EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Technology mapping for non-complete libraries

Kloprogge, R.J.L.

Award date:
1997

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Eindhoven University of Technology
Faculty of Electrical Engineering
Study Information Engineering
Design Automation Section (ES)

Technology mapping for non-complete libraries

(master thesis)

By R.J.L. Kloprogge

By order of: prof. Dr.-Ing. J.A.G. Jess
Supervisors: dr. ir. J.F.M. Theeuwen, ir. C.A.J. van Eijk
Period: December 1994 - August 1995

The Eindhoven University of Technology is not responsible
for the contents of training and thesis reports.

Abstract

The development of a newly designed technology mapper is discussed in this report. Technology mapping is part of the logic synthesis. It is the transition from technology independent combinatorial boolean functions to a technology dependent implementation.

The technology mapper discussed here uses a library of cells which implement simple combinatorial functions. Using the cells in a given library a cover of a given circuit has to be found that implements the same function. The mapper uses as library the more general non-complete libraries, these contain a set of selected cells. This technology mapper assumes that the library only contains single output cells. There are however already some functionalities built in that provide support for multiple output cells.

During the covering a cost function is used, striving for a cover that meets some constraints and is optimized with respect to some objectives. These constraints and objectives can for example involve area, power and delay. The technology mapper can be used with several cost functions. This report discusses primarily the use of area as cost function. Although area is not the most important objective anymore, it is used here as optimization factor because it is a more simple cost function.

The technology mapper consists of three modules: a subcircuit selection routine, a matching routine for boolean functions and a mapping and covering routine. The selection routine is used for searching subcircuits in the given circuit that can be implemented by a single library cell. This is a newly designed algorithm. The matching routine is used for establishing a match between a selected subcircuit and a library cell. It uses input signatures and symmetry classes to increase the efficiency of the matching. Using these matches the mapping routine decides depending on the cost whether it is a useful match. It creates a set of good matches for every signal. Also multiple output cells can be added to these sets. The covering routine then selects the final mapping from these sets. The union of all selected mappings then implements the given circuit.

The technology mapper is compared with three other technology mappers. The results of the mapper are very good. It provides a good cover, without using much CPU time. Also the results of some extensions to the basic algorithm are discussed. The technology mapper has a modular structure and because of this structure, future extensions can easily be implemented.

Contents

1 Introduction **1**

2 Problem description **3**

3 Circuit selection **7**
- 3.1 Circuit structure . . . 8
- 3.2 Circuit selection in trees . . . 8
- 3.3 Circuit selection in DAGs . . . 11
- 3.4 Further extension . . . 16

4 Comparing boolean functions **19**
- 4.1 Representation of boolean functions . . . 19
- 4.2 Input signatures . . . 20
- 4.3 Symmetric inputs . . . 21
- 4.4 Comparing functions . . . 22
- 4.5 Comparing in practice . . . 23
- 4.6 Other types of symmetry . . . 23

5 Mapping and covering **25**
- 5.1 Mapping of subcircuits . . . 25
  - 5.1.1 Mapping multiple fan-out signals . . . 26
  - 5.1.2 Mapping of multiple output library cells . . . 27
  - 5.1.3 Example of a mapping . . . 27
- 5.2 Covering the circuit . . . 29
  - 5.2.1 Covering the example circuit . . . 30

6 Experimental results **33**
- 6.1 Basic results . . . 33
- 6.2 Results for improved circuit selection routine . . . 34
- 6.3 Using all library cells . . . 35
- 6.4 Using specialized decompositions . . . 35
- 6.5 Using complete libraries . . . 36

7 Conclusions **37**

8 Recommendations for future work **39**

References **41**

Chapter 1

Introduction

Now the price of integrated circuits is dropping and the dimensions are diminishing, more and more hardware will be supplied with application specific integrated circuits. Therefore it is necessary to simplify the design of circuits, especially for large systems. This is the task of synthesis tools. Using these tools the efficiency of the design teams can be increased.

One of the steps during synthesis from a high level design specification to a layout is logic synthesis. During this step boolean functions are optimized and mapped onto hardware, resulting in a gate-level network. Technology mapping, which is a part of the logic synthesis process, is the step that converts combinatorial logic to hardware structures. The development of a newly designed technology mapper is discussed in this report.

Technology mapping uses a library containing small, technology optimized combinatorial circuits. The objective is to create a large circuit constructed out of these library cells that implements a given combinatorial logic function. This mapping is done under some user-supplied constraints. The result is a circuit that is optimized in some dimension; this can for example be area, delay or power.

The library can contain both single and multiple output cells. The method presented in this report concentrates on single output cells. Multiple output cells are difficult to implement constructively and will therefore not be used. However, a method will be given to still use these cells in the final mapping.

This report is organized as follows. First technology mapping is discussed in more detail and the corresponding subproblems are introduced. A solution for these subproblems is discussed in the next chapters. This is followed by results of the constructed technology mapper on some benchmark circuits. Finally some conclusions and recommendations for future work are given.

Chapter 2

Problem description

Technology mapping is the construction of a circuit of which the combinatorial function is given, out of a restricted number of smaller circuits. These smaller circuits are given in a library. The circuits in the library are parts that can be implemented on a chip. Here only the logic function and some technical characteristics like area, power and capacity are known. The whole of the logic function and values is called a cell.

The technology mapper uses the more general non-complete libraries instead of complete libraries. Complete libraries contain all possible combinatorial functions up to a certain size, where the size is given in the number of transistors next to each other and the number of transistors in sequence. Non-complete libraries contain only a selection of several functions and are therefore more general, because the user can compose his own library.

Using the cells in the given library a cover must be found of the given circuit. This cover must implement the given combinatorial boolean functions. The costs of the implementation are calculated using the characteristics of the cells. The costs can be a combination of several quantities like area, power and delay. These costs are used as constraint and optimization objective. The designer can provide his own cost function, the constraints and optimization objective.

In modern designs the clock speed at which the circuits have to run is increased continuously. Delay and power are therefore important optimization factors. These are however more difficult to optimize than area. Area is also often used as a benchmark value. Because of the better controllability of area, it is chosen here as the optimization value.

To find a cover of library cells which implement the given function, subcircuits have to be found in the function that can be mapped to library cells. These library cells can then replace that subcircuit. It is possible to create a new decomposition of the given combinatorial function trying to get a better result. The routine discussed here will not make a new decomposition. This is considered to be the task of other synthesis steps. The advantage is that the mapping routine can be controlled by supplying it with another decomposition.

The cover is a connection of library cells that replace the subcircuits in the original circuit. The borders of the cells are represented by signals in the original circuit. To get better results it is better to let the technology mapper

decide the phase of the signals. Because of this additional freedom better covers can be found. Of course the phase of inputs and output of the circuit may not be changed. For example, a multiplexer — $f = a\bar{c} + bc$ — will be covered as in figure 2.1(a) using a library containing ('nand', 'nor', 'inv', 'oai21', 'aoi22') if no phase changes are permitted. But if the phases of some signals are changed, then a better cover can be found as given in figure 2.1(b).

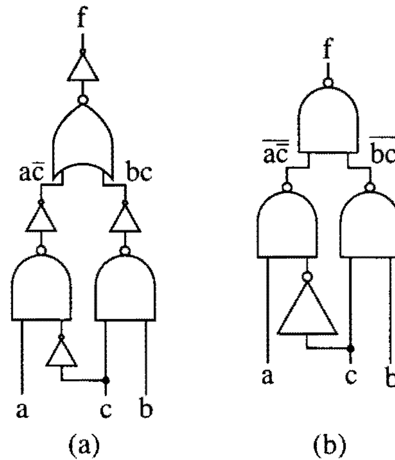

Figure 2.1: Cover of a multiplexer without and with phase change

An even better solution is possible when several signals are joined in one larger library cell. These signals become internal signals in a library cell and can not be reached from outside the library cell. This will results in the following cover, using the library cell $oai21 = \overline{(a1 + a2)b}$.

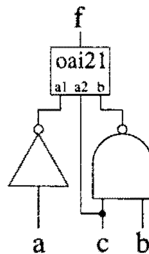

Figure 2.2: Cover of a multiplexer, joining several signals

Another possibility is the use of a larger library cell, namely $aoi22 = \overline{a1a2 + b1b2}$. This solution is given in figure 2.3. However here two inputs of a library cell are joined. It is not very efficient to let the technology mapper search for this kind of covers. It is better to add that complete cell — 'aoi22' plus the joined inputs, not the output inverter — to the library. Then it can efficiently be found. Adding these derived cells to the library can be done as a preprocessing step.

To find a good cover, the search for a cover is done exhaustively. To find the best library cell that can implement a signal, all possible subfunctions of a signal are searched and matched to library cells. Depending on the cost of each implementation, several best cells for each signal will be selected. When

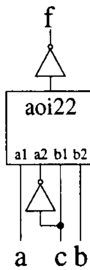

Figure 2.3: Cover of a multiplexer, joining several library inputs

mapping a subfunction to a library circuit, one can compare them functionally or structurally. The use of functional comparison is preferred. Structural equivalence implies functional equivalence. The reverse however is not true. So functional comparison increases the freedom of the mapping. The mapping routine discussed in this report therefore uses a functional comparison.

It is possible to find an optimal implementation of one subcircuit. It is however very hard to find a guaranteed optimal cover. This is a direct consequence of mutual dependency of mapped subcircuits. Because the mapping of a subfunction depends on the mapping of subfunctions it uses as input but also on the mapping of functions using this subfunction. Also it is very hard to decide which phase of a multiple fan-out signal has to be implemented, such that the overall costs of the cover are minimal. This problem is called 'global phase assignment'. An optimal cover can therefore not be found in restricted time. But using heuristics good suboptimal solutions can be found.

