TECHNOLOGY MAPPING FOR THE ATMEL FPGA CIRCUITS

Zoltan Baruch
E-mail: Zoltan.Baruch@cs.utcluj.ro
Octavian Creț
E-mail: Octavian.Cret@cs.utcluj.ro
Kalman Pusztai
E-mail: Kalman.Pusztai@cs.utcluj.ro

Computer Science Department, Technical University of Cluj-Napoca,
26-28 Bariţiu St., 3400 Cluj-Napoca, Romania

ABSTRACT

In this paper we present a technology mapping algorithm for the *ATMEL 6002* FPGA circuits. The algorithm tries to balance cell utilization with the goal of producing routable mappings. The complexity of the technology mapping step may be considerably reduced if the internal representation generation is very well conceived. We implemented a program for internal representation generation, which leads to a very favorable starting point for the technology mapping program, eliminating the need for the partitioning and decomposition steps. Our algorithm takes into account the maximum capacity of the routing channels of the FPGA circuit and performs the partitioning step only if necessary (if the fan-out of a cell is larger than this capacity).

1. INTRODUCTION

A *Field Programmable Gate Array* (FPGA) consists of a prefabricated array of logic blocks and routing resources which can be programmed to perform a particular function. They provide a new approach to *Application Specific Integrated Circuits* (ASIC) implementation that features both large scale integration and user programmability.

The level of integration available in FPGA's is large enough to make manual circuit design impractical and therefore automated logic synthesis is essential for the efficient design of FPGA circuits. Logic synthesis, in general, takes a functional description of the desired circuit, and uses the set of circuit elements available to produce an optimized circuit. For an FPGA the set of available circuit elements consists of the array of logic blocks.

To implement a logic function with an FPGA circuit, one must perform the following tasks: logic minimization, technology mapping, placement and routing. Except for logic minimization, to which traditional techniques for cell-based designs are still applicable, all tasks must take into account the features that are unique to FPGA's.

Technology mapping is the logic synthesis task that is directly concerned with selecting the circuit elements used to implement the optimized circuit. Most approaches focus on using circuit elements from a limited set of simple gates.

The routing resources in FPGA architectures must be balanced against cell resources in order to accommodate a wide spectrum of designs. Although the number of cells available in an FPGA circuit is a hard constraint, minimizing the number of cells is pointless if the mapped design cannot be routed. Typically, only 80% of the cells can be allocated in a mapping that is routable.

In this paper, we present a mapper that tries to balance cell utilization with the goal of producing routable mappings.

2. PREVIOUS WORK

In early logic synthesis systems, such as *SOCRATES* and *LSS*, technology mapping is performed by a series of local transformations to a circuit netlist. These systems include rule-based expert systems used to select the sequence of local transformations. The netlist is initially constructed by implementing each node in the original network by a single circuit element. The area and delay of the circuit are then optimized by selecting the appropriate sequence of transformations [5].

An important advance in technology mapping was the formalization introduced by Keutzer in *DAGON* and used in *misII*: the set of available circuit elements is represented as a library of functions and the construction of the optimized circuit is divided into three sub-problems: *decomposition*, *matching* and *covering* [6].

In ASIC implementation technologies that use cell generators to create circuit elements, the set of available circuit elements consists of a parameterized family of cells rather than a specific library of functions. This cell family contains all members of a class of functions, such as AndOrInverts (AOIs), that do not exceed parameters defining the family. Library-based technology mapping is inappropriate for cell generator technologies when the number of cells in the family is too large to be practically expressed in a library. The key to cell generator technology mapping is the completeness of the cell family. This simplifies the matching of network sub-functions to circuit elements. If a sub-function does not exceed the parameters defining the family, it can be implemented by a cell in the family. In addition, simplified matching makes it possible to improve the final circuit by combining decomposition and matching.

With the increase of the level of integration available in FPGA chips, each logic block may consist of a *k*-LUT (a 1-bit memory with *k* inputs and one single output, implementing the truth table of the logic function). Most of the LUT technology mappers either start from a circuit decomposed into small gates with no more than *k* inputs or they decompose a given circuit into such a form. The initial network is thus already feasible because each node can be implemented by a *k*-input LUT. The objective is then to obtain another feasible network with less LUT's. There are several published technology mappers (*mis-pga*, *xmap*, *chortle*, *Flow-Map*, *Tech-Map* [3]).

