Logic Synthesis in VLSI Design

VLSI Design Paper - Winter Semester

Vinícius Carvalho Gomes
Hochschule Hamm-Lippstadt
viniciuscgomes.98@gmail.com

***Abstract*—This article will make a brief analysis of logic synthesis for VLSI design. The main steps for synthesis (translation, logic optimization and mapping) will be explained, as well as the main algorithms and techniques for each step. Lastly, a short overview of the main tools and softwares available today for logic synthesis will be provided.**

I. Introduction

The assemble of an embedded system is a long and complicated process, from the specification phase to the design phase and then finally to the implementation phase. From beginning to end of the design flow, the design gets from a very high level description gradually translating into a low level description from which the chip can be produced.

This paper will focus on the phase of hardware design when a RTL (resistor-transfer level) description needs to be converted into a gate-level netlist. This phase of the design flow is called "Logic Synthesis".

II. What is logic synthesis?

On the first phase of VLSI design, the hardware is described using high level language, describing the registers used, the operations performed and the general conditions to the correct functioning of the system. Later on, the hardware is described in a logic level (lower level), describing logic gates and storage elements. The process of converting the first description into the latter can be a straightforward task for a small and simple circuit, but for VLSI design it can be a long process and very susceptible to errors.

Algorithms to perform a computer-aided logic synthesis are nowadays available and bring many advantages to the design process, such as optimization of the model, allowing fewer bugs and improved productivity. According to Taraate [5], the logic synthesis process receives as an input a RTL design specification, a standard cell library and a set of design constraints and as an output it produces a gate-level netlist, mapped to the standard cell library.

The main goals of logic synthesis are to minimize the area, minimize the power consumption and maximize the performance. This 3 objectives can be combined with different weights to meet the design specifications.

III. Synthesis process for ASICs and FPGAs

Logic synthesis process for VLSI design can be described with three main steps. First, RTL blocks and operations need to be identified and then translated to gate level Boolean representation. Secondly, the Boolean logic must be optimized. Lastly, the generic Boolean netlist must be mapped to the gates on the specific technology library. These processes can be named translation, logic optimization and technology mapping.

A. Translation

In this step the RTL should be translated into Boolean data structure and mapped to generic, technology independent logic gates. That is, in the RTL, basic logic blocks must be identified (such as arithmetic operations, multiplexers, encoders) and then translated into blocks defined in the used library. The netlist generated should not be technology specific.

B. Logic Optimization

In the previous step each block is treated as a separate part, sometimes generating redundant parts in the system. In this step redundant parts should be removed and the design should be optimized. The main output of this process should be an optimal structure for the circuit independent of the gates available in the technology chosen to be used later on to implement the system.

The outputs can be described as Boolean functions of the inputs. Rudell [3] describes the two main algorithm techniques used in technology independent optimization in order to reduce the number of literals in these output functions: two-level logic minimization and multi-level logic minimization.

- Two-Level Logic Minimization

Two-level logic minimization consists in finding the minimum sum of products (SOP) that covers the output Boolean function. That means that the resulting output will only have two levels, one level of and gates and one single or gate.

The most used heuristic two level minimizer is the Espresso Algorithm. The Espresso algorithm consists in taking any SOP representation and repeating 3 steps (Reduce, Expand and Irredundant) until the product can't be reduced any further.

Taking the representation in a Karnaugh map, we can analyse the example of the Boolean expression $f = \overline{AC} + AD + AC + C\overline{D}$ in figure 1.

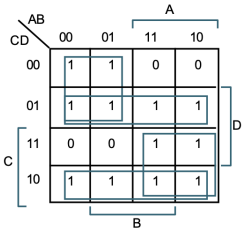


Fig. 1. 4 primes irredundant cover [6]

The first step of the Espresso algorithm would be to reduce the primes but still covering the ON-set. The resulting Boolean expression is $f = \overline{AC} + A\overline{C}D + AC + \overline{A}C\overline{D}$ in figure 2.

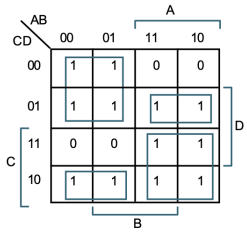


Fig. 2. Still 4 primes cover, but now with 10 literals [6]

The second step is to expand the primes, generating a different boolean expression for the same cover. The resulting Boolean expression is $f = \overline{AC} + AD + AC + C\overline{D}$ in figure 3.

