MASTER

Integrated state encoding and logic optimization

Sleuters, H.J.H.

Award date:
1993

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
Integrated State Encoding
and
Logic Optimisation

H.J.H. Sleuters

Master Thesis

performed: September 1992 - July 1993
by order of: Prof. Dr. Ing. J.A.G. Jess
supervised by: Dr. Ir. J.F.M. Theeuwen

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

This report deals with a new kind of state encoding of Finite State Machines, namely Partial State Encoding (P.S.E.). In the old state encoding methods, the logic optimisation step was performed totally independent of the state assignment step. This means that there was a chance that the logic optimisation step after state encoding could not find the smart things prepared by the state encoder. In my work I tried to reduce this inconvenience by encoding not the whole state at once but one state-bit at a time. Logic optimisation will optimise the expressions that describe the F.S.M. and State encoding will code one bit of the states depending on the results of the optimisation. These two steps will be performed until all bits are encoded.

I implemented the idea in the logic optimisation program log_decom [1] and determined the results for four different methods and compared them to the results that I got from the (old) state encoding program chains [1].

The first conclusion to be drawn is that P.S.E. indeed will decrease the amount of transistors in the final design for F.S.M's. with lots of common subexpressions. A disadvantage of P.S.E. is that if the F.S.M. gets bigger it needs a lot more time to code the states one at a time than the old state encoding step.

I think that much more research on this subject can be done because there are so many ways in which you can try to reduce the amount of transistors in the design. This example of P.S.E. is implemented in the log_decom program but it could be implemented for any logic optimisation program.
# Contents

1 Introduction ................................................. 1

2 The Integration of State Encoding and Logic Optimisation ............. 3  
   2.1 Problem definition ...................................... 3  
   2.2 Basic Definitions ....................................... 4  
   2.3 The idea behind Partial State Encoding ...................... 5

3 The Logic Optimisation step .................................... 9  
   3.1 The Optimisation Program log_decom ........................ 9  
   3.2 Adapting log_decom for PSE ............................... 10  
      3.2.1 Criteria for 'good' kernels ......................... 11  
      3.2.2 List construction 1 ................................. 13  
      3.2.3 List construction 2 ................................ 13  
      3.2.4 List construction 3 ................................ 14  
      3.2.5 List construction 4 ................................ 14

4 The State Encoding Step ........................................ 17  
   4.1 The Encoding ............................................ 17  
   4.2 Searching for Kernels .................................... 18  
   4.3 New Data Structures in log_decom ......................... 20  
      4.3.1 Expr_set Data Structure ............................ 20  
      4.3.2 List_expr Data Structure ........................... 21  
   4.4 Assigning a legal code .................................. 22

5 Results ....................................................... 27  
   5.1 Converting .tab files to .log files ....................... 28  
   5.2 Results with List_construction 1 ........................ 29  
   5.3 Results with List_construction 2 ........................ 30  
   5.4 Results with List_construction 3 ........................ 31  
   5.5 Results with List_construction 4 ........................ 32

6 Conclusions and Recommendations ................................ 33

Bibliography .................................................... 35
Appendix A ................................................................. 37
At the Department of Electrical Engineering at the Eindhoven University of Technology, research is being done on the development of a silicon compiler. One of the steps in this project is the logic synthesis step. In this step four steps can be distinguished: state encoding, logic optimisation, technology mapping and gate sizing. Up to now the logic optimisation step was done independent of the state encoding step. As a result of this inconvenience the logic optimisation had a chance of not finding the smart things prepared by the state encoding step.

This report describes a new way of state encoding that makes use of the results from logic optimisation and encodes a state not at once but one bit at a time.
Integrated State Encoding and Logic Optimisation
Chapter 2

The Integration of State Encoding and Logic Optimisation

2.1 Problem definition

Given a Finite State Machine (FSM) with a State Transition Table (STT), the main goal of state encoding is to code each symbolic state of the machine with a unique binary number in such a way that the expressions, derived for the external outputs and the inputs to the memory elements (Flip-flops), can be simplified by logic optimisation. In 'normal' state encoding, the symbolic states are given a unique binary number at once. The encoding of the states is usually made by the use of some kind of heuristic that looks at the STT and decides what the 'best' encoding is for the states. After this step the above mentioned expressions are derived and given to a logic optimisation program. This optimisation step is totally independent of the state encoding, so there's a chance that the optimisation step may not at all find the good things the state encoding step has prepared! To overcome this problem a new way of state encoding is developed namely Partial State Encoding (PSE).
2.2 Basic Definitions

In this paragraph some basic definitions will be given that will be used in the rest of this report.

- A variable is a symbol representing a single coordinate of the Boolean space (e.g., \(a\)).
- A literal is a variable of its negation (e.g., \(a\) or \(a'\)).
- A cube is a set \(C\) of literals such that \(x \in C\) implies \(x' \notin C\) (e.g., \([a, b, c]\) is a cube and \([a, a']\) is not a cube). A cube represents the conjunction of its literals. The trivial cubes, written 0 and 1, represent the Boolean functions 0 and 1, respectively.
- An expression can be represented by a sum of cubes.
- if \((fg)g = f\) holds, \(g\) divides \(f\) evenly (e.g., \(a\) divides \(a.b + a.c\) evenly).
- An expression \(f\) is cubefree if and only if 1 divides \(f\) evenly (e.g., \(a.b + c.d\) is cubefree but \(a.b + a.c\) is not cubefree).
- A primary divisor of a boolean expression \(f\) is defined as:
  \[\{f/c \mid c = \text{cube}\}\].
- A kernel is a cubefree primary divisor.
- A kernel of level 0 is a kernel which contains no other kernel.
- An ns_expression is an expression for the symbolic next-state-bits that still have to be encoded.
- An nc_expression is an expression for an encoded bit of the next-states.

