HDL Synthesis, Inference and Technology Mapping Algorithms for FPGA Configuration

Nuocheng Wang

Northeastern University, Boston, USA
Email: jw411711848@gmail.com
Manuscript received August 18, 2023; revised September 25, 2023; accepted November 3, 2023; published March 15, 2024

***Abstract*—This paper introduces the logic control flow of Field-Programmable Gate Array (FPGA). The process from analyzing a digital circuit description to component mapping on FPGA is described thoroughly. This transforming process is partitioned into three major stages: combinational logic synthesis, sequential logic inference, and technology mapping. Specific algorithms are discussed for each stage.**

***Keywords*—Hardware Description Language (HDL), mapping, algorithm, Field-Programmable Gate Array (FPGA), configuration**

I. Introduction

A Field-Programmable Gate Array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing. It has been widely used for embedded systems in various applications, such as consumer electronics, medical devices, security systems, and defense industry applications [1]. Different from a CPU, which executes a program in a sequential manner, an FPGA is provided a bitstream to create appropriate hardware. The goal of this paper is to introduce the complete process of the logic control flow of an FPGA, specifically, the transformation from the digital logic description written by the user to the employment of logic components on FPGA. The paper is separated into five main sections describing the history, the architecture, the combinational logic synthesis, the sequential logic inference, and the technology mapping of an FPGA. The algorithms discussed here include the Boolean optimizing algorithm by McCluskey [2], the circuit description inferring algorithm by Gregory and Segal [3], and the component matching algorithm by Micheli [4].

First introduced by Xilinx [5] in 1984, FPGAs evolved a lot under the advancement of process technology and application demand. The evolution was reflected in an increase in capacity and speed as well as a decrease in cost and energy consumption. This period of evolution was divided into three eras: Age of Invention (1984–1991), Age of Expansion (1992–1999), and Age of Accumulation (2000–2007). Developers have traditionally had just a few options at their disposal when considering how to implement a given computation [6].

The Age of Invention was about creating working FPGAs. The first FPGA, the Xilinx XC2064, was equipped with only 64 logic blocks with each holding two Look-Up Tables (LUTs), each with three inputs, and one register [5]. Its size, in contrast to its capacity, surpassed the commercial microprocessors of the time and was a major drawback in an era that appreciated cost containment. In response, the antifuse process technology was able to eliminate the area penalty of memory-cell storage with one-time programmability as a tradeoff. As four-input functions commonly appear in digital designs, Xilinx used 4-input LUTs in their first FPGA designs [5]. However, many functions are not four-input, leading to underutilization of the LUT. To optimize logic cell usage, companies implemented fixed functions in the underutilized areas. Further emphasis on cost containment was shown in the interconnect architecture of the early FPGAs. Short wires between adjacent blocks were favored over long and slow wires of Programmable Array Logics (PALs) [5]. While shorter interconnects saved die area and provided efficiency, the large capacitance and distributed series resistance raised by pass transistors increased signal delay and delay uncertainty. As a result of minimal wiring, early FPGAs were extremely hard to use.

In the next era, the Age of Expansion, manufacturers competed against each other to optimize the design of FPGAs. Many optical flow algorithms have been developed in the last two decades [7]. Silicon efficiency was no longer the priority when designing FPGAs. As new generations of silicon processes were made available, the cost of transistors and interconnects dropped, which made larger capacity possible. Product characteristics, like performance, features, and ease-of-use, were weighed over previously precious area [5]. Furthermore, the continuous expansion of capacity on FPGA devices has made manual design less and less feasible. Demands to automate the design process emerged, giving FPGA companies who possessed design automation technologies leverage over the rest. Another benefit brought by process scaling was the abundance of interconnections, which made the board more programmable. Longer wires gave freedom to the placement of logic blocks, allowing a larger margin of error to the automated placement tools. By the end of the 1990s, automation was essential in the FPGA design process [5].