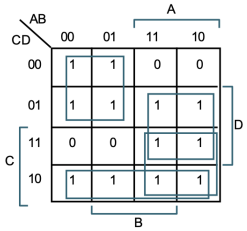


Fig. 3. New 4 primes cover, now with 8 literals [6]

Finally the last step is to eliminate redundant cubes. The resulting Boolean expression is $f = \overline{AC} + AD + C\overline{D}$ in figure 4.

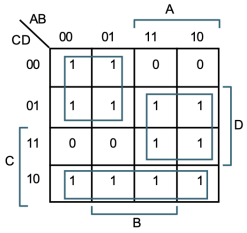


Fig. 4. 3 primes irredundant cover [6]

By the end of the Espresso Algorithm we have a optimal reduced two level description (SOP) achieved in a fast and simplified way.

- Multi-Level Logic Minimization

Though two level logic is widely used and is very effective, in some cases is better to use more levels of logic. The two level minimization techniques are still usable by separating the multi level expression into a set of smaller two level expressions, but they are not sufficient therefore new techniques need to be presented.

There are some operations that can be applied to multi level Boolean functions in order to reduce it. Extraction operations identify common sub expressions and replace them with one variable. Collapsing operations will eliminate intermediate variables that do not affect the final result. Simplification operations will use two level minimization algorithms to simplify smaller parts of the expression.

Boolean functions can also be represented as binary decision diagrams (BDD), a data structure consisting of several decision nodes and two terminal nodes of value 1 and 0. A reduced representation of a BDD can be achieved by applying two rules: merging equivalent leaves and merging isomorphic nodes. An optimized Boolean function is then acquired.

C. ASIC Technology Mapping

This phase of the synthesis receives as input a technology independent optimized netlist. This netlist, generated in the previous steps, may not be the most efficient solution if directly implemented and therefore must be adapted to be implemented in a specific technology.

Technology mapping will receive a Boolean function (the one generated in logic optimization) and transform it into actual gates that can be implemented. In ASIC design, each technology library has a set of specific gates that can be used and each gate is associated with a specific cost. The main focus on technology mapping will be to find the set of gates included in the library that can represent the Boolean function with the minimum cost.

The most famous model of technology mapping is called tree covering. The tree covering process can be separated in three parts: simple gate mapping, tree-ifying and minimum tree covering.

- Simple Gate Mapping

In this part, the logic NAND function property of functional completeness will be used, that is, any Boolean function can be expressed as a set of NAND2 gates. The first step is to represent the Boolean function found in logic optimization using only simple gates, that is, make the Boolean function a collection of only NAND2 and NOT gates. For example the Boolean function $f = a + b$ can be represented as $f = NAND(\overline{a}, \overline{b})$.

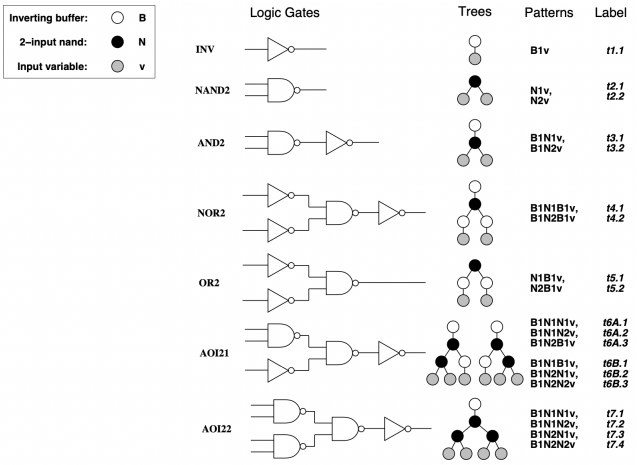


Fig. 5. Logic gates represented as a set of NAND2 and NOT gates [2]

Secondly, the same needs to be done with the standard cell library, as in figure 5. That is, given a technology library consisting of a set of gates each with a cost associated to it, it is necessary to represent all of those gates as a combination of NAND2 and NOT gates (the cost associated to each gate remains the same). For example a XOR gate ($f = XOR(a, b)$) can be represented as $f = NAND(NAND(\overline{a}, b), NAND(a, \overline{b}))$.

