DESIGN AND IMPLEMENTATION OF A MODULE LIBRARY TO SUPPORT A SILICON COMPILER

Frank Kaiser

Master thesis reporting on graduation work performed from 1.06.86 to 22.10.87 by order of prof.dr.ing. J.A.G. Jess and supervised by drs. R. v.d. Born and ir. L. Stok

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

Computer aided integrated circuit design is developing increasingly because the present technology makes design of more complex circuits possible. The availability of stronger computers accelerates this tendency. A silicon compiler is a tool for automatic design. It needs as input an algorithm in a high level language. This algorithm is translated into a data flow graph in which single operations are distinguished. The graph is transformed into a hardware description. The silicon compiler is build up out of several parts. One part is a module library.

In the module library operations and hardware modules, that are used in the composition of the layout description, are described. A datastructure is developed in which the operation and module objects are implemented. Definition and access interfaces are implemented. Using these interfaces communication with the module library is facilitated. Some parts of the silicon compiler are already implemented. Results are given about the application of the module library in these parts of the silicon compiler.
Acknowledgements

I performed my master work at the Automatic System Design Group of the University of Technology in Eindhoven, The Netherlands. It was part of the project ESPRIT 991, supported by the European Community. I was enabled to analyse and develop my own ideas about the problem described in this report. I would like to express my gratitude especially to my supervisors, drs. R. van de Born and ir. L. Stok, who provided me of ideas at significant moments in the project, and to prof.dr.ing J.A.G. Jess who leads and guides the Automatic System Design Group.

Frank Kaiser
October 1987
CONTENTS

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

2. System overview ............................................. 3
   2.1 the silicon compiler ................................. 3
   2.2 operations and modules ............................ 5
   2.3 data structure ...................................... 5
   2.4 interfaces ........................................ 8

3. Object implementation ..................................... 11
   3.1 operations ........................................ 11
   3.2 modules ........................................... 13

4. Interfaces ................................................. 20
   4.1 access interface .................................. 20
   4.2 definition interface ............................... 22
   4.3 screen interface .................................. 25

5. Library maintenance ...................................... 27
   5.1 file organisation .................................. 27
   5.2 object and variable dump ......................... 27
   5.3 system startup ................................... 27
   5.4 change of standard objects ....................... 28

6. Conclusions .................................................. 29

References ..................................................... 30

Appendices ...................................................... 33
   access functions ........................................ 33
   syntax of definition functions ....................... 39
   timing analysis of access functions ................. 44
LIST OF FIGURES

Figure 1. Hardware synthesis system . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
Figure 2. relations between operations and modules . . . . . . . . . . . . . . . . . . . . . . 5
Figure 3. object data structure and interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
Figure 4. operation hierarchy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
Figure 5. aspects in the standard operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
Figure 6. An example of an operation vector . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
Figure 7. example of hierarchy in module definitions . . . . . . . . . . . . . . . . . . . . . . 14
Figure 8. standard module object vector . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
1. Introduction

In the Automatic System Design Group of the University of Technology in Eindhoven, tools are developed for design of integrated circuits. One project in this group is done in order of the European Community, and is named ESPRIT 991. This project concerns the development of a high level circuit synthesis tool or a silicon compiler.

Such a silicon compiler is developed to provide a highly automated design of VLSI circuits. The function that must be fulfilled by the circuit we wish to design, is given by a behavioural description of an algorithm in an abstract, high level language, for instance LISP, C or PASCAL. The silicon compiler transforms the functional description into a hardware description. Therefore first the functional description is decomposed into a description in single operations. Next, the operations are mapped on a set of hardware modules by the silicon compiler. Finally the entire hardware description results.

Automation of the design process gives several advantages. Because of the availability of stronger computers, design time decreases and larger and more complex circuits can be developed. Verification and simulation may be performed by this silicon compiler. This way the correctness of the result is improved, compared to conventional design techniques. By using a high level input algorithm, the designer is able think about the design at a higher abstractional level. Unexperienced users are now enabled to design circuits. They do not need expertise about the technology in which is designed or about the several techniques, necessary to obtain a correct and reliable design.