An example might explain the last two definitions:

**Example:**

for the right S.T.T. of figure 2 of paragraph 2.3 the ns_expressions are:

\[
\begin{align*}
\text{ns_st0: } & i'_1.i'_0.oc_1'.st0 + i_1.i_0.oc_1'.st0 + i_1.i_0.oc_1'.st1 \quad \text{(row 1 + row 2 + row 5)} \\
\text{ns_st1: } & i'_1.i_0.oc_1'.st0 + i_1.i_0.oc_1'.st1 + i_1.i_0.oc_1'.st2 \quad \text{(row 3 + row 4 + row 8)}
\end{align*}
\]
The Integration of State Encoding and Logic Optimisation

\[ \text{ns}\_\text{st2}: i_1, i_0, \text{oc}_1, \text{st}1 + i_1, i_0, \text{oc}_1, \text{st}2 + i_1, i_0, \text{oc}_1, \text{st}3 \quad \text{(row 6 + row 7 + row 11)} \]
\[ \text{ns}\_\text{st3}: i_1, i_0, \text{oc}_1, \text{st}3 + i_1, i_0, \text{oc}_1, \text{st}3 \quad \text{(row 9 + row 10)} \]

the \text{nc}\_\text{expression} is:

\[ \text{nc1: } i_1, i_0, \text{oc}_1, \text{st}0 + i_1, i_0, \text{oc}_1, \text{st}1 + i_1, i_0, \text{oc}_1, \text{st}2 + (\text{ns}\_\text{st1} + \text{ns}\_\text{st3}) \]
\[ i_1, i_0, \text{oc}_1, \text{st}3 + i_1, i_0, \text{oc}_1, \text{st}3 \]

you can see that the \text{nc1} expression is a combination of the \text{ns}\_\text{expressions} corresponding to the '1'-assigned states namely st1 and st3.

the external output expression is:

\[ u0: i_1, i_0, \text{oc}_1, \text{st}1 + i_1, i_0, \text{oc}_1, \text{st}1 + i_1, i_0, \text{oc}_1, \text{st}2 + i_1, i_0, \text{oc}_1, \text{st}2 + i_1, i_0, \text{oc}_1, \text{st}3 + i_1, i_0, \text{oc}_1, \text{st}3 + i_1, i_0, \text{oc}_1, \text{st}3 \quad \text{(row 4 + row 6 until row 11)} \]

\[ \textbf{2.3 The idea behind Partial State Encoding} \]

In partial state encoding the symbolic states are not encoded at once but bit after bit. How does this work. First take a look at the FSM in figure 1. From this state diagram a State Transition Table can be constructed. To implement

\[ \begin{array}{c|c|c|c|c}
\text{Inputs} & \text{Old State} & \text{New State} & \text{Outputs} \\
\hline
i_1, i_0 & s0 & s0 & 0 \\
11 & s0 & s0 & 0 \\
01 & s0 & s1 & 0 \\
00 & s1 & s1 & 1 \\
11 & s1 & s0 & 0 \\
10 & s1 & s2 & 1 \\
10 & s2 & s2 & 1 \\
00 & s2 & s1 & 1 \\
01 & s2 & s3 & 1 \\
00 & s3 & s3 & 1 \\
11 & s3 & s2 & 1 \\
\end{array} \]

\[ \text{figure 1} \]

A State diagram and State Transition Table of an FSM
Integrated State Encoding and Logic Optimisation

this FSM in multi-level logic, expressions for external outputs and new-states must be derived. In the case of 'normal' state encoding the symbolic states are substituted by binary numbers before the derivation. In the case of Partial State Encoding on the other hand, the symbolic states are not substituted totally, so the derived output expressions contain symbolic states.

After the expressions are derived the logic optimisation program can start working on them. The logic optimisation program will create a list of common kernels. Common kernels are smaller expressions which appear more than once in different optimised expressions. So, if these kernels are replaced, by a new internal variables (which represents the kernel), in every expression in which it occurs, the total expression becomes smaller. The kernel itself becomes a new expression with the name of the internal variable. The goal for state encoding is now:

- find an encoding such that one or more of these common kernels appear in the new nc_expression

After Partial State Encoding one bit of every symbolic state has been encoded. In the example, st3 is given a 1-codebit. This means that the symbolic state can

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Old State</th>
<th>New State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 st0</td>
<td>0 st0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0 st0</td>
<td>0 st0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0 st0</td>
<td>1 st1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1 st1</td>
<td>1 st1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1 st1</td>
<td>0 st0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1 st1</td>
<td>0 st2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0 st2</td>
<td>0 st2</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0 st2</td>
<td>1 st1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0 st2</td>
<td>1 st3</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1 st3</td>
<td>1 st3</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1 st3</td>
<td>0 st2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Old State</th>
<th>New State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>st0</td>
<td>st0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>st0</td>
<td>st0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>st0</td>
<td>st1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>st1</td>
<td>st1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>st1</td>
<td>st0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>st1</td>
<td>st2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>st2</td>
<td>st2</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>st2</td>
<td>st1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>st2</td>
<td>st3</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>st3</td>
<td>st3</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>st3</td>
<td>st2</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2
The STT after one bit has been encoded
The Integration of State Encoding and Logic Optimisation

be written as: 1.st3 (where st3 stands for the remaining, still to be coded, bits). These new symbolic states can be substituted in the STT (figure 2). You can see that the encoded bit in the old-state column can be seen as a new input, because the value of this bit will not change anymore in this encoding. The same thing holds for the encoded bit in the next-state column which can be seen as a new output. From this new STT, the output expressions can be derived again! These can be optimised and again one bit can be coded. This process goes on until all bits of the symbolic states are encoded.

The advantage of this way of encoding is that you look at the results of logic optimisation before encoding a state. So you have an indication of what sub-expressions will be substituted and state encoding can look if the new output expression can contain one or more of these sub-expressions.