The trend of increasing capacity carried over to the next age, the Age of Accumulation. Nevertheless, continued expansion in size did not necessarily correspond to market growth when FPGAs of this size could already solve the majority of the designer's problems. Few companies demanded significantly large FPGAs and fewer were willing to pay extra for them. As a solution, FPGA vendors adopted separate approaches to accommodate diverse needs of customers. Products with lower capacity and lower performance were provided at a low cost, while products with libraries of soft logic for crucial functions, e.g., microprocessors, memory controllers, and communication protocol stacks, targeted high-end users [5]. In the 2000s, the application field of FPGAs expanded. Some customers now wanted to implement system standards, mostly communication standards, on the boards. FPGAs started to

play a bigger role in customer's overall system logic, resulting in growing cost and power. To address these concerns, manufacturers shifted their architecture strategy to add more dedicated logic blocks that were more efficient and cost-effective [5]. By 2005, FPGAs had been adapted to accommodate a growing range of applications.

II. Methodology

FPGAs can provide significant improvements in both overall performance and power efficiency by customizing the datapath of the algorithm and optimizing memory accesses [8]. The key to the flexibility of FPGAs is in their architecture, where sets of various kinds of programmable blocks, like logic and IO, are connected to routing tracks through programmable switches. The configuration of these units is controlled by millions of Static Random-Access Memory (SRAM) cells that are programmed at run time to realize a specific function described by the user. Known as field-programmable, FPGAs can easily implement a bug fix or a hardware upgrade by loading a new bitstream after launching. Furthermore, FPGAs, possessing a shorter time-to-market, can carry out new product designs in a matter of weeks. From version to version, FPGA architects use an evaluation process to improve performance. This architecture evaluation flow consists of three sections, namely a suite of benchmark applications, an architecture model, and a Computer-Aided Design (CAD) system [9].

Due to the wide range of applications and future extendibility of FPGAs, an FPGA architecture is evaluated based on its efficiency running various benchmark designs that are representations of popular market applications. These benchmark applications are collected by FPGA vendors from their customers as well as proprietary systems. Subsequently, all the components in a new design will be modeled to provide an overview of the architecture and to justify the decisions made. The organization of blocks and routing structure are represented in architecture description files, while area and timing characteristics are extracted from individual component implementations. The last step of evaluation is to map the benchmark designs onto the FPGA architecture using a CAD system. The performance is then measured by several key metrics: total area occupied by application, maximum frequency of application's clock, and power consumption [9]. Finally, the architecture is evaluated based on the averages across all benchmarks considering its designated purpose.

Programmable logic, being one of the essential parts of FPGA architecture, originated from the Programmable Array Logic (PAL) architecture that was built off from a two-level sum-of-products function. A PAL was configurable and offered constant delay against various logic functions. On the other hand, a PAL was not as efficient when facing large numbers of inputs. Growing device logic capacity would accumulate connecting wires and demand more programmable switches, thus, resulting in slower transmission. Complex Programmable logic Devices (CPLDs), a subsequent design, addressed the scalability problem by cross-connecting multiple PALs on the same die [9]. However, the implementation of CPLDs required more complicated design tools as a tradeoff. In 1984, the creation of the first Lookup-Table-based (LUT-based) FPGA solved the issue with a structure consisting of an array of SRAM-based LUTs interconnected [9]. Compared to the PAL architecture, LUTs were more compact in size, which made it popular ever since.

As the fundamental components of today's FPGA, LUTs were studied by manufacturers to find the most suitable combination of size and speed. A K-LUT can handle any K-input Boolean functions. It stores the truth table of the function in its configuration SRAM cells and uses the K input signals to select an output from the truth table. According to the studies conducted by [9], "LUTs of size 4-6 and logic blocks (LBs) of size 3–10 Basic Logic Elements (BLEs) offer the best area-delay product for an FPGA architecture". LUTs with less inputs correspond to a smaller size, while LUTs with more inputs provide a higher speed. In 2003, Altera introduced fracturable LUTs that were designed to combine the pros of both small and large LUTs. Fracturable LUTs improved the usage of the original under-utilized K-LUT by separating it into two K-1 sized LUT that could be configured as a K-LUT when needed. Fracturable LUTs also support additional inputs with an area cost based on design choice.

Affected by the emergence of fracturable LUTs was the number of Flip-Flops (FFs) per BLE. Early FPGAs with a non-fracturable LUT installed only one FF. Along with the introduction of fracturable LUTs and an increased demand for FFs to achieve higher performances, the number of FFs per BLE had increased significantly, as large as four in the Stratix V architecture [9]. Later, Stratix V replaced its FFs with pulse latches to remove one of the two latches in the master-slave architecture of an FF, further reducing the area and delay.