3. BACKGROUND

The algorithms for technology mapping are based on Boolean techniques for matching, i.e., for the recognition of the equivalency between a portion of a network and library cells.

The computer-aided synthesis of a logic circuit involves two major steps: the *optimization* of a technology-independent logic representation, using Boolean and/or algebraic techniques, and *technology mapping*. Logic optimization is used to modify the structure of a logic description, such that the final structure has a lower cost than the original. Logic optimization has traditionally been done before technology-dependent operations, and in the following is assumed to have already taken place.

Technology mapping is the task of transforming an arbitrary multiple-level logic representation into an interconnection of logic elements from a given library of elements. Technology mapping is a crucial step in the synthesis of logic circuits for different technologies, such as sea-of-gates, gate arrays, or standard cells. The quality of the synthesized circuits depends heavily on this step.

The technology mapping transformation implies two distinct operations: recognizing logic equivalence between two logic functions, and finding the best set of logically equivalent gates whose interconnection represents the original circuit. The first operation, called *matching*, involves equivalence checking and input assignment. Checking for logic equivalence has been proved to be NP-complete. Input assignment is also computationally complex. The second operation, called *covering*, involves finding an alternate representation of a Boolean network using logic elements that have been selected from a restricted set [6].

The two operations intrinsic to technology mapping, matching and covering, are computationally difficult. For this reason, several approaches to technology mapping have been pursued and implemented in research and commercial mapping tools (rule-based technology mappers and heuristic algorithms). In this paper, we consider an algorithmic approach to the technology mapping problem.

Most algorithmic approaches to technology mapping attack the problem by dividing it into sub-tasks. First, Boolean networks are partitioned into an interconnection of single-output sub-networks, with the property that each internal vertex has unit outdegree (i.e., fan-out). Then, each sub-network is decomposed into an interconnection of two-input functions (e.g., AND, OR, NAND, or NOR). Each sub-network is modeled by a directed acyclic graph (DAG), called a *subject graph*. Finally, each subject graph is then covered by an interconnection of library cells, to produce the final circuit.

In the following sections we present the major tasks in technology mapping.

3.1 Partitioning

Partitioning is a heuristic step that transforms the technology mapping problem for multiple-output networks into a sequence of sub-problems involving single-output networks. Partitioning is performed during the initial setup phase and as a part of the iterative improvement of a mapped network. We treat here the first case.

The initial partitioning scheme is achieved by grouping vertices into single-output sub-networks, with the property that each outgoing edge of an internal vertex reconverges at or before the output vertex of the sub-network. Partitioning is also used to isolate the combinational portion of a network from the sequential elements and from the I/O's, where *ad-hoc* techniques for mapping are used.

After the partitioning step, the circuit is represented by a set of combinational circuits that can be modeled by *subject graphs* (single-output Boolean networks).

3.2 Decomposition

Decomposition is applied to each subject graph after partitioning. It yields an equivalent subject graph, where each vertex is a *base function*, e.g., a two-input AND/OR/NAND/NOR function. Decomposition provides a mapping solution for libraries that include the base functions (i.e., almost all libraries). Decomposition also increases the granularity of the network, which is beneficial for the covering step.

3.3 Network Covering

At this point, the logic circuit to be mapped has been partitioned into subject graphs [Γ_1, ... Γ_k], that have been decomposed. We denote by Γ_f a subject graph whose single-output vertex is v_f. We consider here the covering of a subject graph Γ_f that optimizes some cost criteria (e.g., area or timing). For this purpose we use the notions of *cluster* and *cluster function* [6].

A *cluster* is a connected sub-graph of the subject graph Γ_f, having only one vertex with zero out-degree v_i (i.e., a single output). It is characterized by its depth (longest directed path to v_i) and its number of inputs. The associated *cluster function* is the Boolean function obtained by collapsing the Boolean expressions associated with the vertices into a single Boolean function. All possible clusters rooted at vertex v_j of Γ_f and their functions are denoted by $\{\kappa_{i.1}, ..., \kappa_{i.n}\}$.