The technology mapper is ordered as follows. First subcircuits in the circuit are selected that can be mapped to library cells. This is done for all signals. These subcircuits can than be matched to library cells. To find the best implementation for a signal all subcircuits of that signal are selected and matched to library cells. Using these matches several good implementations of each signal can be found. This part is called mapping. The cost of an implementation depends on the costs of the implementations of the signals that are used as input of this subcircuit. The selection of subcircuits and the matching of these, is therefore first done for signals at the inputs of the circuit and ended at the output signals. This way all cost information of preceding signals is available when searching for a good implementation of the current signal. Using the mappings of the signals a cover of the circuit is created. At the outputs the best implementation for that signal is available considering the costs of the implementations of its predecessors. The cover is therefore constructed starting at the outputs, selecting the best mapping for these signals. The inputs of these mappings are then the next signals that will be covered. This can repeated until the inputs are reached.

This process can be divided into several subproblems. First the selection of subcircuits. Second matching these subcircuits to library cells. And finally the mapping and covering, constructing a implementation of the circuit using the library cells. These steps are discussed in this order in the next chapters.

Chapter 3

Circuit selection

To find a good implementation of a signal by some library cell, subcircuits of that signal have to be selected. These selected subcircuits will then be used by the matching routine to find a library cell that implements the same combinatorial function as the subcircuit. Out of all these matches the best match will be chosen during the mapping phase.

Because we want to find the best implementation of a signal, all subcircuits of that signal have to be found. To find the subcircuits, the given combinatorial function is constructed out of two input 'and' nodes with possibly inverters on the inputs and the output. The technology mapper will be used for combinatorial logic, therefore the circuit will not contain any recursive structures. The constructed circuit can be seen as a Directed Acyclic Graph. The use of this two input structure can be a restriction. There are tools available that provide a good two input decomposition for technology mapping purposes. For this two input structure a simple and constructive method to select subcircuits can be constructed. The usage of only one basic structure, the 'and' node, simplifies the selection of circuits. In the DAG that is created the subcircuits are selected.

It is possible to use several nodes as basic structure for example: 'and', 'or', 'exclusive or' and inverters. These can all be implemented using a 'and' node with implicit inverters. Only the 'xor' can not be replaced by an 'and' node. However it can be constructed using three 'and' nodes. An advantage of using only the 'and' node is that it can always be implemented directly, in practice every library contains an 'and' cell and an inverter. Therefore it is always possible to find a cover using only 'and' nodes and inverters. It can not be guaranteed that the library contains a 'xor' node, so if the 'xor' is used as a node then it can not be implemented using only one library cell and no cover can be found.

It is common practice to use inverter pairs between two nodes to indicate the possibility to use both polarities of a signal. This increases the number of nodes in the circuit and the complexity of the selection routine, because a single input node is introduced next to all other two input 'and' nodes. The selection routine must now also provide the phase of the input signals of the subcircuit. Therefore it is necessary to have a tight integration of the matching and selection routine, otherwise circuits are selected that are known to be unmappable. We will not use inverter pairs, instead implicit inverters at the inputs and outputs

of the nodes are used. This leaves only one kind of node in the circuit. The polarization of the signals can now be generated by the matching routine which is better equipped for this task.

3.1 Circuit structure

As mentioned earlier the function that has to be mapped is decomposed in a structure of two input 'and' nodes. On both inputs and the output an inverter can be placed, these inverters are implicit. For example an 'or' is implemented as an 'and' node with inverters on inputs and output. The output of these nodes is called a signal. During this decomposition the original structure of the function is kept, no new shared signals are introduced. The function is now captured in one DAG where the nodes represent the signals and the edges represent the connections between the signals.

3.2 Circuit selection in trees

The selection routine searches for a subcircuit in the circuit. The output of the subcircuit is a given signal in the circuit. The inputs of the subcircuit will also be signals of the original circuit. The number of inputs that will be generated must be given. The selected circuits will be used to find library cells that can implement this signal.

For one signal the circuit selection routine will generate all possible subcircuits. Several subcircuits with the same number of inputs can be found. For example, consider the circuit given in figure 3.1. Using the top signal as output, five different subcircuits can be selected all having four inputs. This is displayed in figure 3.2, the encircled nodes here indicate that a signal is used as an input of the subcircuit.

Because of the structure of the circuit, the selection routine is built as an recursive function. It is a constructive algorithm generating all possible subcircuits.

Starting at the top signal in the example, four inputs are requested, see also the numbers in figure 3.2. It is only possible having this number of inputs if there are three inputs at the left and one at the right, vice versa, or two left and two right.

The routine can now call itself recursively for the left and right predecessor requesting $n - i$ and i inputs, where i ranges from one to three. Of course it is not possible to request more than one input from a signal if that signal is an input of the main circuit. Under the assumption that a signal has either two predecessors or is an input of the circuit, the selection routine for trees would roughly look like the following pseudo C++ code:

```
int circuitSelection(Signal *s, int nr_inputs)
{
  if (nr_inputs == 1)
  {
```

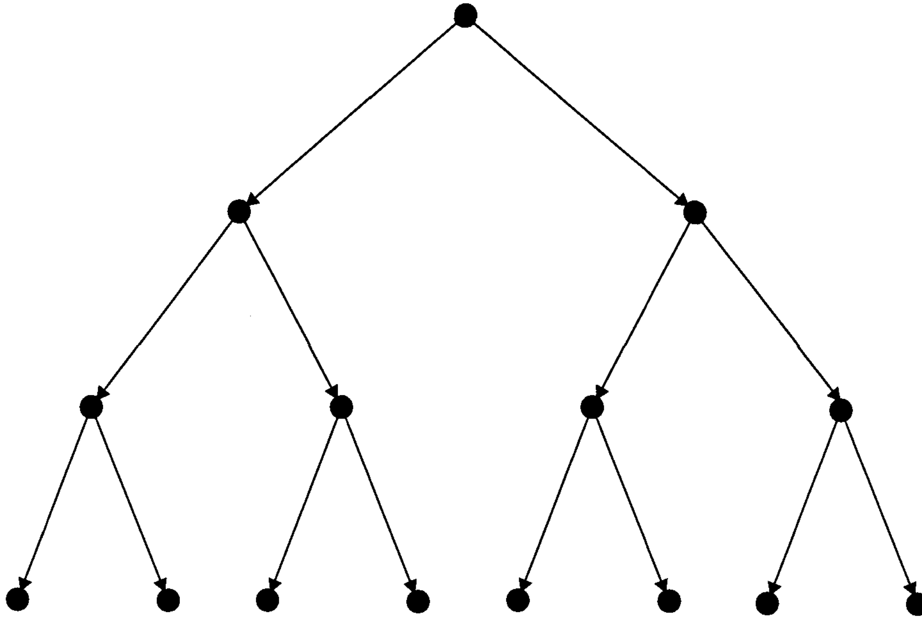

Figure 3.1: Example circuit

```
    if (s->isLockedAsInput())  // a previous run selected this
                               // signal as an input
    {
      s->unLockAsInput();
      return no_subcircuit_found; // unsuccesfull finding 1 input
    }
    s->lockAsInput();          // signal will be used as input
    return 1;                  // of the subcircuit to be mapped
  }

  if (s->isInput())            // an input can only generate zero
                               // or one input
    return no_subcircuit_found;

  for (int i = 1; i < nr_inputs; i++)
  {
    while (leftSubcircuitIsSelected(s))  // previous circuit
                                         // selection at left
                                         // side succeeded
      while (circuitSelection(s->rightPredecessor(),
                             nr_inputs - i) == nr_inputs - i)
        return nr_inputs;
    circuitSelection(s->leftPredecessor(), i);
  }
}
```

Of course it is necessary to remember the counter i between successive calls,

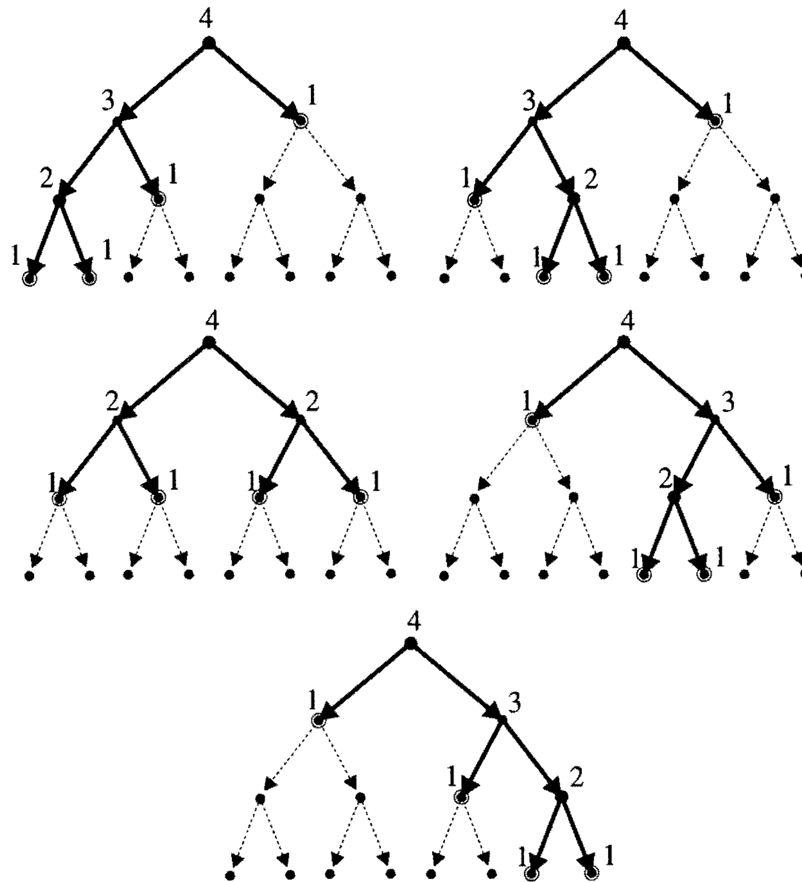

Figure 3.2: All four-input subcircuits of figure 3.1

otherwise the same circuit is selected over and over again.