Programmable routing, another essential part of the FPGA architecture, generally covers more than half of both the die area and the critical path delay of applications. Programmable routing is usually made up of prefabricated wiring segments and programmable switches. FPGA routing architecture consists of two main types: hierarchical and island-style [9].

Hierarchical architecture, by its name, maintains a hierarchy between modules in which high-level modules instantiate low-level modules. The hierarchical design enables more frequent communications between modules that are closer in hierarchy. These communications are implemented with short wires that connect small regions on the chip [9]. On the flip side, the delay of long wires in the upper level does not improve much when the process is scaled. Additionally, physical distance does not correlate to the number of wires and switches needed to connect two logic blocks in a hierarchical architecture. Logic blocks that are physically close to each other may still communicate through a bunch of wires and switches. Thus, this routing architecture is mainly used for small FPGAs.

Island-style architecture constructs each logic block in an isolated manner that communicates with the outside world through switch and connection blocks. This architecture is made up of three key components: routing wire segments, connection blocks, and switch blocks. Connection blocks establish a connection between function block inputs and routing wires, while switch blocks arrange wiring to attain longer routes [9]. Utilizing intelligent placement algorithms, the CAD tool distributes elements of design to function blocks in a way that produces minimum wiring. Therefore,

the connections between function blocks are shorter and fewer routing wires are required in this architecture. Wire length is also an important property to consider in the design of island-style architecture. Modern applications adopt wires with multiple lengths to account for connections of different distances. The most popular option remains the moderate-length wires since a metal stack can only accommodate a certain number of long wires and short wires call for more programmable switches than needed.

A challenge that programmable routing faces is that process scaling decreases the size of processes but not necessarily the propagation time through long wires. On the contrary, the wire delay appears longer in terms of clock periods compared to an increasing clock frequency because of smaller processes. FPGA application developers address this problem by implementing more pipelining in their designs, which allows multiple clock cycles for long routes. They also use multiple clock domains to maximize the frequency that each individual component runs on. To perfect this strategy, some FPGA manufacturers have added registers to the routing network.

Widely used in designing switching circuits, Boolean algebra notation commonly describes the performance of a single-output combinational circuit with some input variables. The behaviour of a Boolean function, also known as the circuit transmission, can be described in a table of combinations, which lists all possible combinations of inputs and their corresponding outputs. Among which, the combinations where an output exists are regarded as elementary product terms (p-terms). A transmission can therefore be represented in a sum of p-terms, called a canonical expansion. Simplifying the canonical expansion can often achieve savings in gates needed to build a circuit, thus making it one of the most important problems of switching circuit theory. Famous simplification methods by Karnaugh, Aiken, and Quine are effective in addressing the problem on a smaller scale, but vulnerable to more complex functions or can hardly be implemented on digital computers. A more thorough approach by McCluskey [2] suggests that a more systematic way of handling complex functions is plausible.

The objective of McCluskey's [2] approach is to determine the minimum sums of a Boolean function, a sum of p-terms function with the fewest terms and fewest literals. The minimum sums can be obtained by choosing the sum functions that have the fewest terms and the fewest literal from an enumeration of all possibilities. However, finding the minimum sums by enumeration does not scale well. A more practical strategy by Quine involves prime implicants. Prime implicants of a function are obtained by a repeated application of theorem $x_1x_2' + x_1x_2 = x_1$ to all possible pairs of p-terms, then the resulting terms until it cannot be applied anymore. Combining the fewest prime implicants that satisfy the table of combinations will produce the minimum sum, with its terms referred to as ms-terms. The shortcoming of Quine's method is exposed when given a transmission containing many variables or p-terms [2]. countered with a solution that simplifies the notation and makes the procedure more systematic. Specifically, the literals are replaced by binary numbers and the omitted literals during simplification are represented using dashes. The revised method for determining prime implicants by McCluskey [2] is as follows:

1. List in a column the binary equivalents of the decimal numbers which specify the function.
2. Order the binary equivalents by the number of 1's and divide them into groups based on the number of 1's. An example is shown in Fig. 1.

