a cost function. Finally, the optimized multi-level logic representation is mapped back into a LUT network (‘lut_mapping’ and ‘collapse_mapping’). The obtained LUT network is written into a file (‘write_blif’) to be read by Yosys.

In practice, the first and last command (‘read_blif’ and ‘write_blif’) of a CirKit optimization script are automatically generated by Yosys and need not be part of the optimization script.

C. LUT resynthesis

LUT resynthesis is an algorithm that translates a LUT network into an arbitrary gate-based network. Each LUT is synthesized into subnetwork of the targeted gate-based network type, based on the LUT’s truth table. The subnetworks are composed according to the LUT structure of the original LUT network.

The flexibility lies in the underlying synthesis algorithm. It is important to note, that for LUT resynthesis the synthesis algorithm must find a subnetwork for all LUT functions f in the LUT network. Possible synthesis algorithms are Shannon decomposition, database lookup, bi-decomposition, exact synthesis [7], or DSD decomposition.

D. Cut rewriting

Cut rewriting is based on the idea to improve a logic network by changing the gate-level structure of a subcircuit. Subcircuits are enumerated using cut enumeration [8]. For each cut, several alternative gate-level structures are computed, and the effect of substituting the structure for the current one is evaluated. Note that not all gates may be removed when removing the current structure, because they are required by other gates outside of the cut. Similarly, the new gate-level structure to be inserted in the logic network may make use of already existing logic. Analysing the impact and gain in this setting is referred to as DAG-aware rewriting [9].

Cut rewriting is implemented as a generic algorithm in *mockturtle*, where the synthesis algorithm to compute alternative gate-level structures for a cut is passed as a parameter. In fact, the same synthesis algorithms that are used in LUT resynthesis can be used for cut rewriting, which is another manifestation of generic logic synthesis in *mockturtle*.

E. LUT mapping

LUT mapping (see, e.g., [10]) addresses the problem of resynthesizing a logic network with small gates into a logic network with larger gates, where the gate’s size refers to the number of inputs. It can be considered the dual problem of LUT resynthesis, in which large gates are resynthesized into smaller gates. A concrete typical instance of LUT mapping, e.g., used in FPGA programming, is to map a gate-level network (with gates that do not have more than 3 inputs) into a logic network that supports gates which can implement arbitrary 6-input functions.

Most LUT mapping algorithms are based on cut enumeration. First, cuts are enumerated for *all* gates in the input logic network. Then some of the gates are determined to be in the mapped network by selecting one of the gate’s cuts, such that the following conditions hold: (i) all outputs must be mapped, (ii) if a gate is mapped, then also each leave in the gate’s selected cut must be either a primary input or mapped as well.

LUT mapping usually only determines which gates are mapped and which cuts are chosen; it does not create the larger LUT network. We follow this convention in CirKit, where the command ‘lut_mapping‘ performs the mapping by annotating gates in the logic network, and ‘collapse_mapping‘ creates the LUT mapping based on the annotations.

III. Experimental Results

Experimental setup. We have evaluated the quality and performance of the proposed RTL synthesis framework considering four different multi-level logic representations: (1) AND-inverter graphs (AIGs), (2) majority-inverter graphs (MIGs),

[2]https://github.com/msoeken/cirkit

TABLE I
NUMBER OF LUTS OF INDIVIDUAL MODULES AFTER RTL SYNTHESIS, LOGIC SYNTHESIS, AND LUT MAPPING

Benchmark (modules opt. 5×)					AIG		MIG		XAG		XMG	
Name	**I**	**O**	**LUTs**	**Levels**	**LUTs**	**Levels**	**LUTs**	**Levels**	**LUTs**	**Levels**	**LUTs**	**Levels**
modular_mul	258	512	512	1	512	1	512	1	512	1	512	1
modular_add	512	256	4827	55	1125	220	1035	72	1039	200	1044	35
modular_dbl	256	256	2287	35	361	62	357	38	360	62	382	13
sha1	948	911	9536	84	1672	48	1834	40	1488	38	1544	29
sha256	1140	1103	14649	77	2443	50	2697	43	2212	40	2241	29
sha512	2200	2162	30337	96	4971	82	5367	69	4246	63	4510	50
md5	10240	2048	83460	70	22086	68	23041	50	16595	59	15069	39
Benchmark (modules opt. until convergence)					**AIG**		**MIG**		**XAG**		**XMG**	
Name	**I**	**O**	**LUTs**	**Levels**	**LUTs**	**Levels**	**LUTs**	**Levels**	**LUTs**	**Levels**	**LUTs**	**Levels**
modular_mul	258	512	512	1	512	1	512	1	512	1	512	1
modular_add	512	256	4827	55	1015	242	1035	72	638	129	766	129
modular_dbl	256	256	2287	35	357	52	357	38	356	52	382	13
sha1	948	911	9536	84	1378	37	1777	40	1404	37	1443	25
sha256	1140	1103	14649	77	2164	38	2691	43	1988	38	2154	24
sha512	2200	2162	30337	96	-	-	5329	69	3627	69	3786	41
md5	10240	2048	83460	70	14605	65	23034	50	14429	62	13916	38