One tool within the silicon compiler is a hardware module library. This report describes the design and the implementation of this tool. In this module library, modules are stored that are used in a circuit design project. Access is provided to the data stored in the library and functions are provided to define or update module descriptions. Thus the results of the effort done in previous module designs and the knowledge obtained, is used in next circuit design projects.

Some aspects of methodological nature and techniques that are used in the software engineering practice must be noted. The silicon compiler and all its composing tools are initially developed on a Hewlett-Packard System 9000 HP-UX computer. Further developments will be made on an Apollo Domain system. Nevertheless, the used hardware system may not have influence on the final result.

The silicon compiler and more specific the module library is designed in Common Lisp. The editor EMACS, that is used, is particularly build for Lisp applications and software development. A special dialect of Common Lisp, HotLisp, is available on the HP system, whilst Common Lisp is available at the Apollo system.
Matters like security and integrity of the library contents in this multi user environment are not discussed in this report.

The design and implementation of the module library is split up into several parts which are discussed in the report. In the next chapter a system overview is given of the silicon compiler and its composing elements. A functional analysis of the module library is made, and the requirements are determined. Objects, data structures and interfaces are discussed. In chapter 3, a structural analysis is made of the operation and module descriptions. Standard objects are defined.

In chapter 4 the interfaces that provide the communication between the user and the data contents of the module library, are described. The functions that are implemented for this purpose are discussed. In chapter 5 some notes are given concerning the development and maintenance of the module library. Chapter 6 contains some results and conclusions are given. A comparisons between different implementations of the module library is made. In the appendix references and syntax descriptions are included. Documentation of the produced software is given in the software files itself.
2. System overview

2.1 the silicon compiler

A silicon compiler or hardware synthesis system, is a system that transforms a functional description into a layout description. The silicon compiler is split up into several tools, which are discussed in the following. The subsystems that are distinguished in the figure below, each represent a phase in the circuit synthesis process. Each tool produces intermediate results that form the input for the next tool. These results may be examined and if necessary, corrections or changes may be applied, before the next tool is used. An outline is given in the figure below. In the picture, boxes and ellipses are distinguished. Ellipses represent tools that process the input. Boxes contain the intermediate results of the several tools.

The high level input language gives an algorithmic description of the function to be fulfilled by the circuit. The parser analyses the description and converts this algorithm to an abstract syntax tree. This tree is converted into a demand graph by the demand graph constructor, which has been described in [Stok86]. An optimiser deletes some inefficiencies of the demand graph, converting it into a functionally equivalent demand graph.

A decomposition from the algorithmic description into single operations is now obtained. To transform this graph into hardware a hardware generator is developed that maps the operations on a hardware description. For each operation or functional group of operations a hardware module can be implemented in the library. The hardware generator [Woud87] finally produces the data path description and the state machine, with the help of the cost estimator [Enge87] and choosing modules from the library.

During the synthesis of a circuit, in each tool amounts of external data are needed and internal data is generated. This data comprises information of the functional and structural aspects of the circuit under design. Internal or temporary data, is generated and has to be stored during the processing time of each subsystem. After completion of the process, most of this data is not needed any more. The responsibility for the handling of this temporary data, is given to each subsystem itself. It must provide its own data structure.

The other type of data is external data and is of a more global and permanent character. This data is not restricted to be used in one tool, or one design project. This data describes operations and previously designed modules and circuits composed of modules. These modules and circuits may have been developed in the past and are therefore available for use in another design project.
Concludingly, it is useful to provide a separate subsystem for external data management. This subsystem, a *module library*, is responsible for a proper handling of operation and module data storage and retrieval actions.