TABLE II — DETERMINATION OF PRIME IMPLICANTS FOR TRANSMISSION

$T = \sum (0, 2, 4, 6, 7, 8, 10, 11, 12, 13, 14, 16, 18, 19, 29, 30)$

(a) I

	$x_5x_4x_3x_2x_1$
0	0 0 0 0 0 ✓
2	0 0 0 1 0 ✓
4	0 0 1 0 0 ✓
8	0 1 0 0 0 ✓
16	1 0 0 0 0 ✓
6	0 0 1 1 0 ✓
10	0 1 0 1 0 ✓
12	0 1 1 0 0 ✓
18	1 0 0 1 0 ✓
7	0 0 1 1 1 ✓
11	0 1 0 1 1 ✓
13	0 1 1 0 1 ✓
14	0 1 1 1 0 ✓
19	1 0 0 1 1 ✓
29	1 1 1 0 1 ✓
30	1 1 1 1 0 ✓

(b) II

	$x_5x_4x_3x_2x_1$
0 2	0 0 0 – 0 ✓
0 4	0 0 – 0 0 ✓
0 8	0 – 0 0 0 ✓
0 16	– 0 0 0 0 ✓
2 6	0 0 – 1 0 ✓
2 10	0 – 0 1 0 ✓
2 18	– 0 0 1 0 ✓
4 6	0 0 1 – 0 ✓
4 12	0 – 1 0 0 ✓
8 10	0 1 0 – 0 ✓
8 12	0 1 – 0 0 ✓
16 18	1 0 0 – 0 ✓
6 7	0 0 1 1 –
6 14	0 – 1 1 0 ✓
10 11	0 1 0 1 –
10 14	0 1 – 1 0 ✓
12 13	0 1 1 0 –
12 14	0 1 1 – 0 ✓
18 19	1 0 0 1 –
13 29	– 1 1 0 1
14 30	– 1 1 1 0

(c) III

	$x_5x_4x_3x_2x_1$
0 2 4 6	0 0 – – 0 ✓
0 2 8 10	0 – 0 – 0 ✓
0 2 16 18	– 0 0 – 0
0 4 8 12	0 – – 0 0 ✓
2 6 10 14	0 – – 1 0 ✓
4 6 12 14	0 – 1 – 0 ✓
8 10 12 14	0 1 – – 0 ✓

(d) IV

	$x_5x_4x_3x_2x_1$
0 2 4 6 8 10 12 14	0 – – – 0

Fig. 1. Determination of prime implicants for Transmission [3].

3. Compare each number with all the numbers from the group with exactly one more 1's.
4. For each number which has 1's or -'s wherever the number with which it is being compared has 1's or -'s, a new character is formed and recorded in a separate column.
5. Place a check mark next to each number which is used in forming a new character.
6. Repeat the process until no further reductions are possible.
7. The unchecked characters represent the prime implicants.

To determine the minimum sums from the obtained prime implicants, a prime implicant table is used, as shown in Fig. 2. Each column in the prime implicant table corresponds to a row in the table of combinations which outputs one, while each row represents a prime implicant. A cross is placed at the intersection if a prime implicant has value one for a transmission. Selecting the fewest rows such that each column has a cross in at least one selected row produces a minimum sum and the selected rows are known as basis rows. as shown in Fig. 3.

TABLE IV — PRIME IMPLICANT TABLE FOR THE TRANSMISSION OF TABLE II

	0	2	4	8	16	6	10	12	18	7	11	13	14	19	29	30	
A	x	x	x	x		x	x	x					x				*
B	x	x			x				x								*
C													x			x	*
D												x			x		*
E									x					x			*
F							x					x					*
G						x					x						*
H					x					x							*

Fig. 2. Prime implicant table for the transmission of table in Fig. 1 [3].

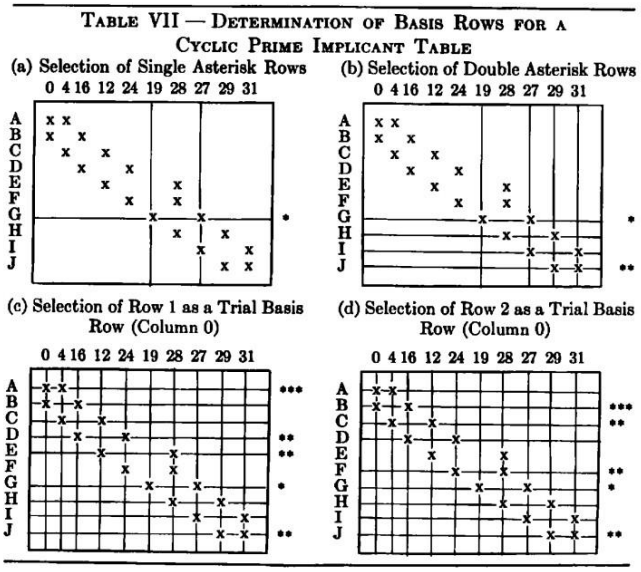

Fig. 3. Determination of Basis Rows for a Cyclic Prime Implicant Table [3].