I've been assigned to find out if this method of State Encoding leads to better results in terms of number of transistors in the final design. The method must make use of the program log_decom decom for the logic optimisation step.
Integrated State Encoding and Logic Optimisation
Chapter 3

The Logic Optimisation step

3.1 The Optimisation Program log_decom

Log_decom is a program with which a set of boolean expressions can be optimised. The optimisation is targeted towards a multi-level implementation of the specified combinational logic. The basic operation performed by log_decom is searching for common kernels in the set of expressions to be optimised. Log_decom operates on the following simplified idea: if a certain kernel exists several times, it might be cheaper to realise this kernel only once. The result of this expression can then be substituted in all the expressions in which it occurs. This action has the following consequences:

- An extra logic level is introduced.
- The number of transistors in the overall circuit becomes smaller.
- The expression become less complex.

The program log_decom can perform four basic operations:

- **simplification**: This is the first step in the optimisation process and must be performed before any other operation is performed. Simplification writes each expression as a minimal sum of product terms. In general the number of literals in an expression...
Integrated State Encoding and Logic Optimisation

will decrease by this operation.

- **distillation:** In this step the program will search for common kernels in the set of expressions. This step consists of three sub-steps:
  1. Determine for all expressions the kernels of level 0.
  2. Compare all kernels with each other and build a list of common kernels.
  3. From this list determine the most suitable kernel and realise this kernel as a new expression.

The user can repeat these steps until there are no common kernels in the list.

- **condensation:** This step is analogous to the distillation step but now the program will search for common cubes in the set of expressions. The three sub-steps from the distillation step also hold for the condensation step.

- **collapsing:** There is a possibility that substitutions made by earlier condensation or distillation steps are no longer significant. The sub-expressions can then be substituted back into the other expressions in which it occurs.

### 3.2 Adapting log_decom for PSE

The goal for PSE is to make a state assignment that will minimize the number of transistors in the resulting design. Because log_decom works with common kernels which it can substitute several times, I used the common kernel idea for my implementation of P.S.E.

P.S.E. looks for 'good' kernels and constructs new nc_expressions in which these kernels appear. So a list of 'good' kernels has to be constructed somehow. Log_decom contains a function that can determine all the kernels in an expression. Furthermore it constructs a list of common kernels of level 0, com_kernel_list, in it's distillation process which will be copied in a global list com_kernel_list.
From this set of kernels, PSE will create a list, the gen_kernel_list, in which the kernels are ordered in a certain order of importance. The 'best' kernels will be placed at the front of the list.

In my research I examined four ways of constructing this list. The differences between the construction of the lists are based on the question which kernels will be used to determine a state assignment:

- Use all common kernels in the list. This construction method is described in paragraph 3.2.2.
- Use only kernels that appear in external output expressions or nc_expressions (not necessarily common kernels). This construction method is described in paragraph 3.2.3.
- Use only those common kernels that appear in output expressions and that are substituted by the distillation process. This construction method is described in paragraph 3.2.4.
- Use a list of three classes of kernels. This construction method is described in paragraph 3.2.5.

3.2.1 Criteria for 'good' kernels

The order in which the kernels are stored in the gen_kernel_list is important for the state assignment. Several criteria can be used to determine which kernels are good and which not.

- If Big kernels are substituted the gain will be higher. So kernels with a great number of cubes are 'good'. (Kernel -> amountofcubes must be big!)
- If small kernels are substituted the gain might be lower but the chance that this kernel can be found in a single or a combination of ns_expressions is higher. So kernels with a small number of cubes are 'good'. (Kernel -> amountofcubes must be small).
- Kernels that appear in several different expressions are more likely to be substituted by log_decom than kernels that appear only in one expression.
So the number of expressions that contain the kernel can be a criterium.

- Kernels with only input variables and or one state variable are 'good' because they will always keep their amount of cubes independent of a state assignment. Kernels with two or more state variables have a chance of being simplified by state encoding, so the original 'big' kernel can be reduced to a small kernel.

Example:

Kernel with 1 state: \(a + b + st1\) contains three cubes. After state assignment \(st1 = e.f.g\) the kernel looks like: \(a + b + e.f.g\) and still contains three cubes.

Kernel with 2 states: \(a + b + st1 + st2\) contains four cubes. After state assignment state \(st1\) and \(st2\) will have a hamming distance of at least one because every state has a unique code: \(st1 = e.f.g\), \(st2 = e.f.g\)

Kernel: \(a + b + e.f.g + e.f.g = a + b + e.f = \) kernel with 3 cubes \(-\) 1 cube is gone!

Kernel with 3 states: \(a + b + st1 + st2 + st3\) contains five cubes.
\(st1 = e.f.g\), \(st2 = e.f.g\), \(st3 = e.f.g\):

Kernel: \(a + b + e.f.g + e.f.g + e.f.g = a + b + e.f + e.f.g = a + b + e(f + g)\). This kernel contains 4 cubes and it is not anymore a kernel of level 0 so \(log_{decom}\) will not recognize this kernel totally anymore if it searches for kernels of level 0 which it does in my implementation. It will find the kernel \(f + g\) which contains only two cubes!

For kernels with more state-variables something similar can be derived. If more state-variables are contained in the kernel then this kernel can be reduced in size more easily.

- The number of different literals in a kernel can be a criterium. If a kernel consists of many literals then the the gain of substituting this kernel will be bigger. \(setcount(kernel \rightarrow union_r)\) must be big.

Of course it is also possible to combine the above criteria in some sort of weight function. Function (1) for example can be used as a weight function in which the amount of cubes in the kernel and the number of literals are modelled. The
constants A and B can be chosen by the user and they reflect the importance of both criteria.

\[ W_{\text{kernel}} = A \cdot \text{Amountofcubes} + B \cdot \text{numberliterals} \]

The list will then be ordered on this weightfunction.

**3.2.2 List construction 1**

List construction 1 will construct the `gen_kernel_list` exactly like the `comkern_list`. This is the most simple way of construction and therefore the construction takes the least amount of time: It's a copy of `comkern_list` but sorted on a criterium mentioned in paragraph 3.2.1.