The function is called several times for the same signal until no more subcircuits can be generated. Then it can be called again for a different number of inputs. All possible subcircuits for several number of inputs can this way be generated.

As one can imagine the number of possible subcircuits increases very fast for increasing number of inputs. For one signal $n-1$ partitions of the inputs are tried, where n is the number of inputs requested of this input. If n is two, only one partition is possible. For every partition all subcircuits on the left are generated for all subcircuits on the right. So the number of generated subcircuits $f(n)$ is equal to:

$$\begin{aligned} f(n) = & \sum_{i=1}^{i=n-1} f(i)f(n-i) \\ f(2) = & \; 1 \\ f(1) = & \; 1 \end{aligned} \tag{3.1}$$

For five inputs this results in 14 subcircuits. For eight inputs already 429 subcircuits can be found. So it is important to limit the number of inputs for which subcircuits have to be found.

3.3 Circuit selection in DAGs

This routine discussed so far will only generate all subcircuits if the circuit is a tree, that is the fan-out of a signal is zero — an output — or one. In general the circuit will be a DAG where the fan-out of a signal is larger than or equal to zero. If a signal is used as an internal signal for a match, then the signal can not be reached from outside this match. So multiple fan-out signals used as internal signals have to be implemented more than once. We assume that it is inefficient to implement a signal twice. Therefore we will not use those signals as internal signals and will always lock them as input.

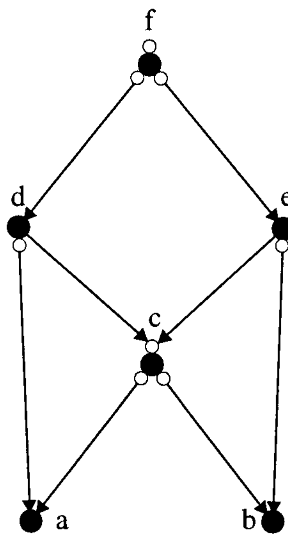

Figure 3.3: Example of a reconvergent multiple fan-out circuit

Consider the circuit in figure 3.3 — the small open circles are implicit inverters, mostly not drawn because they are irrelevant for the selection. This circuit implements an exclusive 'or' function. The signal 'c' has a fan-out of two. If this signal is marked as an input, then the resulting mapping would be something like figure 3.4, even if the library contains an 'xor' cell. The use of an 'xor' cell however would most probably result in lower costs for this mapping.

It is important to notice that the multiple fan-out of the signal 'c' is restricted. The successors are within the cone that lies with the top at signal 'f' and stops at the signals 'a' and 'b'.

If instead of just locking a multiple fan-out signal as input, the number of locks is counted, it can easily be checked whether the fan-out is reconvergent. If the number of locks set is equal to the size of the fan-out, then the fan-out is reconvergent and the signal can be used as an internal signal. The circuit selection routine can now continue down from this signal requesting one more input — the first lock was set by another successor asking one input — than originally requested of this signal.

The first lock on a multiple fan-out signal can only be set if one input is requested by a successor, because this signal will at the start always selected as an input. The next time inputs are requested of this signal by other successors,

Figure 3.4: 'Tree'-mapping of figure 3.3

it can only generate zero inputs, because it is already in use as input. If the number of inputs requested is indeed zero, then the number of locks is increased. If however the number of locks is now equal to the fan-out — a full lock —, then during this last call, more inputs can be requested because the selection can be continued at this signal. Outputs of the circuit can never be used as internal nodes. Therefore they have been given a fan-out that is one larger than the number of successors in the circuit, because there will always be at least one external connection. This way they can never have a full lock.

Using this setup it is possible to request zero inputs from a signal. The selection that finds the complete 'xor' in figure 3.3 is displayed in figures 3.5 and 3.6. The encircled signals are used as input, the signal pointed to is the signal now active in the selection function.

The selection routine for DAGs, but also for trees which inputs are used multiple times, is described in pseudo code down below. Signals that initially have to be used as inputs — inputs of the circuit and multiple fan-out signals — are called leaves.

```
int generateCircuit(Signal *s, int nr_inputs)
{
  if (s->isLeaf() && !s->hasFullLock())
  {
    if (s->isLocked())
      switch (node_count)
      {
        case lock_set_and_is_input: // the previous recursion set
                                    // a lock at this signal and
                                    // selected it as input
          s->removeLock();
```

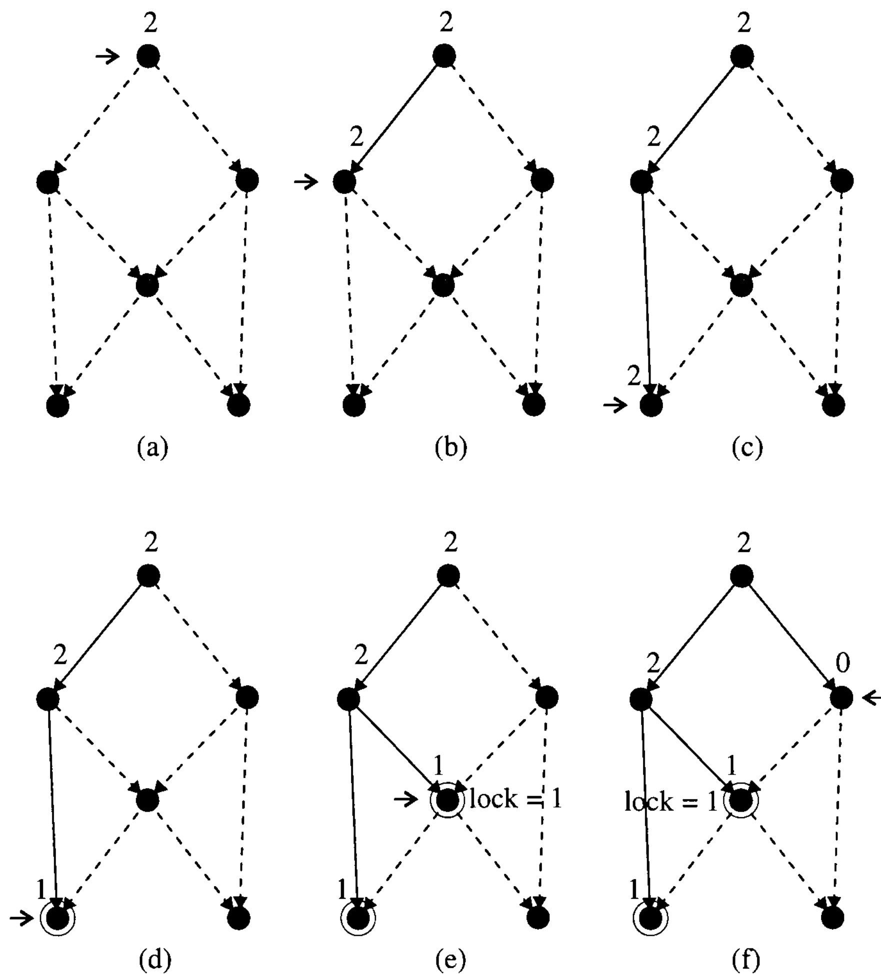

Figure 3.5: Steps of selection routine selecting 2 inputs in figure 3.3, part 1

```
    return no_subcircuit_found;
  case empty:                 // first time this signal is
                              // reached in this part of the
                              // recursion
    s->addLock();             // add lock to signal
    if (nr_inputs == 0)       // if zero inputs were requested
    {
      node_count = s->hasFullLock() ? continue_down : 0;
                              // if the signal now has a full
                              // lock then next time continue
                              // downwards
      return 0;
    }
    else
      if (!s->fullLock())     // more than zero inputs were
                              // requested if signal now does
                              // not have a full lock then
```