TABLE II
NUMBER OF LUTS AFTER RTL SYNTHESIS, LOGIC SYNTHESIS, LUT MAPPING, AND FLATTENING

Benchmark (opt. 5× & flattened)			AIG		MIG		XAG		XMG	
Name	**LUTs**	**DFFs**	**LUTs**	**Time [s]**	**LUTs**	**Time [s]**	**LUTs**	**Time [s]**	**LUTs**	**Time [s]**
modular_mul	527342	0	379442	64.18	355472	55.72	357257	108.84	364142	50.53
sha1	3014	911	1672	109.05	1834	61.58	1488	47.92	1544	62.10
sha256	3420	1103	2443	253.79	2697	85.83	2212	56.39	2241	80.60
sha512	7687	2162	4971	1796.94	5367	418.78	4246	472.47	4510	461.80
md5	23241	36608	22086	582.51	23041	744.46	16595	143.81	15069	244.54
Total runtime:				2806.46		1366.38		829.43		899.57
Average reduction:			27.22%		31.22%		32.39%		31.79%	
Benchmark (opt. until convergence & flattened)			**AIG**		**MIG**		**XAG**		**XMG**	
Name	**LUTs**	**DFFs**	**LUTs**	**Time [s]**	**LUTs**	**Time [s]**	**LUTs**	**Time [s]**	**LUTs**	**Time [s]**
modular_mul	527342	0	350072	175.53	355727	29.10	253982	92.22	293252	34.05
sha1	3014	911	1378	575.16	1777	179.96	1404	134.94	1443	163.77
sha256	3420	1103	2164	1421.30	2691	102.01	1988	236.92	2154	92.97
sha512	7687	2162	-	T/O	5329	1057.04	3627	2285.83	3786	1848.68
md5	23241	36608	14605	2317.90	23034	906.95	14429	240.98	13916	300.60
Total runtime:				4489.89		2275.06		2990.89		2440.07
Average reduction:			32.34%		31.19%		51.23%		44.29%	

(3) XOR-AND graphs (XAGs), and (4) XOR-majority graphs (XMGs).

As benchmarks, we use different arithmetic and cryptographic designs obtained from OpenCores.org (sha1, sha256, sha512, md5) and a hand-crafted modular multiplier design (modular_mul). The benchmarks are optimized with the flow script described in Section II, where 'cut_rewrite' is repeated 5× and repeated until convergence, respectively.

All experiments have been conducted on an Intel® Core™ i7-7567U CPU with 3.50GHz and 16GB RAM. We use a global time limit of 100 minutes for executing an RTL synthesis flow on a benchmark.

Experimental results. Table I presents synthesis results for individual modules of the RTL design. Each row corresponds to one module of a design. Bold font denotes a top-level module. The first five columns from left to right show the module name (**Name**), the number of primary inputs (**I**), the number of primary outputs (**O**), the number of LUTs (**LUTs**) and the number of levels (**Levels**) after LUT mapping of the individual modules. The remaining columns show the number of LUTs (**LUT**) and levels (**Levels**) after logic synthesis and LUT mapping for the four different multi-level logic representations, respectively.

Table II presents synthesis results after flattening the individual modules into one monolithic netlist. The first three columns from left to right show the name of the top-level module (**Name**), the number of LUTs (**LUTs**) and the number of flipflops (**DFFs**) of the flattened benchmarks without additional logic optimization. The remaining columns list the number of LUTs (**LUTs**) and the total run-time (**Time**) required for RTL synthesis with integrated logic synthesis for the four multi-level logic representations, respectively. We

mark the best area result in terms of LUTs in green color.