**3.2.3 List construction 2**

List construction 2 will construct the `gen_kernel_list` with kernels that only appear in external output expressions or `nc_expressions` (not necessarily common kernels!).

The idea behind this `gen_kernel_list` lies in the fact that output and `nc_expressions` will always stay in the set of expressions to be optimised in contrary to the `ns_expressions` which after the last pass will disappear for certain (the encoding is complete and there are no more symbolic state-bits to be encoded) and in all the other passes a certain number of `ns_expressions` will or cannot be used to form an `nc_expression` because the encoding would not be unique. The kernels in output and `nc_expressions` will exist until the end so kernels of these expressions are more important than kernels of the `ns_expressions`!

The idea of using all kernels in output or `nc_expressions` rather than common kernels lies in the fact that a kernel which is not common at the beginning of
PSE has a big chance of being that after PSE, because PSE will only look for those kernels to form new nc_expressions.

### 3.2.4 List construction 3

For list construction 3 log_decom performs a complete distillation. This means that it will continue the distillation process until there are no more kernels that can be substituted. At the end of this process some kernels may be substituted and the corresponding internal expressions will be added to List_of_expr. The construction of gen_kernel_list now consists of adding the substituted kernels to it.

The idea behind using substituted kernels lies in the simple fact that after distillation these kernels will always be found regardless of the new nc_expression. In the construction of the new nc_expression, P.S.E. will try to construct it in such a way that the substituted kernels also appear in this expression. So the kernel will also be substituted in the new nc_expression.

### 3.2.5 List construction 4

After one distillation step the comkern_list is constructed. The common kernels of this list can be divided in three classes of kernels:

- **Class 1**: Common kernels that appear only in ns_expressions.
- **Class 2**: Common kernels that appear only in one external output or a newly created expression (nc_expression), so it is not a common kernel in the output expressions, and in ns_expressions.
- **Class 3**: Common kernels that appear in external output expressions (And maybe also in ns_expressions).

*Example expressions:*
**The Logic Optimisation Step**

ns_st0: \((a + b).c\)  
ns_st1: \((a + b).d\)  
ns_st2: \((e + f).g + a + d\)  
ns_st3: \(a + b\)

\[
\begin{align*}
\text{u0:} & \quad (e + f).c + (a + d).g \\
\text{u1:} & \quad (a + d).e
\end{align*}
\]

\(a + b\) is a common kernel of class 1  
\(e + f\) is a common kernel of class 2  
\(a + d\) is a common kernel of class 3

**Kernels of class 1:**

If PSE constructs a new output expression which contains this class of kernels, the kernels will not be common in the output expressions, which means that in the last pass of PSE an expression is made in which no common kernel is found. If such new expression is made earlier, there is chance that this kernel becomes a common output kernel if other nc_expressions contain this kernel too.

**Kernels of class 2:**

If PSE constructs a new output expression which contains this class of kernels a new common kernel is created.

**Kernels of class 3:**

Kernels of this class are kernels that are common in output expressions so they will always exist because the output expressions will not change anymore (see list_construction_2). A new output expression which contains this class of kernel will increase the number of times that the kernel exists.

So, the common kernels in order of importance: common kernels of class 3, class 2, class 1. So for PSE log_decom has to construct a list of common kernels sorted in descending order of importance. For the Example: \((a + d), (e + f), (a + b)\)
To construct the `gen_kernel_list`, three separate singly-linked lists (for each class of kernels one list), sorted on a criterium of paragraph 3.2.1, will be created and then merged into one. To construct the three lists, first the type of every common kernel in `comkern_list` must be determined. A simple observation learns that to determine the type of a kernel, the number of output expressions that contain this kernel has to be counted: If a kernel is contained in more than one output or `nc_expression` then it is a kernel of class 3. If a kernel is contained in only one output or `nc_expression` then it is a kernel of class 2. If a kernel is not contained in any output or `nc_expression` then it is a kernel of class 1.
In chapter 3 four different methods of creating a 'good' gen_kernel_list were explained. Now this kernel list will be used in the encoding of the states of the F.S.M. In chapter 2 the encoding process was described briefly. In this chapter a more detailed description will be given.

4.1 The Encoding

In chapter two the encoding of one bit of the states was described. This one bit encoding introduced a new output expression nc₁ and a new input variable oc₁. Observation of the STT learns that nc₁ is nothing more then the summation of the ns_expressions corresponding to the '1'-encoded states after substitution of the new variable oc₁. So the new expression that will be created after one state assignment step is the sum of the '1'-encoded ns_expressions. This means that you can 'choose' the nc_expression by assigning the right ns_expressions a '1'-encoding.
Example:

For the STT from figure 2 of chapter 2: nc; ns_st1 + ns_st3

(*: ns_expressions after substitution of the new input variable)

In PSE the state encoding is based on creating common kernels in the new output expressions. So if one or more of the kernels of the gen_kernel_list can be found in ns_expressions, assigning these ns_expression with '1' will result in a new nc_expression that contains these kernels too! The next step for PSE is obviously the search for kernels of the gen_kernel_list in the ns_expressions.

4.2 Searching for Kernels

The search for kernels will result in a new linked list called the Expr_list. The elements of this list are of the type expr_list which will be described in paragraph 4.3.

There are two possibilities for finding kernels in ns_expressions:

- a kernel can be found in one single ns_expression
- a kernel can be constructed by combining several ns_expressions.

The first possibility is the most obvious one. For all ns_expressions in List_of_expr (this is the list that contains all expressions which describe the F.S.M. in log_decom) compare this searchkernel with each of the kernels in that expression. If one of the kernels is equal to the searchkernel then add this expression to Expr_list.

To find a kernel in a combination of ns_expressions is more complex. The following algorithm can be used to solve this problem:
• Take a cube from the searchkernel (searchcube).
• Find a cube in an ns_expression in which the searchcube is contained.
• Divide this cube by the searchcube and call the result divisor.
• Add the ns_expression to $Expr_set$ (the set in which the combination of ns_expressions is stored).
• For all other searchcubes in searchkernel, search a cube in which the corresponding searchcube is contained and divide it by the divisor. If the result equals the searchcube then this expression can be added to $Expr_set$.
• If this search can be completed for all cubes in searchkernel - than the kernel can be found in the summation of ns_expressions in $Expr_set$.