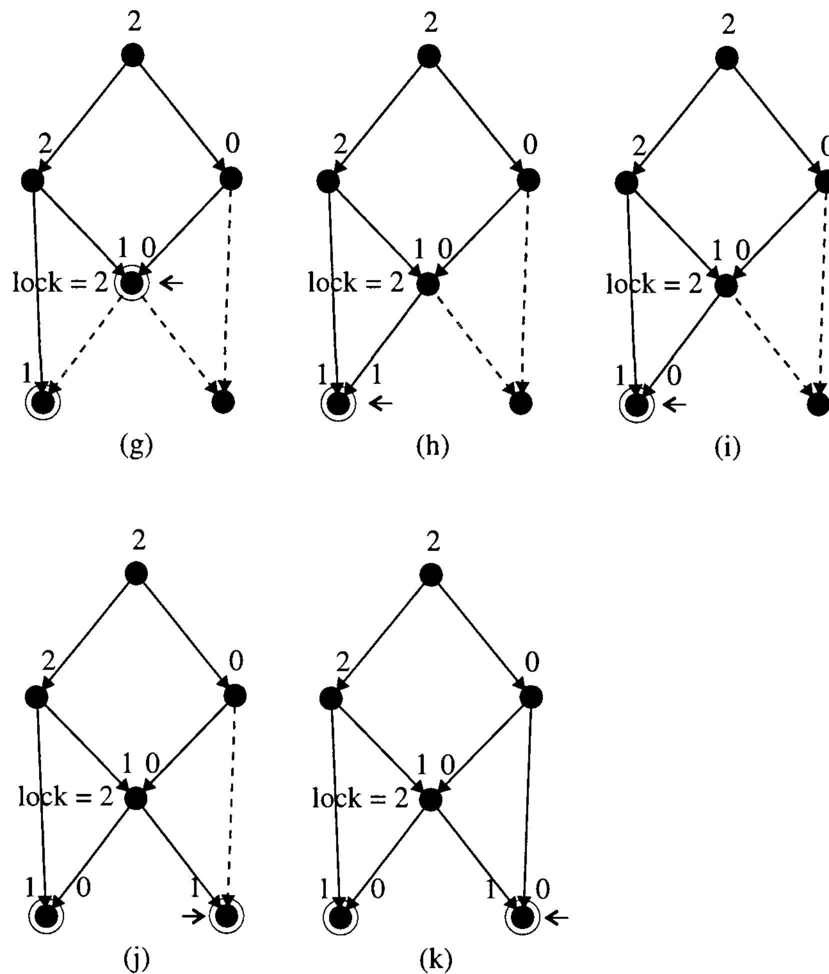

Figure 3.6: Steps of selection routine selecting 2 inputs in figure 3.3, part 2

```
                              // request failed otherwise
                              // continue downwards
        {
          s->removeLock();
          return no_subcircuit_found;
        }
        break;
      case 0:                 // the previous recursion locked
                              // this signal
        s->removeLock();
        return no_subcircuit_found;
    }
  else
  {                           // signal is not locked
    if (nr_inputs != 1)       // only one input can be requested
      return no_subcircuit_found;
    node_count = lock_set_and_is_input; // one input is requested
```

```
                                // so lock the signal and thereby
                                // use select it as input
      s->addLock();
      return 1;
    }
  }
  else
    if (node_count == empty && nr_inputs == 1)
    {
      node_count = lock_set_and_is_input; // single fan-out signal
                                // and one input is requested,
                                // select it as input, next time
                                // continue downwards
      s->addLock();
      return 1;
    }

  if (s->fullLock())            // continue downwards, therefore
                                // remove selection of use as
                                // input
  {
    s->openLock();
    nr_inputs++;
  }

  if (! (node_count >= 0))      // if counter has a special value
  {
    if (node_count == lock_set_and_is_input)
      s->removeLock();          // remove input selection and
    node_count = 0;             // start generating inputs at
    generateCircuitLeft(s, nr_inputs);   // the left
  }

  while (node_count <= nr_inputs) // while not all permutations are
                                // tried
  {
    while (leftSubcircuitIsSelected(s))
    {
      if (generateCircuitRight(s, node_count) == node_count)
        return (s->fullLock()) ? nr_inputs - 1 : nr_inputs;
                                // if signal has a full lock then
                                // the predecessor requested one
                                // input less than now is generated
      generateCircuitLeft(s, nr_inputs - node_count);
    }
    node_count++;               // try next permutation
    if (node_count <= nr_inputs)
```

```
      generateCircuitRight(s, nr_inputs - node_count);
  }

  if (s->fullLock())
    s->removeLock();
  return no_subcircuit_found;
}
```

In this routine `node_count` is a counter that is kept in a separate structure. `node_count` replaces the counter `i` in the selection routine for trees. A tree of such counters is build. This tree is used to remember the counter values which represent the state of previous recursion. The tree has the same form as the circuit's DAG at the signal that is used as root. Multiple fan-out signals have however several counter nodes. This is necessary because every part of the recursion has to remember its state. The correct counter in the tree is provided by the functions `generateCircuitRight()` and `generateCircuitLeft()`. Next to providing the correct counter these functions call `generateCircuit()` again with the appropriate predecessor of the signal.

The improvement that this use of reconvergence provides will only improve the covering results if such structures can be found in the circuit. If the circuit does not contain any multiple fan-out signals which reconverge in a small area then the covering results show no improvement. The computation time will increase because of the additional conditions and increased search space.

3.4 Further extension

The DAG selection routine is a constructive routine. Except for some special structures all subcircuits with a given number of inputs are recognized. An example of such a exceptional structure is given in figure 3.7. The routine will never mark the signals 'a' and 'b' as input, but will stop at the signals 'c' and 'd'. It will try to select the signals 'a' and 'b' from both successors, requesting one input. Both times this request will fail because two inputs are needed. To reach the signals 'a' and 'b' the routine would have to continue from both 'c' and 'd' at the same time, behaving like a wave.

To solve this problem one could first search for the input cone of a signal. This can easily be done by a depth-first search algorithm. Signals with a non reconvergent multiple fan-out, will be selected as an input. Reconvergent fan-out signals, like 'c' and 'd' where the DAG selection routine in figure 3.7 now stops, will not be marked as input. The selection routine can now try to continue until it reaches a marked signal, stepping beyond the signals where it now stops.

Structures like the one in figure 3.7 however are rare. It is very unlikely that they will be created by the logic optimization and decomposition tools that provide the input circuit. The additional effort that is needed to recognize these structures is therefore superfluous. This extension is therefore not implemented.

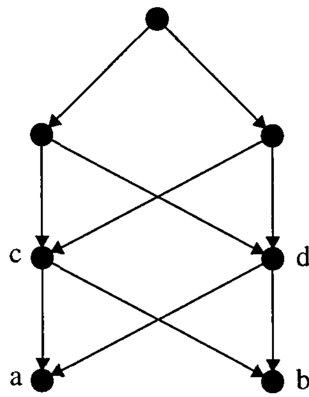

Figure 3.7: Not recognized 2-input DAG structure

Chapter 4

Comparing boolean functions

After the selection of a subcircuit, cells in the library have to be found that can implement the combinatorial function of the subcircuit. To find these cells, their boolean function must be compared to that of the selected circuit.

4.1 Representation of boolean functions

For comparing the functions, a representation of the boolean function is needed. The truth table of the functions is useful representation and will be used here. This table is stored as a bit vector. One bit vector is created for the output signal of the function.

c	b	a	$a\bar{c} + bc$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

msb lsb

1	1	0	1	1	0	0	0

Figure 4.1: Bitvector as representation of boolean functions

A bit vector is an array of integers where every bit represents one output value in the truth table. For n inputs 2^n bits are needed. On a computer using 32 (2^5) bit integers this results in an array of 2^{n-5} integers, the minimum size is of course 1 integer. Most cells in a library have not more than six inputs. For these sizes a bit vector is very memory efficient, in only $2^{6-5} = 2$ integers the complete truth table can be stored. A bit vector of this size is also efficient for computing boolean operations.

One of the operations that will be needed in chapter 4.3 is the calculation of the boolean difference, $\frac{\partial f(x)}{\partial x_i}$, where $x = (x_1, x_2, ..., x_n)$. The boolean difference is defined as:

$$\frac{\partial f(x)}{\partial x_i} = f(x)|_{x_i=1} \oplus f(x)|_{x_i=0} \tag{4.1}$$

Using bit vectors this can be calculated using only the original truth table. The function $f(x)|_{x_i=0/1}$ is called the negative/positive cofactor. The truth table of a cofactor is half the size of the original table. In further calculations it is easier to have a table of the same size. Therefore a table is created where the cofactor is still a function of x_i but is independent of it, the function does not change when x_i changes.

When a bit vector is shifted downwards over 2^{i-1} bits then the output results of $f(x)$ for x_i is true are moved to x_i is false. This is only correct if $\forall_{x_i=0} f(x) = 0$. The shifting of the vector will then result in $f(x)|_{x_i=\overline{x_i}}$. Example:

$a, b, c:$	111	110	101	100	011	010	001	000
$f(x) = ab + \overline{a}c$	1	1	0	0	1	0	1	0
a	1	1	1	1	0	0	0	0
$f(x) \cdot a = ab$	1	1	0	0	0	0	0	0
$(f(x) \cdot a) \gg 4$	0	0	0	0	1	1	0	0
$(f(x) \cdot a)\|_{a=\overline{a}} = \overline{a}b$	0	0	0	0	1	1	0	0

Likewise a vector can be shifted upwards, under the restriction that $\forall_{x_i=1} f(x) = 0$. An upwards shift operation of f over k bits is indicated as $f(x) \ll k$ and downwards as $f(x) \gg k$.

The cofactors of $f(x)$ can now be generated as follows:

$$\begin{aligned} f(x)|_{x_i=0} &= (f(x) \cdot \overline{x_i}) + ((f(x) \cdot \overline{x_i}) \ll (i-1)) \\ f(x)|_{x_i=1} &= (f(x) \cdot x_i) + ((f(x) \cdot x_i) \gg (i-1)) \end{aligned} \tag{4.2}$$

Here x_i is a truth table of the function $f(x) = x_i$ with the same bit vector size as $f(x)$. Example:

$a, b, c:$	111	110	101	100	011	010	001	000
$f(x) = ab + \overline{a}c$	1	1	0	0	1	0	1	0
$f(x) \cdot a$	1	1	0	0	0	0	0	0
$(f(x) \cdot a) \gg 4$	0	0	0	0	1	1	0	0
$(f(x) \cdot a) + ((f(x) \cdot a) \gg 4)$	1	1	0	0	1	1	0	0
$f(x)\|_{a=1} = b = ab + \overline{a}b$	1	1	0	0	1	1	0	0

4.2 Input signatures

The bitvector can now be used for comparing the functions of the selected subcircuit and that of the library cell. When matching a subcircuit to a cell one does not know which input of the subcircuit corresponds to which input of the library cell. Because the phase of the signals may be changed, it is also possible that an input or output inverter is needed to make the match.