Example. For the following Boolean network:

$$\begin{aligned} f &= j + t \\ j &= xy \\ x &= e + z \\ y &= a + c \\ z &= \bar{c} + d \end{aligned} \tag{1}$$

there are six possible *cluster functions* containing the vertex v_j of the subject graph Γ_f:

$$\begin{aligned} \kappa_{i.1} &= xy \\ \kappa_{i.2} &= x(a + c) \\ \kappa_{i.3} &= (e + z)y \\ \kappa_{i.4} &= (e + z)(a + c) \\ \kappa_{i.5} &= (e + \bar{c} + d)\, y \\ \kappa_{i.6} &= (e + \bar{c} + d)\, (a + c) \end{aligned} \tag{2}$$

The covering algorithm attempts to match each *cluster function* $\kappa_{i.k}$ to a library element. A cover is a set of clusters matched to library elements that cover the subject graph. A cover may optimize the overall area and/or timing. The cost of a cover is computed by adding the cost of the clusters corresponding to the support variables in the *cluster function* $\kappa_{i.k}$ to the cost of the library element, for any permutation of its variables.

4. OUR APPROACH

4.1 Internal representation generation

The starting point is the design specification in the *ABEL-HDL* hardware description language. The .ABL source file is compiled and an equations-based specification is obtained, in the PDS format (the .PDS file is generated using the *Easy-ABEL* development system).

Our program generates the design netlist, first by building a component graph. In this graph, the *vertices* represent basic logic components (e.g., 2-input logic gates, flip-flops, tri-state buffers, etc.), and the *edges* represent the nets (interconnections between basic logic components). The basic components were chosen to be the simplest possible, because the logical network will be used in the technology mapping step, and it corresponds perfectly to the type of FPGA chip used (*Atmel 6002*).

Each equation in the .PDS file is parsed and a rooted graph is generated, the root being the vertex corresponding to the output signal for the parsed equation. A distinction is made between combinatorial and registered assignments.

After building each equation's graph, it must be appended to the general circuit graph. A very important step is *redundancies elimination*, resulting in a considerable amount of vertices being eliminated from the graph; this operation is done when parsing each equation.

Finally, according to the pin list in the .PDS file, input and output pins are identified. The program makes the distinction between output pins and internal nodes, which are signals used only internally, important for optimization.

For example, for the circuit given by the following equations:

$$\begin{aligned} Q &:= (A*B+C) \\ Q.CLKF &= (CLOCK) \\ Q.TRST &= (/OE) \\ Q.SETF &= (D + E) \\ Q.RSTF &= (F) \end{aligned} \tag{3}$$

the generated logical network is shown in Figure 1.

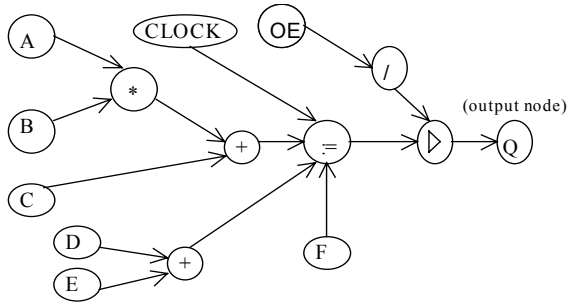


Figure 1. Internal representation generated from the equations (3).

The corresponding circuit is presented in Figure 2.

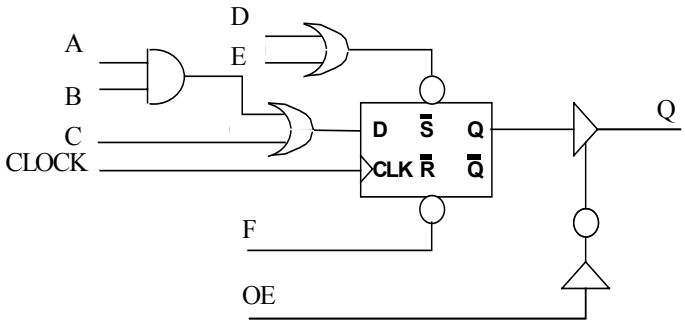


Figure 2. The corresponding circuit for the graph in Figure 1.

The program generates a file containing a netlist. It contains an adjacency matrix, where $a_{ij} = 1$ if there is a connection between the nodes i and j (an edge from vertex i to vertex j or from j to i inside the graph).

4.2 The Technology Mapping Algorithm

Our algorithm uses the logical network – obtained after the internal circuit representation step – as a starting point. This logical network has some interesting properties which will be used by the algorithm.

As mentioned in Section 3, the first steps in technology mapping are partitioning and decomposition. Here we do not have to implement these steps, because our program has already generated a logical network which has the required properties. We do not partition again the logical network after its generation, because, as mentioned in Section 4.1, the redundancies have been eliminated during the generation. In addition, the logical network is already decomposed (i.e., it contains only the basic logic components).