An example might explain this algorithm:

---

**Example:**

$Expr_set = \emptyset$.

Searchkernel: $a + b + c$

ns_1: \( f(a + b) + g.c \)

ns_2: \( d.e + f.c \)

Searchcube: $a$

Cube $fa$ of ns_1 contains the searchcube $a$.

$f.a/f = a = \text{divisor!}$

$Expr_set = Expr_set \cup \{ns_1\} = \{ns_1\}$

Searchcube: $b$

Cube $fb$ of ns_1 contains the searchcube $b$.

$f.b/f = b = \text{Searchcube!}$

$Expr_set = Expr_set \cup \{ns_1\} = \{ns_1\}$

---
Searchcube: c

Cube \( g.c \) of ns_1 contains the searchcube c.
\[ g.c = \emptyset \neq \text{Searchcube}. \]
Cube \( f.c \) of ns_2 contains the searchcube c.
\[ f.c = c = \text{Searchcube!} \]

\[ \text{Expr}_\text{set} = \text{Expr}_\text{set} \cup \{ \text{ns}_2 \} = \{ \text{ns}_1, \text{ns}_2 \} \]

So kernel \( a + b + c \) can be found in a combination of ns_1 and ns_2:
Combination: \( f.(a + b) + g.c + d.e + f.c = f.(a + b + c) + g.c + d.e \)
The combination indeed contains the searchkernel!

An observation of the algorithm learns that kernels that appear in one single ns_expressions can be found too by this method. So to construct Expr_list only this algorithm has to be used!

### 4.3 New Data Structures in log_decom

As mentioned in the above paragraph some new data structures are added to log_decom to store the Expr_list.

#### 4.3.1 Expr_set Data Structure

The expr_set data structure is added to log_decom to store sets of expressions. So it can be used to store the combination of ns_expressions which - if combined - contain a kernel from the gen_kernel_list. Also this structure is used to store the set of states. The structure is described below:

```c
struct _expr_set *expr_set;
struct expr_set {
    numofexpr exprnum
};
```
int torcount;
int legalcode;
char used;
identstryp name;
char code[20];
ext_set next;

The field exprnumber identifies an (ns_)expression. The field torcount contains the number of transistors in the combined expression. The fields legalcode, used, name and code are used in the actual encoding of the states and will be explained in paragraph ?.

4.3.2 List_expr Data Structure

The list_expr data structure is used to store the total result of the search for kernels of the gen_kernel_list. The structure is described below:

```c
struct _expr_list *expr_list;
struct expr_list {
    int torcount;
    expr_set set;
    int num;
    kernel_ptr kernel;
    expr_list next;
}
```

The field torcount contains the number of transistors in the combined expression of all expressions in set. set contains the set of expressions that - if combined - contain kernel kernel.

The order in which the elements are stored in Expr_list depends on the way the elements of gen_kernel_list are stored. If a kernel A is positioned in front of kernel B in the gen_kernel_list then all the combinations of ns_expressions that
form kernel A will be placed in front of all the combinations of ns_expressions that form kernel B.

Because there can be more than one combination of ns_expressions which form the same kernel, it is also possible to sort these combinations in some order of importance. A few possibilities for the sort order are:

- Sort on the number of ns_expressions that form the kernel. If many ns_expressions form a kernel then it will not be necessary to add more ns_expressions to form a new nc_expression. (A certain number of ns_expressions have to be coded '1' to get a legal encoding). So the expression stays smaller.
- Sort on the number of transistors in the combined expression.
- Of course it is possible to combine several different weights in a cost function.

4.4 Assigning a legal code

Now that the Expr_list is constructed, the actual encoding of the states can start. First the number of state bits to be encoded, \( N_b \), has to be calculated. To encode \( n \) states in such a way that all the states are given a unique code formula (2) can be used.

\[
N_b = \lceil \log_2(n) \rceil \tag{2}
\]

This number also represents the number of times P.S.E. will perform the optimisation and state assignment step.

In paragraph 4.1 was stated that you could choose the ns_expressions to form an nc_expression. This is not really true. Because the encoding of all the states has to be unique a couple of states have to be assigned '1' and another couple of states have to be assigned '0'. To make the encoding problem clear look at an
encoding of an FSM with 16 states:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Step 4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

In all the steps 8 states have to be assigned '1' and eight states have to be assigned '0'. In the first step eight states may be chosen anyway you like it. In the second step there is a restriction to the assignments: four of the states that have the same encoding up to now may be assigned '1'. If you assign five states that '1' then the encoding could never be unique because you will have only two encoding steps (bits) to encode five states uniquely. This cannot be done. In the third step only two of four states that have been assigned the same code up to now may be assigned '1' and in the last step only one of the two states that have the same code up to now may be assigned '1'.

From the example of encoding 16 states uniquely the general encoding scheme can be given as follows:

Given an F.S.M. with \( n \) states calculate \( N_k \). Use the variable \( k \) to keep track of the encoding step your in (for the first step \( k = 1 \), for the second step \( k = 2 \), etc.). In step \( k \) precisely \( 2^{N_k - k} \) states that have the same encoding up to step \( k - 1 \) can be encoded as '1'.

It is obviously very important to determine the states that have the same code up to step \( k - 1 \) for each state. For this purpose the field `legalcode` in de structure `expr_set` is introduced.

At the start of the encoding the `legalcode` for each state has been assigned the value 0. Then for a state that has been assigned '1' add \( 2^{2k - 1} + 1 \) to `legalcode`
and for a state that has been assigned '0' add $2^{(k-1)}$ to legalcode. Then after $k$ steps, the states that have the same encoding up to now have the same legalcode!