If all possibilities are checked this results in $n! \cdot 2^{n+1}$ comparisons. For larger circuits this can take a long time. If the inputs can be given a signature that is independent of the input permutation and sign then the number of possibilities can be decreased.

Several signatures have been evaluated. The method discussed in [Kap95] does not use input signatures but signatures of the complete function for all input permutations. Comparing functions is simplified but the signature itself is not input sign independent. This all results in a large amount of data to specify a library function. The signature that will be used is a simple and small variant of the one mentioned in [Moh93].

The signature of an input is the minterm count of the positive and negative cofactor of that input, $(\mid f(x)|x_i = 0 \mid, \mid f(x)|_{x_i=1} \mid)$, as discussed in [Moh93]. The count is denoted as $\mid f(x) \mid$. It is obvious that $\mid f(x)|_{x_i=0} \mid + \mid f(x)|_{x_i=1} \mid = 2 \mid f(x) \mid$. So if the minterm count of one cofactor is known then the count of the other can easily be derived. That is why the signature contains only the minimum of both counts for every input plus an indicator which one is used:

$$\begin{aligned} signature(x_i, f(x)) = & \ (min\{\mid f(x)|_{x_i=0} \mid, \mid f(x)|_{x_i=1} \mid\}, \ indicator(x_i, f(x))), \\ indicator(x_i, f(x)) = & \begin{cases} 1 & if \ \mid f(x)|_{x_i=0} \mid > \mid f(x)|_{x_i=1} \mid \\ 0 & otherwise \end{cases} \end{aligned} \quad (4.3)$$

4.3 Symmetric inputs

For most functions several inputs can be swapped without changing the result of the function. Inputs are called symmetric if:

$$f(x_1, ..., x_i, ..., x_j, ..., x_n) \equiv f(x_1, ..., x_j, ..., x_i, ..., x_n).$$

This will from now on be denoted as:

$$f(x_{ij}) \equiv f(x_{ji})$$

If these symmetric inputs are joined in classes then the search for input permutations can be reduced to the search for symmetry class permutations. This reduces the search space significantly. Inputs can also be anti-symmetric, they can be exchanged but then they have to be inverted.

$$f(x_1, ..., x_i, ..., x_j, ..., x_n) \equiv f(x_1, ..., \overline{x_j}, ..., \overline{x_i}, ..., x_n).$$

Because the phase of a signal can be changed it is convenient to add anti-symmetric inputs to the same class.

When all input signatures of a function are known, the inputs are joined in symmetry classes. Inputs are only checked for (anti-)symmetry if the first part of their signatures is equal. If the indicator parts of the signatures are equal then the inputs can only be symmetric, otherwise they can only be anti-symmetric.

To check for symmetry the following statement can be used:

$$\left(\left(\frac{\partial f(x)}{\partial x_i} \oplus \frac{\partial f(x)}{\partial x_j}\right) \cdot x_i \cdot x_j \equiv 0\right) \equiv \left(f(x)|_{x_i x_j} \equiv f(x)|_{x_j x_i}\right). \quad (4.4)$$

Proof:

$$\forall_{f(x)} \exists_{f_a, f_b, f_c, f_d}\ f(x) = f_a x_i x_j + f_b \overline{x_i} x_j + f_c x_i \overline{x_j} + f_d \overline{x_i}\,\overline{x_j}$$

f_a here is short for $f_a(x_1, ...x_{i-1}, x_{i+1}, ..., x_{j-1}, x_{j+1}, ...x_n)$, the same counts for f_b, f_c and f_d. If x_i and x_j are symmetric then:

$$\begin{aligned} f(x_{ij}) &= f_a x_i x_j + f_b \overline{x_i} x_j + f_c x_i \overline{x_j} + f_d \overline{x_i x_j} \\ f(x_{ji}) &= f_a x_j x_i + f_b \overline{x_j} x_i + f_c x_j \overline{x_i} + f_d \overline{x_j x_i} \\ \left(f(x)|_{x_i x_j} \equiv f(x)|_{x_j x_i}\right) &\Rightarrow (f_b \equiv f_c) \end{aligned}$$

Expanding formula 4.4 gives:

$$\left(\frac{\partial f(x)}{\partial x_i} \oplus \frac{\partial f(x)}{\partial x_j}\right) \cdot x_i \cdot x_j = (f_a(f_b \oplus f_c) + \overline{f_a}(f_b \oplus f_c)) x_i x_j$$

This is equal to zero for all x_i and x_j if and only if:

$$f_b \equiv f_c$$

□

If they are anti-symmetric then the following counts, for which the same proof can be given:

$$\left(\left(\frac{\partial f(x)}{\partial x_i} \oplus \frac{\partial f(x)}{\partial x_j}\right) \cdot x_i \cdot \overline{x_j} \equiv 0\right) \equiv \left(f(x)|_{x_i x_j} \equiv f(x)|_{\overline{x_j x_i}}\right). \quad (4.5)$$

The symmetric and anti-symmetric inputs are joined in symmetry classes. These classes are disjunct and the union of these classes hold all inputs of the function. This results from the fact the symmetry is an equivalence relation.

These symmetry classes can also be given a signature, namely the first part of the signature of the inputs in the class. Using this signature it can easily be checked whether a symmetry class in one function is equivalent to a symmetry class in the other function. If the classes are equivalent then they must also hold the same number of inputs.

Using these symmetry classes, the worst case number of comparisons is reduced to $c! \cdot 2^{c+1}$, where c is the number of symmetry classes. Mostly this number will not be reached because of the signatures of the classes.

4.4 Comparing functions

Using the information of the symmetry classes the functions can be compared fast. But before the input correspondence is searched some other data can be checked.

Only functions with the same number of inputs are compared. It is assumed that the matching to functions with a larger number of inputs where some inputs are constants or joined is not desired. If necessary these functions with constant or joined inputs can be added to the library as a preprocessing step.

The same idea as used for the input signature can be used as a simple signature of the complete function. Functions can only be equal if their minterm count is equal. If $| g(x) |= 2^n - | f(x) |$ then an output inverter is needed. If $| f(x) |= 2^n - | f(x) |$ then a match without and with an output inverter should be tried.

After these first checks, the symmetry classes are matched. If the number of classes is not equal or some classes can not be matched to those of the other function then the functions are not equal. Only classes with the same signature and number of inputs can be equal. The number of possible class permutations is limited by the number of classes with the same signature. Classes where the signature of the first input does not define the sign of the input — the minterm count of the positive cofactor is equal to that of the negative cofactor — are tried twice, once normal and once inverted. If the first gives a match then the second does not have to be tried.

The inputs in the symmetry classes can be matched directly to those of the other function in the corresponding class. First inputs with the same setting of the symmetry indicator are matched. If this is not possible any more then the inputs are matched using an input inverter. This way a match with minimal number of input inverters is created.

Now the functions can be checked for equivalence. If they are not equivalent then another symmetry class permutation is tried or classes, where the minterm count of positive and negative cofactor of the inputs is equal, are inverted. This is continued until every possibility is tried or the functions with the found input permutation and negations are equivalent.

4.5 Comparing in practice

It seems that comparing functions is still a lot of work, but considering that most library functions have several symmetric inputs and the number of inputs is limited, only a small number of symmetry classes have to be matched. For the *MCNC* library the average number of possible permutations and inversions of classes is 1.7. The average number of input permutations within a symmetry class is 30 and they are all logically equivalent. Even when all possible input permutations are tried only 52 matches are possible. Compare this to the average number of permutations and inversions of inputs without any additional information, which is 7517, and these are not all equivalent. As is obvious the gain of using signatures and symmetry classes is significant.

4.6 Other types of symmetry

It must be noted that the symmetry classes used here do not cover all cases. Consider the following function: $f(a, b, c, d) = ab + bc + cd + da$. One cannot ex-

change any two variables without changing the function. But if the order of the variables is rotated, then the function stays the same: $f(a,b,c,d) \equiv f(d,a,b,c)$. This can be useful information, but the functions that have this kind of symmetry are not very common. None of the functions in the *MCNC* library show rotational symmetry. So the addition of this kind of symmetry classes does not increase the efficiency, but one should keep in mind that the used symmetry classes do not provide all symmetry information about a function.

Chapter 5

Mapping and covering

The previous chapters discuss methods to select subfunctions and compare them to library functions. All ingredients to cover a function are now available. The goal of mapping is to find a match between subfunctions and library cells, while minimizing the cost of the implementation. The union of the mapped signals should then create a cover of the function using several library cells.

When searching for a good mapping of a signal the costs of the mapping of the inputs that this mapping uses also determine the cost of this mapping. Because of this a cheap library cell is not always the best. An 'and' port is always most efficient for implementing an 'and'. If the mappings of the inputs that are used, are very expensive it can be better to use another match that uses other inputs. As a result of this accumulation of costs the circuit has to be mapped from inputs to outputs.

5.1 Mapping of subcircuits

First an index of the signals in the circuit is constructed. The order of the index is defined such that the index of all predecessors of a signal is lower than the index of the signal itself. So inputs will be at the head of the index and outputs at the end. Traversing through the index from bottom to top guarantees that one reaches an signal after all it predecessors. If the mapping of signals is done in this order all mapping information of signals of lower level is available when mapping a signal. The inverse order can not be used because the cost of a mapping of a subfunction mainly depends on the costs of the mapping of its inputs.