- Tree-ifying

In order to use the tree covering algorithm it is necessary to work on trees. However, not all logic network is a tree. A given logic network is not a tree if there is a fanout greater than 1 at any gate output, that is, a gate output must represent a single gate input. Any nodes that have a fanout greater than 1 must be cut, generating two separate trees, as shown in image 6.

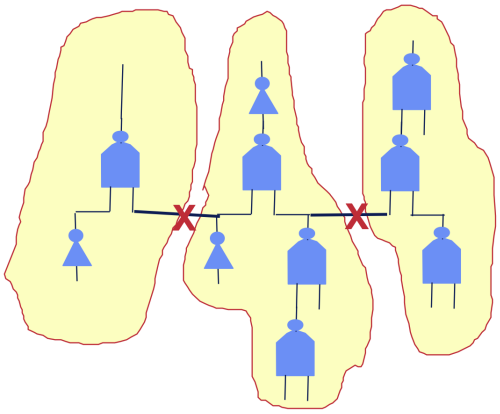


Fig. 6. This logic network needs to be split into 3 trees [4]

- Minimum Tree Covering

In this step the main goal is to find the minimum cover for the trees generated in the previous step. That is, find the way with minimal cost to cover the tree using the gates available in the cell library.

The best way to find the minimum tree covering is to apply a recursive algorithm. Starting at the output of the tree, it is necessary to find all the gates available at the library that match a sequence from the top node of the tree. The same is then done recursively for all the inputs for the previous gates until the whole tree is covered. Finally the cost of each resulting cover is compared and the minimum cover is selected.

In image 8 is possible to find the tree cover using the technology library presented in image 5. All the nodes have only one matching possibility in the library except for node O, that has three matching possibilities. The algorithm would choose out of the possibilities found the one that has the minimum possible cost. It is assumed that a cover is always possible (there is always a NAND2 gate in the technology library).

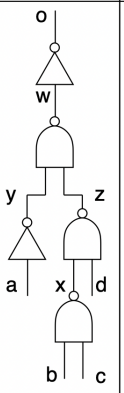
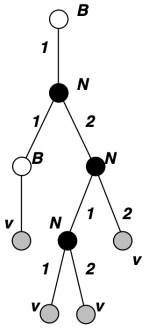
Network	Subject graph	Vertex	Match	Gate	Cost
		x	*t2*	NAND2(b,c)	NAND2
		y	*t1*	INV(a)	INV
		z	*t2*	NAND2(x,d)	2 NAND2
		w	*t2*	NAND2(y,z)	3 NAND2 + INV
		o	*t1*	INV(w)	3 NAND2 + 2 INV
			t3	AND2(y,z)	2 NAND2 + AND2 + INV
			t6B	AOI21(x,d,a)	NAND2 + AOI21

Fig. 7. Matching the paths in the Boolean Network with the technology library available. [2]

D. LUT Mapping (FPGA)

For Field Programmable Gate Array (FPGA), the basic logic elements are usually lookup-tables (LUT). A LUT of K logic inputs (K-LUT) can implement any Boolean function of K variables. A LUT is an array where a value is stored in a directly addressable slot, that is, each key is associated directly to the address where the value will be stored. Using a LUT means that for each combination of the K inputs of the structure, there is an output value stored in a specific slot that can be easily accessed.

In FPGA design, the problem of technology mapping is different from ASIC design, the Boolean function found in logic optimization must be mapped to the LUT available at the FPGA. In a K-LUT mapping, LUTs with K inputs and one output will be used to describe sets of gates in a Boolean network.

LUT mapping must try to achieve optimization according to certain criteria. Timing optimization is to cause minimal delay in the implementation. Area optimization is to make the implementation compact (usually fitting the most gates inside a single LUT). Power optimization is to make the implementation as low as possible in power consumption.

A LUT mapping can be analyzed in accordance to the optimization criteria. The area of a mapping is the total number

of LUT used. The maximum delay of a mapping is the maximum amount of LUT in a path from an input to the output.

The first step in LUT mapping is to define the possible cuts in the Boolean function. That is, to find the possible paths with K inputs that will result in a single output. The cuts can vary from covering one single logical gate to covering the whole Boolean network.