Finally the *layout generator* converts the symbolic data path description and state machine into a detailed layout description. For each module a layout description is comprised by the module library. From this layout description an integrated circuit is produced.
2.2 operations and modules

In the module library, operations and modules are described. These descriptions are comprised by a hierarchical set of structured standard objects.

The operation descriptions form the set of operations that is implemented in the module library system. Some aspects of these operations are important during the circuit design and are therefore recorded in the operation description.

Operation descriptions are mapped on module descriptions by the silicon compiler. The modules perform one or more operations out of the operation set. A relationship exists between the set of operation descriptions and the set of module descriptions. This relationship is showed in the figure below, where, as an example, a small set of operations is performed by an arbitrary set of modules.

![Diagram of operations and modules](image)

**Figure 2.** relations between operations and modules

2.3 data structure

The module library comprises details of the description of operations and modules, which may be seen as sets of data or objects. Objects are fully described by a name and aspects
with values. To identify an object, it is necessary to know its name. The objects are used within the module library and by the tools of the silicon compiler that are supported by it. The object name must be unique in this system and is determined by the user who defined the object. Changes of the object identifier may lead to inconsistencies in the module library. The set of description data of an object is contained in the aspects with subsequent values. The contents of the object description are obtained by referring to it by its name.

Two standardised objects are described. To be able to place or store this data in, respectively retrieve it from the module library, a set of functions is provided by the module library. These functions operate on the data and the data structure in which the operations and modules are described. Thus two major items of interest may be distinguished. First an internal object oriented data structure is implemented which provides an efficient and correct response to data storage and retrieval requests. Second, interface facilities through which these requests are handled and through which the communication is realised between the module library and the users, are developed.

The data structure must provide consistency, efficiency and extendibility. Furthermore, the possibility to derive new objects from existing objects is implemented. This way previously developed object data may be used to base new objects on. New objects then inherit aspects with subsequent values of existing objects.

An impression must be obtained of the complexity of the data management system of the module library. There are several aspects concerning this complexity. One aspect is the complexity of the structure in which the objects are captured. The data structure must be able to assimilate extensions of the set of objects, without affecting the consistency and the efficiency.

The second aspect is the complexity of the module description data, which is coherent to the internal complexity of an object description, and the extent or size of the circuit description to be stored. Though in practice many relatively simple and few complex modules occur, the module library must be able to accept whatever object is developed. As long as storage and retrieval of data is at hand, the implementation of these functional actions should not put any limitations to the complexity of the data to be stored.

2.3.1 Flavors
For object oriented programming, studies have been made of the data structure in which the objects could be implemented [Wolf86], [Zara85], [Weis86], [Afsc86]. One system that is developed is called Flavors. Flavors is an object oriented extension of the programming language Common Lisp. It provides a flexible means to define an extendible set of object data structures or flavors [Wins84]. Flavors is described often in the literature concerning design data management. A few characteristics of the system are analysed in the
In Flavors an object is described by a data construct, called a flavor. An extendible number of aspects may be described in a flavor. Appending aspects to a flavor is done by defining a new flavor based on an existing one, with the new aspect included in the definition statement. One or more flavors may be mixed to obtain a new flavor that has the aspects of both the parent flavors. The aspects of the existing flavor are then inherited by the newly defined one. Newly occurring aspects of an object may be implemented directly by defining a new flavor. It is not necessary to redefine the existing flavors.

Out of each flavor, instances may be defined, which have all the aspects of the originating flavor. The flavor is a kind of template that prescribes the form of its derived instances. The values of the aspects are assigned on instantiation. Multiple instances can exist, all based on a single flavor.

The aspects of a flavor are operated on by methods, or functions that retrieve or store data in an aspect. Internally, Flavors places the data in a list structure. A list is a sequence of elements of which the first points to the second, the second to the third element and so on. To reach the last element of a list, all element must be passed. The methods that operate on flavors are manipulating these lists. On our system Flavors is not provided. Therefore the data structures that may be implemented in Common Lisp are investigated next.