Every signal in the function is mapped to a set of library functions. This set can be divided into functions providing the positive phase and functions providing the negative phase of the signal. This is necessary because it is not known whether the successors of this signal need the positive or negative phase. If one only tries to optimize for area, only one mapping of both phases is needed because different drive capabilities or delays are not taken into account. But when optimizing for delay or power it can be necessary to have more mappings of the same phase. Each mapping is then the most cost efficient for a certain load on this signal. This is called 'load binning'. During the covering phase,

the load that the successors impose is known and a mapping can be selected for that load.

In the latter case it may be necessary to check several input permutations during the matching. Because logically equivalent inputs can have different capacity or delay, all possible permutations have to be tried, to find the match with the lowest cost. It may seem that the use of the matching optimization loses its value here. But considering section 4.5, the number of matches is still significantly lower than without the use of a specialized matching routine.

To generate this set of mappings for a signal all possible subcircuits are selected. The number of inputs of the subcircuits ranges from two to five. Subcircuits with only one input can only be inverters or buffers. The inverters are used implicitly in the cover. Buffers are not used, because they do not decrease the cost when optimizing for area. Examining the *MCNC* test library one notices that most functions have five inputs or less. The only exceptions is a small minority of some six input cells. The chance that a match to one of those is found, is small. To find and match all subcircuits with six inputs there is also additional computation time required, because the number of subcircuits increases very fast for a increasing number of inputs. This can be seen clearly in the test results of section 6.3. Therefore the mapping is restricted to five inputs. This number can always be increased, but a increase in computation time should be expected. And the cost decrease will be less than linear compared to the extra time needed.

All the subcircuits of a signal are matched to the library functions. Depending on the final goal, the best or several bests matches for both positive and negative phase will be stored. As said before, when optimizing for area only one match of both phases has to be stored. When optimizing for power or delay it can be an advantage to have several different mappings available with different drive capability. When the load that the successors impose is known, the best mapping out of this set can be selected.

The cost of a mapping is the cost of the library cell itself plus the costs of the mappings of the signals used as input. The cost of these mappings can not be known because the best mapping out of the set can not be chosen yet. For the moment the cheapest are chosen for cost calculation.

Also the load imposed by the successors on the signal is not known so the cost can not be calculated exactly. An educated guess of the load is needed. For area optimization this does not pose any problems, because the load is not part of the cost function.

Because the costs of the current mapping influences the mappings chosen for the successors that use this signal as input, it is important that the costs are representative. So no large faults may be introduced when calculating the costs that the mappings of the inputs of this mapping impose.

5.1.1 Mapping multiple fan-out signals

For multiple fan-out signals it is more difficult to reduce the number of faults in the cost calculation. If the successors chose a different mappings out of the set of this signal, then the costs of some successors will not be correct. In the

final cover this signal will only be implemented once, so only one mapping will be correct. The costs that some successors have used is therefore incorrect.

This is an important problem while mapping. A solution is using just one mapping out of the set for cost calculation by successors. The selection of the mapping used will still have to be done during the covering, whether just one or several mappings are used during the cost calculation. One can not know what the influence is of a difference between final and used mapping on the mapping of successors.

To correct this problem, the mapping can be done iteratively. After the covering, one knows what mapping most probably will be used. To find a better cover the mapping can be started again. This time it can be started at the multiple fan-out node instead of the inputs of the circuit. Now the mapping that was used in the cover should be used for cost calculation. Of course this iterative process increases the computation time of a cover.

It is however possible that the mapping and covering will never stabilize. Therefore the iterative process should only be done if the cost difference of the mappings is significantly large. The technology mapper discussed here does not incorporate such an iterative process.

5.1.2 Mapping of multiple output library cells

Multiple output cells will not be found because the circuit selection and matching routine are not equipped for this task. Suppose that other tools are available that can find these cells. Then they can be used as a preprocessing step. These tools can search the circuit for the use of multiple output cells. For signals where such a cell can be used, the cell can be added to the mapping set. During the covering it is then possible to choose between the single output cells found and the added multiple output cells.

Clearly the mapping of a circuit is a very exhaustive search. For all signals, all subcircuits are generated and matched to all library cells. For all matches that succeeded the costs are calculated. From those the cheapest of each phase is chosen. This exhaustive search is very important for finding a good cover. To keep the routine fast it is necessary that the subcircuit selection routine and the matching routine do their work very fast. This is why a good and fast selection and matching routine were needed.

5.1.3 Example of a mapping

An example of the mapping of a circuit, optimizing for area, is given in figure 5.1.

In the two input DAG every signal will be mapped. All possible subcircuits of that signal are selected. These subcircuits are then matched to library cells. The cheapest implementation of each phase will be added to the mapping set. Using all these mappings, a cover will later be generated. The library used is given in table 5.1.

Working from inputs to outputs, the mapping is started at signal 'f'. This signal can be mapped to an 'and' port. The inverse of 'f' can be mapped to an 'or' port with inverters on both inputs. Both will be added to the set of

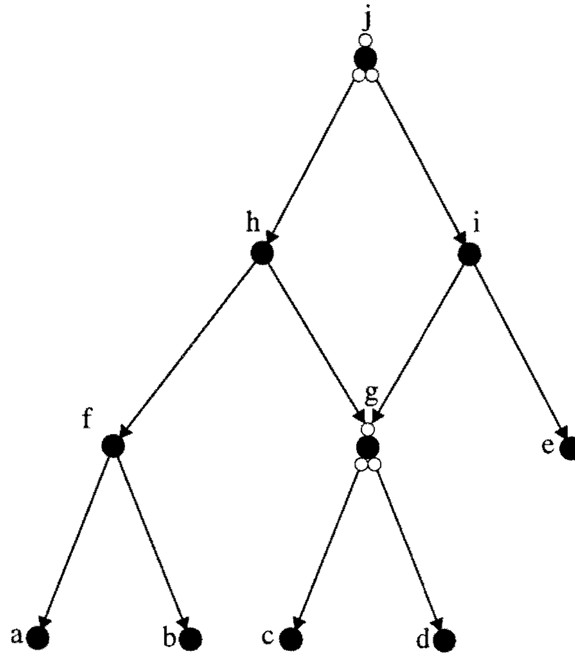

Figure 5.1: Mapping an example circuit

Name	function	size
inv	$\overline{a}$	1
and2	ab	4
and3	abc	7
or2	$a+b$	7
oai22	$\overline{(a+b)(c+d)}$	16

Table 5.1: Library for mapping example

mappings. The same can be done for the signals 'g' and 'i'. For signal 'h' two mappings for the positive phase are possible, an 'and2' and an 'and3' port. The one with the lowest cost, the 'and3' port, will be added to the set. The same is true for signal 'j'. The 'oai22' port is added to the set of this signal. All mappings found are displayed in figure 5.2 using accentuated ellipses. The costs and the mappings that are used, are displayed in table 5.2. The mapping set contains one mapping for the positive phase and one for the negative. This also demonstrates the use of the reconvergence checking discussed in section 3.3. The 'oai22' mapping of 'j' includes the multiple fan-out signal 'g'. One also sees that sometimes it is cheaper to use the positive phase plus an inverter if the negative phase is needed, as is done for 'f' at the 'or2' mapping of 'h'.

Even in this small circuit one can see that some false decisions can be made. If both $\overline{i}$ and h are implemented, the mapping they chose for g is different. One chooses $\overline{g}$ and the other g. Because only one mapping will be implemented, at least one cost calculation of $\overline{i}$ and h will be incorrect. This indicates why it is hard to find an optimum cover. In practice however the costs of the

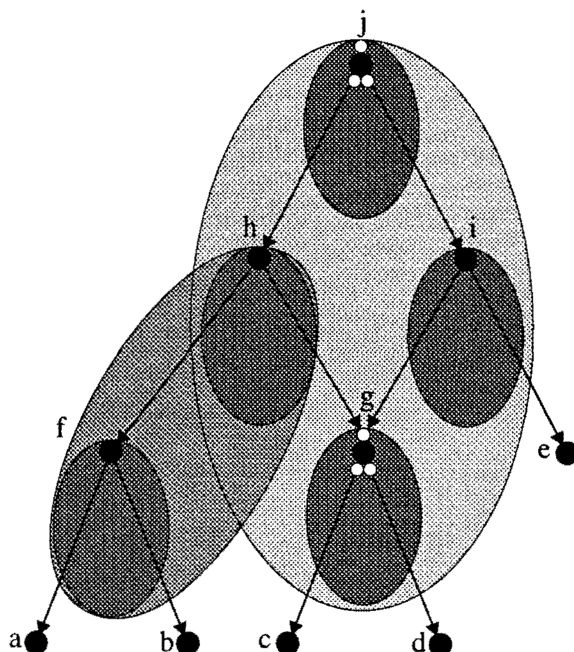

Figure 5.2: All mappings of example circuit in figure 5.1

different mappings of the multiple fan-out signal do not differ much. This is also demonstrated in the example by signal 'g'. It is not possible to guarantee that an optimum cover will be found. But the cover found will be near the optimum.

The problem in the case of area optimization, is a problem of global phase assignment. It can be proven that this is a NP-hard problem [Wan89] and can not be solved in short time. For other cost functions where input load is important this problem is even more difficult.

5.2 Covering the circuit

When all signals are mapped, a cover has to be created. If the circuit is tracked again, but now from outputs to inputs, the cover can easily be found. The mapping at the outputs is the cheapest found mapping for this signal considering the mappings of the predecessors. This mapping is the start of the cover. It uses some signals as inputs. If these signals are not inputs of the circuit itself, then a mapping has to be selected for these signals. This can be continued until all inputs of the circuit are reached and no part of the circuit is not covered.

For this backtracking the same index can be used as during the mapping phase, but this time we start at the signal with the highest index and end at the one with the lowest. This index namely also guarantees that the index of a signal is lower than the index of all its successors.