Therefore, the only step that must be implemented is the network covering (which includes the Boolean matching). For the network covering, we have to take into consideration the library of available logical cell's configurations in the *Atmel 6002* FPGA chip.

An important aspect that we have taken into account is that the technology mapping step will be followed by the placement step and by the routing one. If the number of logic cells generated by the technology mapping step is too large, that will considerably increase the complexity of the placement step.

As can be noticed, there has to be a balance between:

- *generating as few cells as possible for reducing the complexity of the placement step*, and
- *generating enough cells so that the mapped design will be routable*.

In the internal representation, a vertex has one of the following types:

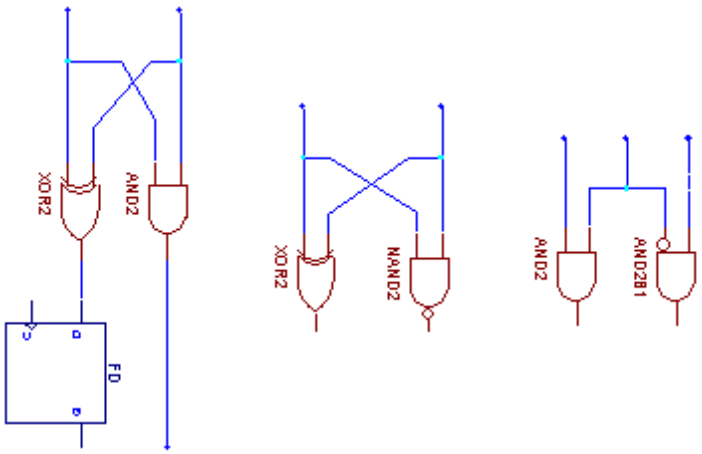
- input node
- output node
- D flip-flop
- logic gate (AND, OR, or INVERT)
- output tri-state buffer

In addition to the given types, there are a few more types which had to be introduced because of the particularities of the *Atmel* library of logic configurations. Therefore, the program detects the EXCLUSIVE-OR (XOR) gates and the 2-to-1 MULTIPLEXER (MUX) configurations embedded in the components graph. These configurations will be used in the effective technology mapping step.

It is important to start from the outputs of the design and execute the mapping process recursively towards the inputs, because in this approach the result is optimal (we tested this approach and the approach in which the starting point is a random point inside the graph, and the results were considerably better in the former case).

An important feature of the *Atmel* library of logic cells is that there are some complex cells which have *sub-configurations*, as shown in Figure 3. Consequently, after the first pass, the program has to merge together those configurations which can be mapped into the same cell (for example, the cluster functions $f = A * B$ and $Q := A \oplus B$ are mapped in the first cell (Figure 3.a), and $g = A \oplus B$, $h = /(A * B)$ – in the second cell (Figure 3.b). Of course, if there are other cluster functions '$A \oplus B$', it will not be possible to map them into the cell 3.a, because the only output of the cell sub-configuration is connected to the D input of the D flip-flop.

Therefore, after detecting and introducing the XOR and MUX configurations, the program starts from the outputs towards the inputs and maps the cluster functions (logic sub-trees) into the configurations or sub-configurations of the logic cells from the *Atmel* 6002 library. After that comes an *optimization* phase in the sense that the mapped structure is scanned and the program merges together the sub-configurations of the same cell, if possible. For example, we can merge together the cluster functions $g_1 = A \oplus B$ and $h = /(A * B)$ into the logic cell from Figure 3.b, but we could not do the same with the cluster functions $g_2 = A \oplus C$ and $h = /(A * B)$, because the signal C is not shared by the two sub-configurations of the cell, so g_2 and h will be mapped in two different instances of this type of cell.



a) **b)** **c)**

Figure 3. Three examples of Atmel logic cells with sub-configurations

Every cluster functions can be mapped in several sub-configurations. Every time a configuration is chosen, the program stores all the other possible mappings of that cluster function. For example, the cluster function $f = A * B$ will be mapped into the cell of Figure 3.a (the right-hand-side sub-configuration), but the program will also store the cell of Figure 3.c as an alternative for mapping the AND function. After the first mapping, the optimization phase will try to merge cells and if two clus-

ter functions can not be merged together in the same cell, then there will be taken into consideration the alternatives for each cluster function, and the merging process is restarted. That ensures that we obtain the best possible fit for the given library of components.