The choice of basis rows can be complex, as it is an NP-complete set covering problem. McCluskey [2] gives some heuristics that work well:

1. Translate the table into a Boolean expression where addition stands for the choice of row in a column and multiplication connects such a condition with all columns.
2. Assign each row in the expression with a weight, w = n-log_2 k, where n is the number of variables in a transmission and k is the number of crosses in a row.
3. Compute the total weight of each row set by summing the weights of its rows. The row set with the smallest total weight corresponds to the minimum sums.

An example of such procedures is shown in Fig. 4. Even though the expressions derived from the table and the multiplication process can be lengthy, this method is systematic and suitable for automation.

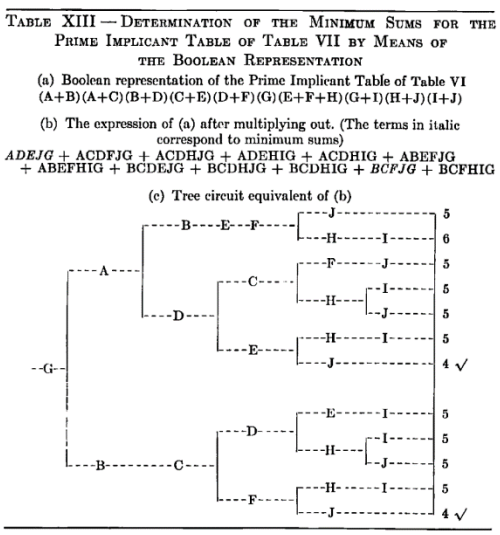

Fig. 4. Determination of the minimum sums for the prime implicant table in Fig. 3 by means of the Boolean representation [3].

Now that the user description specifying the combinational signals of a logic network can be simplified, a method is needed to convert the hardware independent description written by the user to a logic network, namely, a net list of logic components. Before this point, design was done by schematic capture. The creation of Hardware Description Language (HDL) opened the door to inference. HDL presented users with a way to describe the functionalities of a desired logic circuit and was also technology independent. On the flip side, early HDLs supported limited operation descriptions of circuit elements. Complex circuit elements like high impedance drivers, level sensitive latches and edge sensitive flip-flops would require the user to first specify the element type and then describe its connections, which assumed that the user had detailed knowledge of the desired circuit. Thus, HDL was only suitable for designers that knew both the operational behaviours and the hardware elements of the desired circuit.

To broaden applicable users, Gregory and Segal [3] suggested an automated logic design system with a circuit element independent HDL. The method proposed by Gregory and Segal [3], a logic circuit synthesizer, consisted of a preprocessor and a logic circuit generator. The preprocessor transformed the user description of signals and conditions into an equivalent representation in terms of nodes connected by edges, which indicated the conditions of the node being traversed. The logic circuit generator would then turn this structure into the actual logic network that performed as described by the user. In detail, the preprocessor was made up of a parser, a graph generator, and a condition generator. Upon receiving the signal information specified in the user description, the preprocessor would parse the description, convert it into a control flow graph, and determine the edge conditions for each node. The parser in the preprocessor would parse the statements in the description and store parsed results in a parse tree and symbol table for the graph generator, while the latter would then construct a control flow graph from them. Passing on to the condition generator, every node in the control flow graph would be analyzed to determine the condition necessary to reach that node. As the second portion of the logic circuit synthesizer, the logic circuit generator would take over the edge conditions and the nodes of the control flow graph generated by the preprocessor and convert them into assignment conditions. Overall, the logic circuit generator consisted of an assignment condition generator and a hardware generator. The task of the assignment condition generator was to produce an assignment condition matrix using the conditions and the control flow graph generated previously. Ultimately, following the assignment condition generator, the hardware generator was responsible for creating a logic circuit for each row of the assignment condition matrix.