While traversing the circuit from outputs to inputs, all information of the successors is available, the load they impose and the phases they need. So the best mapping out of the set of mappings for some signal can be selected.

Signal	library cell	mapping	costs	mapping set
f	*and2* *or2*	$f = (ab)$ $\overline{f} = (\overline{a} + \overline{b})$	$and2 = 4$ $or2 + (inv) + (inv) = 11$	(and2, or2)
g	*or2* *and2*	$g = (a + b)$ $\overline{g} = (\overline{a}\ \overline{b})$	$or2 = 7$ $and2 + (inv) + (inv) = 8$	(or2, and2)
i	*and2* *or2*	$i = (ge)$ $\overline{i} = (\overline{g} + \overline{e})$	$and2 + (g) = 11$ $or2 + (\overline{g}) + (inv) = 17$	(and2, or2)
h	*and2* *and3* *or2*	$h = (fg)$ $h = (abg)$ $\overline{h} = (\overline{f} + \overline{g})$	$and2 + (f) + (g) = 15$ $and3 + (g) = 14$ $or2 + (inv + f) + (\overline{g}) = 21$	(and3, or2)
j	*or2* *and2* *oai22*	$j = (h + i)$ $\overline{j} = (\overline{h}\ \overline{i})$ $\overline{j} = (\overline{(f + e)(c + d)})$	$or2 + (h) + (i) = 32$ $and2 + (inv + h) +$ $(inv + i) = 33$ $oai22 + (f) = 20$	(or2, oai22)

Table 5.2: Mapping results for example circuit

For area optimization only the phase information is needed. In the designed mapping routine only one foregoing mapping decides which phase will be implemented. It is however an important consideration to first check all foregoing mappings for the phase they need and then decide which mapping will be implemented.

From the chosen mapping the signals in the circuit that are used as input can be derived. For these signals the best mapping can again be selected. This is continued until the inputs of the circuit are reached.

In the process a lot of signals in the circuit are skipped. They have become internal signals. It may seem that the mapping of these signals was useless. The mapping of these signals is still necessary, because it can not be foreseen that these signals will be removed. These mappings are needed to find the best mappings for successors. Otherwise it is not possible to compare the costs of several mappings for one successor.

If the exact costs of the covering are needed, they must be calculated again. During the mapping some guesses were needed for cost calculation. The costs calculated at the output signals will mostly not be correct. But now the complete circuit is covered, the costs can be calculated exactly. If the costs are higher than what is needed then the circuit should be mapped and covered again. This time however one knows what choices will be done so the number of guesses during the mapping can be decreased. Also mappings that are now known to cause high costs can be removed.

5.2.1 Covering the example circuit

The covering process will be demonstrated on the example of section 5.1.3. The covering is started at the output 'j'. The positive phase is needed as it is an output, but the cheapest mapping for node 'j' is the negative phase plus an inverter. The cost of this implementation is $20 + 2 = 22$. Using this mapping, the signals 'h', 'i' and 'g' are skipped. The next signal to be covered is 'f', the

signals 'c', 'd' and 'e' do not have to be covered because they are inputs of the circuit. For 'f' the positive phase is needed and can best be implemented using the positive phase mapping. If the costs are calculated again, one sees that the costs calculated during the mapping process were correct. This is because the covers do not use a multiple fan-out signal as input. The result of the covering can be seen in figure 5.3.

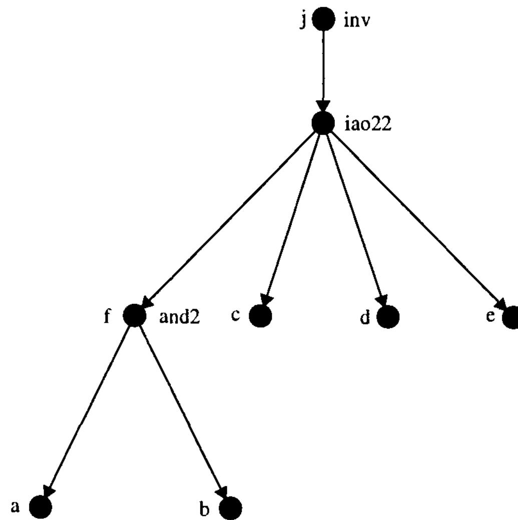

Figure 5.3: Cover of example circuit

Chapter 6

Experimental results

To check the results generated by the technology mapper that is designed and described, some tests have been performed. The results have all been checked for logical equivalence with the original function. The technology mapper designed has been given the internal name R_aCE.

The results given in this chapter use the *LGSynthesis '91* benchmark circuits. The library used is the *MCNC* library published together with the benchmark circuits. These circuits were chosen because several other articles used them as benchmarks. R_aCE is the first functional technology mapping routine for non-complete libraries written at *EUT*, so some comparison with other mapping routines is necessary. The results are compared to *SIS* 1.3 and *MARS* [Kap95]. The computations were run on a *HP 9000/735/90*. The circuits mapped are the same as those in [Kap95]. This was the latest published article available at the time.

The results will show the covering costs of the benchmark circuits as created by the basic routine. Also the results of improvements like recognizing reconvergent fan-out structures (section 3.3), the use of a two input decompositions made for technology mapping purposes and extensive mappings — trying all input permutations — are given.

6.1 Basic results

The first results are generated using the selection algorithm for trees, so multiple fan-out signals will be implemented exactly once. During the matching to library cells only the number of inverters on the inputs to make the match is optimized. The first such match found is used. Every subcircuit of every signal is compared to every library cell. No clever hashing routine or otherwise is used. The computation times given for R_aCE include the time for reading the library and circuit and the computation of the signatures for the library cells. The results for *MARS* are published in [Kap95] and were done on a *SPARCStation 10*. The mapping with *SIS* uses the mapping options: '*-m 0 -f 2*', *SIS* therefore does recognize reconvergent fan-out structures. The timings given are in seconds.

The results of R_aCE compared to *MARS* show an improvement of about

	MARS		*SIS*		R_aCE	
Circuit	costs	time	costs	time	costs	time
C432	230144	20	228288	7	213440	2
C499	379552	17	487200	11	364704	2
C880	351248	22	379552	13	346144	2
C1355	609696	39	748896	21	485344	2
C1908	607376	47	638928	17	558656	4
C3540	1247696	89	1255120	114	1256050	14
C6288	2939904	29	3291616	305	2697700	11
Total	6365616		7029600	488	5922038	37

Table 6.1: Results of basic mapping routine compared to *MARS* and *SIS*

7 percent. Compared to *SIS* the improvement is even 16 percent. It must be noted that R_aCE only optimizes for area and does not check for load violations. This can however not account for a difference of 7 or 16 percent. Despite the exhaustive search R_aCE is a factor 13 faster than *SIS*. The timings of *MARS* can not be compared because it is run on another system.

6.2 Results for improved circuit selection routine

Now the improvement given in section 3.3 — the recognition of reconvergent multiple fan-out — is used. The results of the basic routine denoted by $R_aCE$are given as comparison. The results for the improved selection routine are given by R_aCE^*.

	R_aCE		R_aCE^*	
Circuit	costs	time	costs	time
C432	213440	2	213440	1
C499	364704	2	364704	3
C880	346144	2	346144	3
C1355	485344	2	485344	4
C1908	558656	4	557728	7
C3540	1256050	14	1252340	18
C6288	2697700	11	2697700	11
Total	5922038	37	5917400	47

Table 6.2: Results of mapping routine using reconvergence information

The use of this improved selection routine does not improve the results considerately. Most circuits have the same costs and some are a bit better. This was expected because the only improvement can be found for some special structures. The reconvergence must be found within the limit of the number of inputs that the searched subcircuit must have. It does increase the computation time however with 27 percent. Considering the speed at which the circuits are

covered this not a very important drawback. The next tests will therefore still use this improved routine. For larger circuits it must be taken into consideration to use the other selection routine when computation speed is more important.

6.3 Using all library cells

The previous mappings only used library cells up to five inputs. The *MCNC* library however also contains some six input functions. To see what the increase in time and the decrease in cover cost will be using these cells, a batch of tests is run using the same benchmark circuits. The results of mapping using library cell with up to five inputs are given by R_aCE^*, the results using also the six input cells are given by R_aCE^*-6.

	R_aCE^*		R_aCE^*-6	
Circuit	costs	time	costs	time
C432	213440	1	212976	2
C499	364704	3	364704	7
C880	346144	3	340112	4
C1355	485344	4	485344	8
C1908	557728	7	557728	15
C3540	1252340	18	1233310	29
C6288	2697700	11	2719970	13
Total	5917400	47	5914144	78

Table 6.3: Results of mapping routine also using 6 input cells

The costs have not decreased for all circuits. The time needed to cover the circuit is increased as expected, in some cases it even increases with a factor two. The improvement in the costs of the cover do not justify this increase in time. The decision to use only cells up to five inputs was correct. Only in cases where computation time is not a prime aspect and the costs of the cover are, it can be considered using all library cells.

6.4 Using specialized decompositions

Because the technology mapper does not make a new decomposition itself, the results will depend on the input circuit provided. To see what improvements can be reached using a special decomposition, the circuits are decomposed by an external routine. The decomposition is made with delay optimization during technology mapping in mind. The adapted circuits are then covered using library cells up to five inputs and reconvergence checking again. The results of mapping the standard circuit are given by R_aCE^*, the results using the adapted circuits are given by R_aCE^*-2.

On average the results have improved about 6 percent. Although the costs of one circuit have not improved. The computation time has also decreased.