2.3.2 common lisp

The programming language in which the module library is implemented is Common Lisp. It provides the two important data structures, which are investigated on their applicability, namely symbols with property lists and vectors.

Symbols with property list are lisp provided data constructs. It is possible to assign an aspect, or property, with a value to a symbol. The Lisp functions put and get are available for this purpose. Retrieval of data is done by travelling through the property list. Response time increases linear with the length of the list.

Vectors, data types of one dimension, are implemented in an indexed way. This means that there is an order in the elements in a vector. Vectors elements are directly accessible, when the appropriate index of the element in the vector is given. Response times do not depend on the length of the vector and are therefore constant. This gives a faster response to a retrieval request than a property list structure.

Creating vectors is a more complex job than creating symbols with property lists. Lisp provides functions for the manipulation of property-lists. In the module library system a set of definition functions must be created that takes care of the creation and update of
the vectors. Furthermore the organisational overhead by the module library system must be handled. This task is executed by an interface between the user and the vector data structure. This intermediate definition step will do the storage of object data and a fast retrieval response is not obstructed.

Both methods have been implemented to be able to analyse there performance. Comparisons are made between them, concerning the time response to access requests. In the appendix numerical results of these experiments are found. Better results are obtained with the vector datastructure. Therefore, this structure is further implemented.

An object vector in Lisp is initially defined with a fixed length. The positions or indexes of the elements are fixed too. This way the storage and retrieval actions can be administrated by the module library system properly. Within these limitations it is well possible to design a data structure in which values can be assigned to the aspects or elements of a vector. New vectors of the same length may be derived from existing ones, by inheriting the values of the vector elements. Two general standard objects are defined, that are applicable in the occurring objects. They are operations and modules.

Concludingly, Flavors, which is described in the literature concerning design data management gives flexibility to the designer, which is not provided in the same extent by a vector data structure. The retrieval response of this vector structure however is expected to be better than of Flavors.

2.4 interfaces

The module library is communicating with a number of tools of the silicon compiler. This is done by aid of a standard set of functions, forming the interfaces. This set of functions operates on the object data.

In practice before the module library is accessed by the silicon compiler, modules are developed and brought into the module library by a storage mechanism. After that, hardware design will take place, during which information is retrieved from the module library. These two different tasks of the module library led to the implementation of two interfaces, that each provide a set of limited set of standard functions. One interface includes functions for storage, update and deletion of operation and module data. The other provides read-only access functions that are used during circuit design.

Storing data is done via an editing mechanism. On one hand, a simple editing mechanism in which the user directly produces the data structure, gives a simple input language. By using this language, objects are easily defined or changed. Because of this directly implemented data structure however, retrieval is a costly and inefficient process. On the other hand, when a fast response to a retrieval request is desired, the system asks for an internally efficient object structure. Editing and retrieval of data thus give rise to a
conflict because there is a duality between the complexity in those two tasks [Zara85] in a data management structure.

By the use of interface facilities, the data structure as it is implemented in the computer system, is hidden from the user. The task of the module library is to produce object data on request of the user and to store new object data. It results in an external user view of the objects, without the user having to concern about the details of the implementation. Another advantage of interfaces is their transparency. Changes in the internal data structure may be applied, without changing the use of the interface.

The module library is addressed during the circuit design process. Most transactions are data retrieval requests. They occur often and are done by processes that are time consuming as it is. As most retrieval transactions with the module library are short and frequently, a fast response to retrieval requests is of main interest. Still, a simple and efficient editing facility is desired.

It is possible to come up to both of these wishes by approaching the interface problem from two sides. Separate storage and retrieval interfaces are then distinguished. The interface that provides storage is a mechanism which transforms the plain external user defined input structure to an efficient internal data structure. The retrieval or access on the other hand is performed by a set of functions or interface that ask simple input as well. It acts directly upon the object data structure. It solves the duality in editing and retrieval.