Digging deeper into the inferring process that transforms user descriptions to hardware, the logic circuit generator is first tasked to determine an assignment condition for each variable in a Hardware Description Function (HDF). By definition, an assignment condition represents the condition under which the HDF is true for a particular variable. HDFs can be separated into synchronous and asynchronous functions, where synchronous functions are variable assignments on clock edges and asynchronous functions are all the other assignments. In the assignment condition matrix suggested by Gregory and Segal [3], six HDFs are considered. They are asynchronous Load Function (AL), Asynchronous Data Function (AD), Synchronous Load Function (SL),

synchronous Data Function (SD), Don'T Care function (DC), and High-Impedance function (Z). AL refers to the condition under which a variable is assigned any value, while AD tells the condition under which a variable is assigned the value one. Similarly, SL and SD represent the same conditions as AL and AD respectively but only on a clock edge. DC describes the condition under which a variable is assigned the value "X", meaning that the signal level of the variable is not important. Z, by its name, reflects the condition under which a variable is assigned the value "Z", saying the variable is high-impedance.

An example conversion from a user description, shown in Table 1, to its corresponding assignment conditions, shown in Table 2, is complemented for further comprehension. According to the user description, variable *P* is always assigned a value and is assigned the value one only when the condition COND1 is true. Neither assignment is associated with clock edges. Therefore, the AL of variable P is one and the AD of variable P has the value of COND1. Conversely, variable Q specified by the user is only assigned a value of B when COND2 is true. Thus, the AL of variable Q is COND2 and the AD of variable Q is the product of COND2 and B.

Table 1. An example of user description [3]

An Example of User Description 110
if (COND1)
P:=1
else
P:=0
endif
if (COND2
Q:=B
else
endif.

Table 2. Assignment conditions for the user description oF Table 1 [3]

	Assignment Conditions					
Variable	**Load Function (AL)**	**Asynchronous Data Function (AD)**	**Synchronous Load (SL)**	**Data Function (SD)**	**Don't Care (DC)**	**High-Impedance (Z)**
P	1	COND1	0	0	0	0
Q	COND2	COND2×B	0	0	0	0

The final step of the logic circuit generator will be transforming the produced assignment conditions to the corresponding hardware components. The hardware generator will generate a logic circuit for each row of the assignment condition matrix. Using the assignment conditions, shown in Table 6 above, for variable Q as an example, the behaviour of the assignment conditions related to variable Q can be replicated by a flow through latch controlled by AL where AD is the data input signal and the value of Q is the output signal. Seeing the complete converting process, this strategy by Gregory and Segal [3] presented a systematic way of inferring from user descriptions to generate a corresponding logic circuit.

In recent years, a number of different schemes have been proposed to implement optical flow algorithms in real-time [10]. Taking the logic network generated from inference, technology mapping transforms it into an interconnection of components that are instances of elements of a given library. Formally, technology mapping, also known as cell-library binding, is the process of transforming an unbound logic network into a bound network. Searching for the most optimized library component that performs the desired task presents a complex and difficult problem. Area cost, propagation delays, and testability enhancement are aspects to be considered in the binding process. Existing solutions are categorized into two types: heuristic algorithms and rule-based approaches. Both have their advantages and drawbacks. Heuristic algorithms handle only single-output combinational cells, while rule-based approaches are capable of complex libraries but require much effort in creation and maintenance of the set of rules and are also slower. Thus, the two approaches are often combined to achieve optimum performance.

Assuming a combinational logic network that has already been optimized using the Boolean minimization and inference techniques discussed previously, cell-library binding aims at finding an equivalent logic network composed of instances of library cells. Finding such a logic network is very complex, because the essential determination of equivalence between an unbound and a bound network presents a tautology problem. In general, the library binding problem can be interpreted as a network covering problem where a portion of an optimized logic network is substituted by cell-library elements with minimized area or delay. A cell is said to match a subnetwork when they have the same functionalities. When library instances achieve one-to-one mappings with the vertices in a network, the binding is trivial. For the network covering problem to have a solution, each local function needs to have at least one match in the cell library. Nevertheless, solving the library binding problem with the network covering approach is difficult due to its binate nature. The choice of any library cell in a network requires the selection of other cells to prove its connectivity. Therefore, heuristic algorithms were developed to approximate the solution of the network covering problem.