Once the possible cuts are defined, the next step is to select the cuts according to the desired optimization criteria. In this step, the whole Boolean network must be covered. It is possible (although not optimal) that redundant cuts need to be used, that is, cuts that cover nodes already covered by previous cuts.

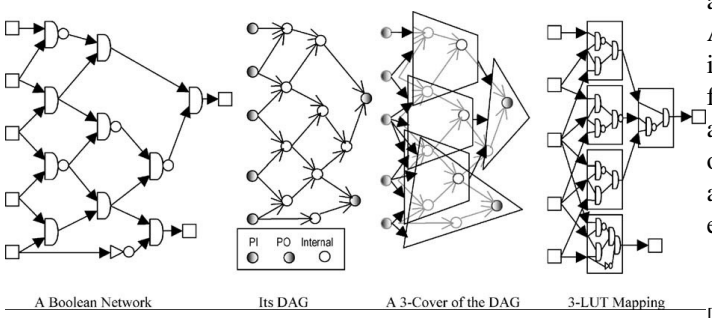


Fig. 8. LUT mapping example for K = 3, the mapping area is 5 LUT and the maximum delay is 2 LUT [1]

IV. Synthesis tools

Nowadays there are many available solutions for logic synthesis. Although most solutions are proprietary and hardware specific, there is a number of open source tools that attempt to make synthesis process more accessible.

Vivado Design Suite is the software designed by Xilinx for analysis and synthesis of HDL design. The software supports Xilinx devices, such as Xilinx Atix-7. Similarly, the Intel Quartus Primeis a software for analysis and synthesis of HDL code that supports intel devices, such as the Cyclone FPGA family. Although Xilinx and Intel are the market leaders for FPGA design and both offer proprietary design tools, there are other companies that offer synthesis solutions. For example, Synplify (by Synopsys) offers VHDL and Verilog synthesis solution for a variety of FPGA vendor, such as Intel, Xilinx, Achronix, Lattice and Microsemi.

For ASIC design there are also different solutions for logic synthesis, with Synopsys being the market leader. Open source solutions are also starting to emerge with non-proprietary solutions for ASIC and FPGA synthesis. For example, Yosys offer a Verilog RTL synthesis, beeing able to perform technology mapping for ASIC standard cell libraries and LUT mapping for Xilinx 7-series and Lattice FPGAs.

V. Conclusion

Logic synthesis is a fundamental step of VLSI design. In summary, starting with a RTL description, synthesis will result in a technology independent solution as a Boolean network. Later on, Mapping will turn the general purpose solution into specific logic gates or LUTs depending on the application technology chosen. Logic synthesis is still a developing technology, and as many new algorithms are emerging trying to find the most efficient solution to logic synthesis, it is fundamental to understand the basic concepts and functioning of the process to choose the best suiting approach to a given problem.

As much as classic algorithms and techniques for synthesis are important as the base of logic synthesis, various issues can be found with algorithms presented in this paper. For example, the tree covering algorithm is limited to tree structured networks and can become very high cost for larger networks, since all possible cover paths need to be analyzed.

Tools and software for logic synthesis is a growing market, and each time more solutions are being presented. Since most ASIC and FPGA fabricators keep their technology proprietary, it is hard to develop general purpose tools that will work for multiple technologies. Instead, most tools are specific to a certain type of FPGA or ASIC application. Nevertheless, open source tools for ASIC and FPGA design and synthesis are becoming each time more common and effective as an effort of the community.

References

[1] Jason Cong. Fpga technology mapping, 1992.
[2] Frédéric Mailhot. Technology mapping for vlsi circuits exploiting boolean properties and operations, 1991.
[3] Richard L Rudell. Logic synthesis for vlsi design, 1989.
[4] Rob A. Rutenbar. Technology mapping. https://course.ece.cmu.edu/ ee760/760docs/lec10.pdf, 2001.
[5] Vaibbhav Taraate. *ASIC Design and Synthesis*. Springer Singapore, 2021.
[6] Dr. Adam Teman. Digital vlsi design: Logic synthesis. https://www.eng.biu.ac.il/temanad/files/2017/02/Lecture-5-Synthesis.pdf, 2016.