![Figure 3. object data structure and interfaces](image-url)
The computer aided circuit design environment is one of development of new objects, or change of existing objects. Objects may be appended to the library, updated or deleted. Change therefore is rule rather than exception. The definition interface functions must provide a flexible and consistent tool. The changability and the extendability are important features of the module library and must be covered by the data structure and its interfaces.

To increase consistency of the module library, changes at one place may not affect data at another place in a destructive way. Therefore a structure must be found where changes are adapted by the system consistently. A strategy of one-entry-update is implemented as much as possible to protects the system. Changes then ripple through the system until the appropriate data in the module library is updated. As multiple users have access to the module library, update permission must be controlled. This control can be provided by the operating system that is used, UNIX or the EMACS system, on which the module library is implemented.

In total, in the module library system, 3 interfaces are distinguished. Two were already outlined, namely the definition interface and the access interface. The third interface is a screen facility, that provides a set of interactive functions that facilitate editing and retrieval of object information on screen. Main interest of this interface is user friendliness. Aspects like speed and efficiency are of less importance.
3. Object implementation

3.1 operations

Two types of objects are described in the module library, namely operations and modules. In this chapter operation descriptions are further analysed. Operation descriptions are needed in the design tools that precede the hardware generation. In these tools the algorithmic input description of the silicon compiler is decomposed into a set of single operations. This set is analysed and if possible optimised by aid of the information contained in the library.

3.1.1 operation set

At this moment a small set of operations is implemented that may be divided into several subsets of a certain class. In the future however, this set may be extended so that the subdivision may have to be revised. However at this moment, based on the implementation of the demand graph constructor a division into io-transfer operations and non-io-transfer operations is made (see figure). The non-io-transfer operations are subdivided into dyadic and monadic operations, i.e. operations that need two operators, and one operator respectively. The dyadic operations are further divided into boolean, arithmetic and relational operations, taking into account that the latter are always dyadic. The dyadic arithmetic operations are divided into two groups, namely, the additive and the multiplicative operations. The monadic operations are subdivided into two subsets, namely boolean and arithmetic operations. The figure includes operations that are implemented at this moment.

3.1.2 standard operation object

Before implementation of the operation set is possible, a data model of an operation description is determined. An operation has several characteristics. to describe an operation a standard operation object is developed. The standard is described in the variable *lib-operation-aspects*. By convention, Lisp variables are surrounded by stars. The number of elements in this variable determines the length of the operation vectors, later to be defined. The index of the several aspects of an operation are also determined from this variable, namely the position of the aspect in this variable is directly the index of the aspect in an operation description. This indexation is used by the definition and the access functions that refer to operations. When an operation is defined, a vector is created according to the standard given in the mentioned variable.

First an operation description has a name which is the system-wide used function symbol, for instance "+", "nand" or ">=". Second, this object has aspects with values. These aspects give a description of the operation, as is needed by the several tools of the silicon compiler. The values of these aspects follow from the type of the operation, and are
based on general mathematical rules. Third, the operation has a global function definition, according to which it performs an action on one or more operators.

At this stage of implementation, an operation has four aspects, namely class, commutativity, associativity and modules. It is well possible that in the future, more complex operations will be described, which are not covered by the standard operation object. Appropriate aspects may be appended to this standard object if necessary.

Class contains information about the class in hierarchy of operations as mentioned in the figure above. All elements of the path that is followed are included. Commutativity and associativity are boolean variables that record whether an operation is commutative respectively associative, or not. Modules, finally, gives a list of defined modules, that perform the particular operation. This list however, only has a value after modules have been defined.
3.2 modules

3.2.1 module descriptions
The silicon compiler provides a mechanism that maps operations on hardware modules. Both kinds of objects are defined in the module library system. Modules are sets of design data, used to compose a hardware description. The description comprises a module name and aspects with values. The modules are placed in a larger structure by the silicon compiler, thus composing a circuit.