A very important aspect is that the mapped structure will has to be routable. It is not enough to determine which pairs of cells must be connected, we must supply more information. For example, simply saying that the cell of Figure 3.a must be connected to the cell of Figure 3.b does not provide enough information, because we must know also which sub-configurations are in fact connected, otherwise the routing program will be able to route those signals correctly. In order to satisfy these requirements, the output file format generated by our program contains in addition the following information: for each vertex in the newly generated cell graph (each vertex is a cell from the *Atmel* library) the used sub-configurations are specified; for each edge in the cell graph, the *start* sub-configuration and the *destination* sub-configuration are explicitly stored. That allows the following programs in the design process (the placement and the routing ones) to properly do their job.

A simple example: suppose we have to map the following cluster functions:

$$\begin{aligned} f &= A * B \\ g &= A \oplus C \\ h &= /(A * C) \\ k &= C * / B \end{aligned} \tag{4}$$

After the first pass, the mappings will be: f – in the cell of Figure 3.a, g and h – in the cell of Figure 3.b, k – in the cell of Figure 3.c. But after the optimization phase, the mappings will be: f and k - in the cell of Figure 3.c, g and h - in the cell of Figure 3.b, because f and k share the B signal, and g and h share both A and C signals just the way the cell structure requires it. This way we reduced the number of necessary cells from three to two.

Following is the description of the technology mapping algorithm.

Algorithm 1. Technology mapping.

```
forall input nodes
        map nodes to input pads;
endfor;

forall tri-state output buffers
        find successor (node);
        map tri-state output buffer to an output pad;
endfor;

forall remaining output nodes
        map node to an output pad;
endfor;

forall D flip-flops
        map node to one cell;
        assign cell to the same column; /* the clock is distributed on columns */
endfor;
forall mapped output pads
```

```
        recurse (sub-tree (node))
        boolean_match (node, library_element); /* based on node's label and*/
                                               /* library_element's label    */
        node = current_node;
endfor;

forall nodes in the graph /* now all the nodes are mapped – this is the optimization */
                          /* phase                                                    */
        forall other_nodes in the graph
        merge(node, alternatives, other_node) /* attempts to merge the nodes into the*/
                                              /* same cell                          */
        endfor;
endfor;
```

5. CONCLUSIONS

Technology mapping is a major step in design automation. It has a considerable influence on the placement and routing steps, which may be infeasible if the technology mapping step is not appropriately completed.

The technology mapping step's complexity may be considerably reduced if the internal representation generation is very well conceived. We implemented our own program for the internal representation generation, which leads to a very favorable starting point for the technology mapping program, eliminating the need for the partitioning and decomposition steps. In fact, we can consider the internal representation generation as a part of the technology mapping step.

Our algorithm has two phases: in the first one, the graph is recursively traversed and a first mapping is done (the cluster functions are assigned to cells sub-configurations). In the second phase, an optimization is done, by merging together into the same cell the cluster functions which have been mapped in sub-configurations of the same cell, if possible – the condition is that the sub-configurations share a given number of signals, depending on the type of the logic cell).

The generated output file contains not only the technology mapping information, but also indispensable information for the programs that will follow in the automatic design flow (the placement and the routing programs).

6. REFERENCES

[1] Atmel Corp., (1995) "Configurable Logic. PLD, FPGA, Gate Array", Data Book".

[2] Baruch, Z., Creţ, O., Pusztai, K., (1997) "Partitioning for FPGA Circuits", in Proceedings of MicroCAD'97 International Computer Science Conference, p113-116, Miskolc, Hungary.

[3] Chang, S. C., Marek-Sadowska, M., Hwang, T., (1996) "Technology Mapping for LUT FPGA's Based on Decomposition of Binary Decision Diagrams", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 10/96, p1226-1235.

[4] Chang, S. C., Tsay, Y. W., Hwang, T., Wu, A. C. H., Lin, Y. L., (1995) 'Technology Mapping for LUT FPGA's Based on Decomposition of Binary Decision Diagrams", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 9/95, p1076-1083.
[5] Francis, R. J., (1993) "Technology Mapping for Lookup-Table Based Field Programmable Gate Arrays", Ph.D. Thesis, University of Toronto, Canada.
[6] Mailhot, F., de Micheli, G., (1993) "Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 5/93, p599-620.
[7] Micheli, G., (1994) "Synthesis and Optimization of Digital Circuits", McGraw-Hill, New York.