**Example:** number of states = 8, $N_s = 3$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>legal code</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Step 2</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>legal code</td>
<td>10</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td><strong>Step 3</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>legal code</td>
<td>42</td>
<td>26</td>
<td>38</td>
<td>22</td>
<td>41</td>
<td>25</td>
<td>37</td>
<td>21</td>
</tr>
</tbody>
</table>

Now that the number of '1' assignments can be determined, *Expr_list* will be used to determine which states will be assigned as '1':

1. Take the front element of *Expr_list* and determine if all the states in the set (of ns_expressions) can be assigned as '1'.
2. If this is not possible dump the element and goto step 1.
3. If the assignment is possible then assign these states '1' and update the legalcode and the used field of these states.
4. Check for all the states in set, if by assigning this state '1', the maximum number of '1' assignments, for states with the same legalcode in step $k - 1$, is reached. If so, encode all the other states with the same legalcode in step $k - 1$ as '0' and update the legalcode and used field of these states.
5. Dump the element.
6. Goto 1 until *Expr_list* is empty or until all states have been encoded.

Now two situations can occur:
The State Encoding Step

1. All states have been encoded.
2. $Expr\_list$ is empty but the assignment is not complete yet.

In the case of situation 1 nothing else has to be done to get a legal encoding for this step. Only step 1 of the following steps has to be performed. In the case of situation 2 there are still some states to be encoded. For this situation steps 1 to 4 of the following steps have to be performed:

1. Combine the ns_expressions which correspond to the \textquote{1} assigned states into one expression and call this expression $comb\_expr$.
2. Combine this $comb\_expr$ with each of the not yet encoded ns_expressions and record the number of transistors in each of the combinations.
3. Add the ns_expression which in combination with $comb\_expr$ results in the smallest amount of transistors to $comb\_expr$.
4. Goto step 1 until all states have been encoded.

If all the state bits have been encoded for this step, the expressions in $List\_of\_expr$ will be updated. A state $st$ which has been assigned \textquote{1} will be replaced by $st.\text{oc}\_N_b\_k$ and a state $st$ which has been assigned \textquote{0} will be replaced by $st.\text{o'c}\_N_b\_k$, where $\text{oc}\_N_b\_k$ is the new input (in the last step $k = N_b$ only the new inputvariable will be inserted for $st$ because all statebits have been encoded). Also a new outputexpression $nc\_N_b\_k \ (= \ comb\_expr)$ will be added to $List\_of\_expr$.

Now the encoding for step $k$ is complete.
Chapter 5

Results

In the following results the number of transistors, according to \textit{log\_decom}, after encoding and optimisation are given for the four different list constructions. In the first column the results for the encoding of the benchmark circuits by the 'old' state encoding program \textit{chains}, after simplification, distillation and condensation by \textit{log\_decom} are displayed. In column A, B and C the results of an encoding for the benchmark circuits by P.S.E. are displayed. The different columns show the results for different sort orders of \textit{gen\_kernel\_list} and \textit{Expr\_list}:

- Column A: \textit{gen\_kernel\_list} is sorted on a descending amount of literals in the kernel and \textit{Expr\_list} is sorted on a descending amount of literals in the kernel.
- Column B: \textit{gen\_kernel\_list} is sorted on a descending amount of cubes in the kernel and \textit{Expr\_list} is sorted on a descending amount of literals in the kernel.
- Column C: \textit{gen\_kernel\_list} is sorted on a descending amount of cubes in the kernel and \textit{Expr\_list} is sorted on a descending weight of the function: \( W = \) amount of cubes in the kernel - amount of transistors
in the combined expression of ns_expressions which form the kernel.

Furthermore the CPU time required to run the testfiles is displayed. The test runs were performed on a HP 9000/S735 machine. The CPU time is determined for the encoding of column C. This time is also representative for the encoding of columns A and B, because only the sort equations in the sort algorithm change and not the coding algorithm.

5.1 Converting .tab files to .log files

P.S.E. is tested on some benchmark circuits. These circuits were stored in a .tab file (State Transition Table like). To use them with log_decom they first had to be converted to the .log log_decom format. For this purpose I wrote the tab2log program. The tab2log program codes all the don't cares in a .tab file to '0'.

28
5.2 Results with List_construction_1

Table 3: The results for encodings with Chains and the three different methods of P.S.E. with list_construction_1.

<table>
<thead>
<tr>
<th>Method: Circuit:</th>
<th>Chains</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>CPU (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbara</td>
<td>104</td>
<td>88</td>
<td>78</td>
<td>78</td>
<td>0:03</td>
</tr>
<tr>
<td>bbsse</td>
<td>166</td>
<td>156</td>
<td>165</td>
<td>165</td>
<td>0:19</td>
</tr>
<tr>
<td>bbtas</td>
<td>29</td>
<td>36</td>
<td>37</td>
<td>27</td>
<td>0:00</td>
</tr>
<tr>
<td>cse</td>
<td>278</td>
<td>284</td>
<td>252</td>
<td>252</td>
<td>0:58</td>
</tr>
<tr>
<td>dk14</td>
<td>139</td>
<td>140</td>
<td>135</td>
<td>135</td>
<td>0:05</td>
</tr>
<tr>
<td>dk15</td>
<td>93</td>
<td>73</td>
<td>82</td>
<td>82</td>
<td>0:01</td>
</tr>
<tr>
<td>dk16</td>
<td>350</td>
<td>323</td>
<td>305</td>
<td>306</td>
<td>1:30</td>
</tr>
<tr>
<td>lion</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>0:00</td>
</tr>
<tr>
<td>lion9</td>
<td>73</td>
<td>84</td>
<td>64</td>
<td>64</td>
<td>0:03</td>
</tr>
<tr>
<td>mc</td>
<td>24</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>0:00</td>
</tr>
<tr>
<td>modulo12</td>
<td>40</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>0:00</td>
</tr>
<tr>
<td>s1</td>
<td>462</td>
<td>485</td>
<td>409</td>
<td>330</td>
<td>1:49</td>
</tr>
<tr>
<td>s8</td>
<td>66</td>
<td>73</td>
<td>73</td>
<td>73</td>
<td>0:01</td>
</tr>
<tr>
<td>train11</td>
<td>89</td>
<td>94</td>
<td>94</td>
<td>94</td>
<td>0:02</td>
</tr>
</tbody>
</table>