Discussion. Logic synthesis techniques allow us to reduce different cost functions (area, depth, etc.) when integrated into an RTL synthesis flow. In this work, we focus on area reduction for FPGAs. In our experiments, applying a logic synthesis flow reduces the number of LUTs by up to 51.23% depending on the intermediate logic representation in use. In general, cut rewriting for majority-based intermediate representations tends to converge faster, whereas allowing XOR gates in the intermediate representation leads to a better average reduction of the number of LUT gates. The latter effect is likely due to the choice of optimizing arithmetic and cryptographic benchmarks, which both typically contain many XOR gates. The time limit of 100 minutes is only once reached for the benchmark md5 for which LUT resynthesis into ANDs and inverters does not finish in time. Overall, XAGs and XMGs perform best: 51.23% of LUTs are reduced in 59.82m when using XAGs and 44.30% in 40.67m when using XMGs.

IV. Conclusion

We have proposed an integration of RTL synthesis with Generic Logic Synthesis. This integration allows us to run RTL synthesis while choosing one of many multi-level logic representations in technology-independent logic optimization. We have shown the synthesis results after LUT mapping for FPGAs considering four different RTL designs from OpenCores.org and a hand-crafted modular multiplier considering four different logic representations (AIGs, MIGs, XAGs, XMGs). The experimental results indicate that high-effort logic optimization techniques are capable of reducing the number of LUTs of these benchmarks by up to 51.23%. The results also show that the choice of the multi-level logic representation has an important impact on the compaction of the logic and the performance of the synthesis process. For instance, switching from AIGs to XAGs for the hand-crafted modular multiplier leads to a reduction of approximately 100'000 LUTs (27.45%) and almost reduces the total runtime by almost $2\times$ (47.47%).

The proposed integration of Generic Logic Synthesis and RTL synthesis enables several promising directions for future research:

1) **Mixing logic representations**: In the experiments, we chose the same logic representations for all modules in the RTL design. The presented approach, however, is capable of choosing different logic representations for each module of an RTL design. Designs such as processors, that integrate arithmetic as well as control logic, may substantially benefit from a more fine-tuned selection of logic representations. An interesting challenge for future work is to automatically decide which logic representations to chose for a module at hand.
2) **Mapping into technology libraries**: The proposed synthesis flow interacts with logic synthesis on the granularity of modules. Each combinational module is extracted from the RTL design, optimized, and re-integrated into the design. The modules are represented as technology-mapped LUT networks using Yosys standard gate library. Mapping from this standard gate library into the respective logic representation and back leads in some cases to suboptimal results. For instance, the number of LUTs increases after re-integrating the optimizing MIG of modular_mul into Yosys, such that optimizing the MIG $5\times$ leads to a better result than optimizing the MIG up to convergence. Experimenting with different gate libraries has the potential to result in better synthesis quality or runtime.

V. Acknowledgments

This research was supported by the Swiss National Science Foundation (200021-169084 MAJesty), by the European Research Council in the project H2020-ERC-2014-ADG669354 CyberCare, and by the EPFL Open Science Fund.

References

[1] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. G. Amarù, G. De Micheli, and M. Soeken, "Scalable generic logic synthesis: One approach to rule them all," in *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, 2019, pp. 70:1–70:6. [Online]. Available: https://doi.org/10.1145/3316781.3317905

[2] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *International Workshop on Logic Synthesis*, 2006, pp. 15–22.

[3] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, "The EPFL logic synthesis libraries," May 2018, arXiv:1805.05121.

[4] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: An open source framework from verilog to bitstream for commercial FPGAs," in *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*, 2019, pp. 1–4. [Online]. Available: https://doi.org/10.1109/FCCM.2019.00010

[5] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, 2010, pp. 24–40. [Online]. Available: https://doi.org/10.1007/978-3-642-14295-6_5

[6] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, 2019, pp. 1649–1654. [Online]. Available: https://doi.org/10.23919/DATE.2019.8715185

[7] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Trans. on Computer-Aided Design*, 2019, accepted, in press. [Online]. Available: https://ieeexplore.ieee.org/document/8634910

[8] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, FPGA 1999, Monterey, CA, USA, February 21-23, 1999*, 1999, pp. 29–35. [Online]. Available: https://doi.org/10.1145/296399.296425

[9] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, 2006, pp. 532–535. [Online]. Available: https://doi.org/10.1145/1146909.1147048

[10] J. Cong and Y. Ding, "FPGA technology mapping," in *Encyclopedia of Algorithms*, 2016, pp. 773–777. [Online]. Available: https://doi.org/10.1007/978-1-4939-2864-4_148