Circuit	R_aCE^* costs	time	R_aCE^*-2 costs	time
C432	213440	1	190704	1
C499	364704	3	354496	3
C880	346144	3	323872	2
C1355	485344	4	362848	3
C1908	557728	7	390688	2
C3540	1252340	18	973936	13
C6288	2697700	11	2981660	7
Total	5917400	47	5578204	31

Table 6.4: Results of mapping routine using special decomposition

It is should therefore be taken into consideration that a special decomposition can improve the results significantly.

6.5 Using complete libraries

To compare the designed technology mapper using complete libraries against a specialized mapping routine, another set of tests is run. The library used contains all cells that can be implemented by a maximum of three transistors next to each other and three transistors in sequence. This library contains 87 cells. To have a correct comparison with the other technology mapper which uses the number of literals as cost function, the area cost for the library cells was set to the number of literals in the function.

Circuit	literals	R_aCE literals
C432	307	284
C499	725	736
C880	561	536
C1355	729	773
C1908	698	729
C2670	1029	1072
C3540	1858	1581
C5315	2322	2379
Total	8229	8090

Table 6.5: Results of mapping routine using 3x3 library, cost = literal count

Although R_aCE is not designed for complete libraries and therefore not uses the additional knowledge that this provides, the results can certainly be compared to the specialized mapper. The cover of some circuits is not better but overall the costs are 2 percent lower.

Chapter 7

Conclusions

Within the limited time space of a thesis project, a good method for technology mapping has been created. It consists of three modules: a subcircuit selection routine, a matching routine for boolean functions and a mapping and covering routine. The presented selection routine generates the subcircuits which can be mapped to a single library cell; it is also able to handle reconvergent fan-out. The matching routine is based on an adapted technique discussed in [Moh93]. The selection and mapping routines use an exhaustive approach in order to find a near-optimal cover. Despite the exhaustive nature of the designed technology mapper the computation time needed to find a cover is limited. The covering routine does not contain any clever heuristics but still the found covers give good results for area.

The routine is built using several separate modules. These modules have also been discussed separately. This modular structure provides the possibility for easy improvements in the future. One can adapt one module without changing other parts. This way also other cost functions can be used; it may be necessary to change the mapping or covering modules, but the selection and matching routines do not have to be modified. Using mapping sets, it is possible to implement multiple output library cells even though they can not be selected and matched using the existing routines. With routines dedicated to finding multiple output cells in the circuit, they can still be used.

Despite the rudimentary of the designed technology mapper, the results are already good. We have compared it to two other mappers for non-complete libraries and in all cases the covers created by our technology mapper have lower costs. It is also compared to a technology mapper for complete libraries and the overall costs of our method are lower. Thanks to the modular structure it can easily be adapted in the future and is a good basis for a technology mapper with a lot of possibilities.

Chapter 8

Recommendations for future work

As this technology mapper is the first start of a functional technology mapper for non-complete libraries, there is still some area uncovered. For example the use of multiple output cells should be supported. The subcircuit selection routine and the matching routine cannot be used efficiently to find multiple output cells. The mapping and covering routine however can use these cells because of the mapping sets that are used. Multiple output cells can improve the covering results and it is therefore important that they can be used. For future improvements a tool has to be designed that can find multiple output cells in the circuit efficiently. Also the mapping routine will have to be adapted a bit so that the costs of these cells is calculated correctly. Probably it is sufficient to order the index such that the output signals of a multiple output cell are adjacent in the index. One can expect a problem when several cells overlap. But these are premature conclusions because we have not looked at the implementation of these cells yet.

As mentioned it is also possible to do some preprocessing on the used library. Cells that are created by adapting library cells can be added to the library. Here it is necessary to have a routine that makes the derived cells by joining inputs possibly one inverted or putting a constant signal — zero or one — at one of the inputs. Using the matching routine one can check whether a cell like the one created, is already in the library. If there is none or the derived cell has some better characteristics, then the derived cell can be added to the library.

For practical use it is important that other cost functions are implemented. Probably it will be necessary to use load binning; this can easily be done using the already available mapping sets. For these other cost functions the selection and matching routines do not have to be changed. Also the mapping and covering routine do not need to be altered. Perhaps it is possible to devise a mapping and covering routine that is specialized in optimizing for delay. This is a case for future research.

The matching routine now only provides one match between a selected subcircuit and a library cell. For other cost functions or better global phase assignment decreasing the number of inverters used, it will be necessary to provide

several matches, all using an other input permutation. These other matches having another input permutation can be derived from the first match found. This opens the possibility to use a hashing routine. At one hash index all logically equivalent library cells using input permutation and negation can be put, together with all logically equivalent input permutations and negations of those library cells. For each hash entry with the same index, a conversion function of input permutations and negations to the first hash entry can be calculated. If a match can be found between a selected subcircuit and the first hash entry, then matches to all other entries can be calculated using the conversion functions. This way the number of times a match is searched is reduced. The time needed to find all matches between a selected subcircuit and all library cells can be decreased.

For the circuits that were used for testing and are presented as in chapter 6, some statistics were created for the matching routine. It was calculated that on average 14% of the run time was used by the matching routine. Only 1% of all selected subcircuits could not be matched. The remaining subcircuits generated on average 21 equal bitvectors. Therefore it could be a good suggestion to use a cache so that a known match does not have to be calculated again. This can significantly reduce the computation time used for matching.

Finally a subject not discussed in this report is incorporating design for testability during technology mapping. When producing ic's, some percentage of them holds some faults. To find the correct working ic's, they have to be tested. This job can be simplified if the ic was designed for testability. With decreasing ic-dimensions this becomes more and more important. As a starting point for testability design in technology mapping the article [Pom94] can be used.

References

[Bur92] Burch J.R. and D.E. Long
Efficient Boolean Function Matching.
In: 10th IEEE / ACM International Conference on Computer-Aided Design. Santa Clara, California, November 8–12, 1992. Ed. by: P. Storms.
Los Alamitos: IEEE Computer Society Press, 1992. P. 408–411.

[Che93] Cheng D.I. and M. Marek-Sadowska
Verifying Equivalence of Functions with Unknown Input Correspondence.
In: The European Conference on Design Automation with The European Event in ASIC Design. Proc. 4th Euro ASIC, Paris, France, February 22–25, 1993. Ed. by: P. Walker and E. Straub.
Los Alamitos: IEEE Computer Society Press, 1993. P. 81–85.

[Kap95] Kapoor B.
Improved Technology Mapping Using A New Approach to Boolean Matching.
In: The European Design and Test Conference ED & TC 1995. Paris, France, March 6–9 1995. Ed. by: B. Werner.
Los Alamitos: IEEE Computer Society Press, 1995. P. 86–90.

[Lai93] Lai Y-T. and M. Pedram, S.B.K. Vrudhula
BDD Based Decomposition of Logic Functions with Application to FPGA Synthesis.
In: 30th ACM / IEEE Design Automation Conference, Dallas, Texas, June 14–18, 1993.
Baltimore: IEEE, 1993. P. 642–647.

[Mai90] Mailhot F. and G. De Micheli
Technology Mapping Using Boolean Matching and Don't Care Sets.
In: 1st The European Design Automation Conference, Glasgow, Scotland, March 12–15, 1990.
Washington: IEEE Computer Society press, 1990. P. 212–216.

[Mai93] Mailhot F. and G. De Micheli
Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations.
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 12 (1993), no. 5, p. 599–620.

[Mic91] Micheli, G. De
Technology Mapping of Digital Circuits.
In: Advanced Computer Technology, Reliable Systems and Applications. Proc. 5th Annual European Computer Conference CompEuro '91, Bologna, Italy, May 13–16, 1991. Ed. by: V.A. Monaco and R. Negrini.
Los Alamitos: IEEE Computer Society Press, 1991. P. 580–586.

[Mic94] Micheli, G. De
Logic level synthesis and optimization.
New York: McGraw-Hill, 1994.

[Moh93] Mohnke J. and S. Malik
Permutation and Phase Independent Boolean Comparison.
In: The European Conference on Design Automation with The European Event in ASIC Design. Proc. 4th Euro ASIC, Paris, France, February 22–25, 1993. Ed. by: P. Walker and E. Straub.
Los Alamitos: IEEE Computer Society Press, 1993. P. 86–92.

[Pom94] Pomeranz I. and S.M. Reddy
Testability Considerations in Technology Mapping.
In: Proceedings of the Third Asian Test Symposium, Nara, Japan, November 15–17, 1994.
Los Alamitos: IEEE Computer Society Press, 1994. P. 151–156.

[Sch92] Schlichtmann U. and F. Brglez, M. Hermann
Characterization of Boolean Functions for Rapid Matching in EPGA Technology Mapping.
In: 29th ACM / IEEE Design Automation Conference. Anaheim, California, June 8–12, 1992. Ed. by: R. Werner.
Los Alamitos: IEEE Computer Society Press, 1992. P. 374–379.

[Sch93] Schlichtmann U. and F. Brglez
Efficient Boolean Matching in Technology Mapping with Very Large Cell Libraries.
In: IEEE 1993 Custom Integrated Circuits Conference. Proc. 15th annual Custom Integrated Circuits Conference, San Diego, California, May 9–12, 1993.
New York: IEEE, 1993. P. 3.6.1–3.6.6.

[Wan89] Wang A.
Algorithms for Multilevel Logic Optimization
PhD thesis, U.C. Berkeley, April 1989

[Wu94] Wu Q. and C.Y.R. Chen, J.M. Acken
Efficient Boolean Matching Algorithm for Cell libraries.
In: Proceedings IEEE International Conference on Computer Design, VLSI in Computers and Processors, Cambridge, Massachusetts, October 10–12, 1994.
Los Alamitos: IEEE Computer Society Press, 1994. P. 36–39.