The sort order used in column C gives the best encoding for all three P.S.E. encodings for list_construction_1. Furthermore it can be seen that for bigger the CPU time increases very fast. For small benchmark circuits the encoding of chains results most of the time in a smaller amount of transistors than the P.S.E. encodings using list_construction_1.
Integrated State Encoding and Logic Optimisation

5.3 Results with List_construction 2

*Table 4:* The results for encodings with Chains and the three different methods of P.S.E. with list_construction_2.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Chains</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>CPU (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbara</td>
<td>104</td>
<td>105</td>
<td>96</td>
<td>84</td>
<td>0:02</td>
</tr>
<tr>
<td>bbsse</td>
<td>166</td>
<td>164</td>
<td>164</td>
<td>164</td>
<td>0:12</td>
</tr>
<tr>
<td>bbtas</td>
<td>29</td>
<td>27</td>
<td>27</td>
<td>34</td>
<td>0:00</td>
</tr>
<tr>
<td>cse</td>
<td>278</td>
<td>248</td>
<td>255</td>
<td>255</td>
<td>1:08</td>
</tr>
<tr>
<td>dk14</td>
<td>139</td>
<td>147</td>
<td>120</td>
<td>133</td>
<td>0:03</td>
</tr>
<tr>
<td>dk15</td>
<td>93</td>
<td>73</td>
<td>82</td>
<td>82</td>
<td>0:00</td>
</tr>
<tr>
<td>dk16</td>
<td>350</td>
<td>321</td>
<td>327</td>
<td>347</td>
<td>0:36</td>
</tr>
<tr>
<td>lion</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>0:00</td>
</tr>
<tr>
<td>lion9</td>
<td>73</td>
<td>87</td>
<td>74</td>
<td>76</td>
<td>0:01</td>
</tr>
<tr>
<td>mc</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>0:00</td>
</tr>
<tr>
<td>modulo12</td>
<td>40</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>0:00</td>
</tr>
<tr>
<td>s1</td>
<td>462</td>
<td>443</td>
<td>457</td>
<td>457</td>
<td>0:46</td>
</tr>
<tr>
<td>s8</td>
<td>66</td>
<td>73</td>
<td>86</td>
<td>86</td>
<td>0:03</td>
</tr>
<tr>
<td>train11</td>
<td>89</td>
<td>100</td>
<td>94</td>
<td>94</td>
<td>0:02</td>
</tr>
</tbody>
</table>

For list_construction_2 the sort order used in column B gives the best results for P.S.E. For small benchmark circuits like lion, bbtas and mc this list_construction gives better results than list_construction_1. Observation learns that this construction takes less CPU time for bigger files than list_construction_2 but the encoding of these files is better in list_construction_1.
5.4 Results with List_construction 3

Table 5: The results for encodings with Chains and the three different methods of P.S.E. with list_construction_3.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Chains</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>CPU (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbara</td>
<td>104</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>0:02</td>
</tr>
<tr>
<td>bbsse</td>
<td>166</td>
<td>171</td>
<td>171</td>
<td>171</td>
<td>0:06</td>
</tr>
<tr>
<td>bbtas</td>
<td>29</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>0:00</td>
</tr>
<tr>
<td>cse</td>
<td>278</td>
<td>248</td>
<td>248</td>
<td>248</td>
<td>0:55</td>
</tr>
<tr>
<td>dk14</td>
<td>139</td>
<td>122</td>
<td>122</td>
<td>122</td>
<td>0:02</td>
</tr>
<tr>
<td>dk15</td>
<td>93</td>
<td>87</td>
<td>87</td>
<td>87</td>
<td>0:01</td>
</tr>
<tr>
<td>dk16</td>
<td>350</td>
<td>295</td>
<td>295</td>
<td>332</td>
<td>0:45</td>
</tr>
<tr>
<td>lion</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>0:00</td>
</tr>
<tr>
<td>lion9</td>
<td>73</td>
<td>62</td>
<td>62</td>
<td>62</td>
<td>0:00</td>
</tr>
<tr>
<td>mc</td>
<td>24</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>0:00</td>
</tr>
<tr>
<td>modulo12</td>
<td>40</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>0:00</td>
</tr>
<tr>
<td>s1</td>
<td>462</td>
<td>431</td>
<td>431</td>
<td>431</td>
<td>1:05</td>
</tr>
<tr>
<td>s8</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66</td>
<td>0:01</td>
</tr>
<tr>
<td>train11</td>
<td>89</td>
<td>105</td>
<td>105</td>
<td>105</td>
<td>0:02</td>
</tr>
</tbody>
</table>

The results for all three sort orders A, B and C of list_construction_3 are almost identical. Only the result for the dk16 circuit of order C differs from the other results. So it is most likely that the gen_kernel_lists and the Expr_lists of all the three sort orders are the same.
5.5 *Results with List_construction 4*