Algorithms for library binding can be divided into two

major types based on how the network and the library are represented: Boolean and structural. In the Boolean representation, the library cells and the portion of the network of interest are described in Boolean equations. In the structural representation, the Boolean equations describing the library cells and subnetworks are further reduced into an algebraic representation that can be cast into a graph. The usage of the two representations mainly differs in matching strategies.

Structural matching identifies the common patterns between a subject graph, obtained from the logic network, and its corresponding pattern graphs, associated with library elements. The subject and pattern graphs are acyclic and rooted. The method can then be simplified by representing library cells in the form of rooted trees. Tree matching and tree covering problems can be solved in linear time.

Two tree-based matching methods are considered here: simple tree-based matching and tree-based matching using automata. The simple matching method determines if a pattern tree is of the same structure as a portion of the subject tree. Starting from the root of the pattern tree and a vertex of the subject tree, the simple method compares the degrees of pairs of vertices in both trees until the leaves of the pattern tree are reached. The runtime of this algorithm is linear in the size of the graphs. The other tree matching method relied on an encoding of the trees by strings and a string recognition algorithm to represent the cell library systematically. In short, the algorithm would process strings that encode paths in the subject tree and recognize those that match paths in pattern trees. In comparison, the method using automata could match all patterns concurrently while the simple method could only match one at a time. However, this advantage would be countered by the increased complexity of handling trees as separate strings.

To obtain optimum tree covering, dynamic programming would be used. First, the minimum-area covering problem was considered. The tree covering algorithm would traverse the subject graph in a bottom-up fashion to minimize the total area of the bound network knowing the area cost of independent cells. For all vertices of the subject tree, the algorithm would match the locally rooted subtrees with the pattern trees. Next, the minimum-delay problem would be considered, with each cell's timing cost characterized by its input and output propagation delay. The subject tree would be traversed in the same bottom-up fashion to find the binding that minimized the data-ready time at each vertex, and thus, the minimum delay at the root. The runtime of these algorithms are linear with respect to the size of the subject tree.

Nevertheless, the covering algorithms utilizing structural matching had certain drawbacks. First, a library cell could be associated with more than one pattern graph, which led to cumbersome testing for the vertices of the subject graph. Second, there were cells that could not be represented by trees, like the EXOR and EXNOR gates. Although they could be replaced with leaf-dags, directed acyclic graphs where paths stem from the root can only reconverge at the leaves, the usage of these cells would be limited. Third, structural matching did not support the use of don't care in the binding process. The result could be not perfectly optimized.

Boolean matching complemented the disadvantages of structural matching. Boolean matching would perform an equivalence check between two functions, one being the cluster function that represented a portion of the network and the other being the pattern function that represented a cell. The Boolean covering algorithm would identify matches between the cluster functions of subnetworks and the pattern functions in the library, and subsequently, choose an amount of identified matches that minimize the area cost or delay of the network. Although Boolean covering and matching seemed more computationally expensive given that it required function derivations, studies showed that its computing times were comparable to that of structural covering and matching. One advantage that the Boolean covering algorithms had over the structural counterparts was their ability to exploit the degrees of freedom provided by don't care conditions and discover matches missed by algorithms based on structural matching, resulting in better solutions.

Given that FPGAs are pre-wired circuits, specific library binding techniques are required. Two types of FPGAs are considered in the following discussion of binding algorithms: Look-Up Table FPGAs and Anti-Fuse-Based FPGAs. For look-up table FPGAs, the virtual library is too gigantic for binding algorithms using enumeration and varies depending on the FPGA model. Thus, the suggested binding algorithm for look-up-table-based FPGAs focuses on combinational networks. The objective is to find an equivalent logic network with minimum critical path delay such that each vertex is matched by a function implementable by a look-up table. Micheli [4] adapted tree covering in their binding algorithm so that a subject graph was decomposed into parts that were put into a look-up table. Each decomposed part of the subject graph was meant to possess as much functionality as possible within the input size constraint.