*Table 6:* The results for encodings with Chains and the three different methods of P.S.E. with list_construction_4.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Chains</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>CPU (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbara</td>
<td>104</td>
<td>97</td>
<td>102</td>
<td>102</td>
<td>0:07</td>
</tr>
<tr>
<td>bbsse</td>
<td>166</td>
<td>165</td>
<td>168</td>
<td>170</td>
<td>0:20</td>
</tr>
<tr>
<td>bbtas</td>
<td>29</td>
<td>37</td>
<td>35</td>
<td>26</td>
<td>0:00</td>
</tr>
<tr>
<td>cse</td>
<td>278</td>
<td>251</td>
<td>300</td>
<td>263</td>
<td>2:18</td>
</tr>
<tr>
<td>dk14</td>
<td>139</td>
<td>133</td>
<td>133</td>
<td>135</td>
<td>0:25</td>
</tr>
<tr>
<td>dk15</td>
<td>93</td>
<td>88</td>
<td>79</td>
<td>82</td>
<td>0:01</td>
</tr>
<tr>
<td>dk16</td>
<td>350</td>
<td>355</td>
<td>330</td>
<td>349</td>
<td>1:35</td>
</tr>
<tr>
<td>lion</td>
<td>24</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>0:00</td>
</tr>
<tr>
<td>lion9</td>
<td>73</td>
<td>84</td>
<td>76</td>
<td>69</td>
<td>0:02</td>
</tr>
<tr>
<td>mc</td>
<td>24</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>0:00</td>
</tr>
<tr>
<td>modulo12</td>
<td>40</td>
<td>42</td>
<td>42</td>
<td>41</td>
<td>0:00</td>
</tr>
<tr>
<td>s1</td>
<td>462</td>
<td>417</td>
<td>417</td>
<td>428</td>
<td>1:59</td>
</tr>
<tr>
<td>s8</td>
<td>66</td>
<td>73</td>
<td>64</td>
<td>64</td>
<td>0:02</td>
</tr>
<tr>
<td>train11</td>
<td>89</td>
<td>91</td>
<td>86</td>
<td>91</td>
<td>0:02</td>
</tr>
</tbody>
</table>

This construction is the most time consuming one of the four list constructions.
After I've implemented the ideas in the log_decom program and tested it the following conclusions could be drawn:

- The P.S.E. method gets better results than the old state encoding program chains for files with a great number of kernels.
- The P.S.E. method takes a lot more time than the old state encoding program because it has to make $N_b$ simplification and distillation steps instead of one.
- Different list construction methods lead to different results. Some work better on small F.S.M's and some better on big F.S.M's. The overall best list construction for the test-circuits is list_construction_1. This is the construction that uses all common kernels. My assumption that looking for common kernels in mainly external output and nc_expressions apparently didn't hold completely.

More research can be done on the following items of P.S.E.:

- I coded don't cares in the State Transition table to 0. Maybe better results could be achieved by coding them smartly.
Many other methods can be used for generating the gen_kernel_list. I've only implemented four ways.

Encoding states that have not been assigned a value when the gen_kernel_list is empty can be encoded in another way too.

I used P.S.E. based on common kernels in my research. Maybe using kernels in general will lead to better results.

In the last encoding step all the symbolic states disappear in the expressions that describe the F.S.M. this could result in kernels that disappear in the simplified expression. Maybe for this last step another algorithm could be used.

It is also possible to use all four list constructions in one implementation. The encoding that leads to the smallest amount of transistors for that F.S.M. will then be given.

Other methods for P.S.E. could also be implemented it doesn't necessarily have to work with kernels.
Bibliography


Integrated State Encoding and Logic Optimisation


Appendix A

The log_decom enc command

Now that the common kernel based implementation of P.S.E. is explained, the user has to be instructed how to use this new option in log_decom.

First from the .tab file that describes the F.S.M the primary output and ns_expressions have to be derived. This can be done by using the tab2log program that converts the .tab file to a .log file that can be read by log_decom:

Type:     tab2log <tabfile>       (Don’t enter the .tab extension)

After this is done a .log file is created with the same name as the .tab file but with the extension .log. Now start log_decom with this new file as input file:

Type:     log_decom <logfile>    (Don’t enter the .log extension)

at the decom prompt type the following sequence of commands:

Type:     s <ENTER>             Simplification step
        enc <ENTER>             Start the encoding step

You will the be prompted to select one of the four List_constructions explained in this report. e.g. for List_construction_2 type 2:

Type:     2 <ENTER>

Now P.S.E. will be started and it will use the list construction you chose. The
**Integrated State Encoding and Logic Optimisation**

`gen_kernel_list` will be sorted on the amount of cubes in the kernel and `Expr_list` will be sorted on the weight function \( W = \text{amount of cubes in the kernel} - \text{amount of transistors in the combined expression of ns_expressions} \) which form the kernel (The sort order for `gen_kernel_list` can be changed by altering the sort conditions in the source code of List_construction 1 to 4 in the pseoptimise.c file and the sort order of `Expr_list` can be changed by altering the sort condition in the UpdateExprSet function of the stateassign.c file).

After an encoding is made, `log_decom` creates a `.pse` file in which the encoding for each state is stored. The expressions that describe the F.S.M. after state assignment can then be optimised by the following command sequence:

- **Type:** dd <ENTER> \( \text{Complete distillation step} \)
- **Type:** cc <ENTER> \( \text{Complete condensation step} \)
- **Type:** cl = 1 <ENTER> \( \text{Collapse all internals that appear once} \)

The resulting expression represent the encoded and optimised expressions which describe the F.S.M.

---

*Example .pse file (st0 = 00, st1 = 10, st2 = 01, st3 = 11):*

- st0: 00
- st1: 10
- st2: 01
- st3: 11

---

38
File changes:

Veranderingen gemaakt in de source code van log_decom:

toevoegen aan menu.c: enc command
toevoegen aan copycom.c: command ENC
toevoegen aan defines.h: command ENC
toevoegen aan distill.c: Copy_comkern om comkernel_list globaal te maken

types.h:

    struct _expr_set
    struct _expr_list
    GLOBAL kernel_ptr
    GLOBAL kernel_ptr
    int
    char
    int
    expr_list
    expr_set
    expr_set
    cube

    found_kernels;
    comkern_list;
    numstates;
    *input_name;
    Nb, keer;
    Expr_List;
    State_list;
    ns_set;
    state_set;

defines.h:

#define COM_ENC
#define COMS_ENC
#define NUMOFCOMMANDS 62
#define 61
#define "enc"

Sources toegevoegd:

pscoptimize.c
stateassign.c
onestep.c
cases.c
combtocom.c
gupprocs.c
mysets.c
Tablog.c