An algorithm that optimizes the capacity of look-up tables is proposed by Micheli [4]. Any n-input single-output function can be represented in a sum of products form with at most n variables. First, the algorithm finds the product term with the most variables and puts it into the first table that fits. If none exists, a new table will be created to store the term. This process is repeated until all product terms are assigned to a table. Next, the table with the most variables is declared final and is associated with a new variable. Again, the algorithm looks for possible fitting tables to hold the new variable. The number of required tables decreases on every iteration. In the end where there is only one table left, that table produces the final result. This algorithm works well when the product terms are disjoint and the size of input n is less than or equal to six.

For anti-fuse-based FPGAs, the virtual library holds all logic functions implementable by personalizing the logic module. Since the layout of the FPGAs and the type of logic module differ from model to model, the problem was generalized to accommodate various anti-fuse-based FPGAs. All programmable modules are assumed to implement the same single-output combinational function, from here on referred to as the module function. Binding aims to determine an equivalent logic network with a minimum number of vertices such that each vertex can be matched with a personalization of the module function. In general, the size of the virtual library is so large that the previously mentioned

binding algorithms will not be useful. Alternative binding algorithms are proposed based on structural and Boolean representations. When employing a structural representation, the algorithm takes advantage of the uncommitted module to decompose the subject graph. Then, the library can be represented by pattern graphs that have a similar decomposition. Finally, the results can be obtained through structural covering with dynamic programming. Boolean approaches make use of the previous Boolean covering algorithm along with a modified matching algorithm that checks the implementability of a cluster function by module personalization.

III. Conclusion

Putting together the combinational logic synthesis, sequential logic inference, and technology mapping sections, an automated process of transforming the user circuit description to logic components employment on FPGA is obtained, which represents the logic control flow of FPGA. Another crucial aspect of FPGAs is place-and-route, which is not covered in this paper. In further studies, algorithms that optimize placing and routing among hardware components can be investigated.

Conflict of Interest

The author declares no conflict of interest.

Acknowledgment

I deeply appreciate Professor Bill Nace, who is professor of Electrical and Computer Engineering department in Carnegie Mellon University, for guiding me through every step of this project. Professor Bill Nace has shown the utmost dedication and passion towards helping me comprehend many concepts that were novel to me.

In addition to understanding books and articles and their implications to this paper, Professor Nace was also of immerse help during the drafting and editing phase. Many of my ideas came into shape with his guidance. His help made this project possible.

References

[1] H. Yu, H. Lee, S. Lee, Y. Kim, and H.-M. Lee, "Recent advances in FPGA reverse engineering," *Electronics*, vol. 7, no. 10, 246, 2018.

[2] E. J. McCluskey, "Minimization of Boolean Functions*," *Bell System Technical Journal*, vol. 35, no. 6, pp. 1417–1444, Nov. 1956. doi: 10.1002/j.1538-7305.1956.tb03835.x

[3] B. Gregory and R. Segal, *Method for Generating a Logic Circuit from a Hardware Independent User Description Using Assignment Conditions*, May 5, 1998

[4] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill Science, Engineering & Mathematics, 1994.

[5] S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," in *Pro. the IEEE*, vol. 103, no. 3, pp. 318–331, Mar. 2015. doi: 10.1109/JPROC.2015.2392104

[6] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D.-J. Lee, "Real-time optical flow calculations on FPGA and GPU architectures: A comparison study," in *Proc. 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, April 14–15, 2008. doi: 10.1109/fccm13926.2008

[7] Z. Wei, D.-J. Lee, B. Nelson, and M. Martineau, "A fast and accurate tensor-based optical flow algorithm implemented in FPGA," in *Proc. 2007 IEEE Workshop on Applications of Computer Vision (WACV '07)*, Feb. 21–22, 2007. doi: 10.1109/WACV.2007

[8] J. Monson, M. Wirthlin, and B. L. Hut, "Implementing high-performance, low-power FPGA-based optical flow accelerators," in *Proc. International Conference on Application Specific Systems (ASAP), Architectures and Processors*, June 5–7, 2013. doi: 10.1109/ASAP31104.2013

[9] A. Boutros and V. Betz, "FPGA Architecture: Principles and progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021. doi: 10.1109/mcas.2021.3071607

[10] Z. Wei, D.-J. Lee, and E. N. Brent, "FPGA-based real-time optical flow algorithm design and implementation," *Journal of Multimedia*, vol. 2, no. 5, Sep. 2007.