These modules are high level primitives, which are an abstract description of an electrical subcircuit. This set of modules is extendible by definition of new modules by the user. Module descriptions are based on a parent, which is in itself another module description. In the existence of parents for each module lies a hierarchical structuring mechanism of the set of module descriptions. The highest level parent is a generic module template. This generic module template gives default values for the aspects of a module. Several modules are derived from this generic module template. Again other modules are derived from these modules. This way a path of parents is created. Following this path points out in which steps a module was designed. When a module is defined, its aspects get values...
Three ways through which modules get their values, are distinguished:
- assigning values by an aspect-value list
- inheritance of values of a parent
- fixing parameters, using a parameterised module description as parent

In an aspect value list, a number of aspects are included, with subsequent values. The list is a part of the argument of a definition function that is described later. As described, new objects can be defined by the user, but they can also be derived from an already existing object description. Two methods provided for this derivation mechanism. One method is parameter fixing. The new module is derived from a parameterised module description. The parameterised regular expressions that are found in the parent, are evaluated whilst a parameter value is given. If one of a modules parameters is fixed, this parameter is deleted from the list of parameters in the aspect parameters. When this aspect has the empty list as value, it is called a static module description, otherwise there is talk of a parameterised module description.

The other method to assign values to a module description, is inheritance of the values of a parent's aspects. This method completes the former two. When aspects are not included in the aspect-value list, or not changed by fixing a parameter, values that are found in the description of the parent are inherited by the new module description.

To explain the derivation of objects, or definition out of existing ones, attend to the following example. Let's suppose that an object exists, which performs the operation +. In other words, the value of the aspect operations is +. No other aspects are defined for this object. From this parent object, a new object may be defined, that performs the same operation as its parent, but also gets new aspect values, for instance an area of 160 square microns and a dissipation of 50 mW. Only these new aspects must be listed in the argument of the definition function. The operation that it performs, +, is inherited from
its parent. Thus, a proceeding development of more refined objects is facilitated, without the user having to define the new object entirely.

3.2.2 standard module object
Before implementation is possible, a data model of a module description is made. The requirements for the standard module object are determined by the tools of the silicon compiler that make use of the modules. The module descriptions are placed in a standard module object. The standard is described in the variable *lib-module-aspects*. The number of elements in this variable determines the length of the module vectors, later to be defined. The index of the several aspects of a module are also determined from this variable, namely the position of the aspect in this variable is directly the index of the aspect in an module description. This indexing is used by the definition and the access functions that refer to modules. When a module is defined, a vector is created according to the standard given in the mentioned variable. All occurring modules must fit in this standard object. As said before, these objects are described by a name and aspects with subsequent values. The name of a module is a system-wide unique symbol, that is defined by the designer of the module. The aspects describe the physical structure, the behaviour (algebraic, as well as physical) and the interface of the module. Furthermore design administration data is kept in the module description.
The algebraic behaviour is described by the aspect \textit{operations}. This aspect describes in a list of one or more symbols, which operations the module is able to perform. The value is assigned through an argument in the module definition function.

The physical structure of a module is comprised by aspects like \textit{length}, \textit{width}, \textit{area}, \textit{layout} and \textit{technology}. From these aspects, a characteristic as \textit{aspect-ratio} may be derived. Sometimes only the \textit{area} is given and length and width are derived in a later phase in the design process. The layout and technology information is used for mask generation, later in the design process. The values of the aspects \textit{length} and \textit{width} may be integers, but also parameterised expressions may occur. For instance the values may depend on the bit width of the operators, which is seen in the example. Thus the user can derive the value when a parameter-value is given, by evaluation of this expression. The value of the aspect \textit{technology} is a string that denotes which technology is used for the module design.
The aspect layout may be a pointer to a layout file, existing elsewhere in the circuit design system, but also another way of pointing at the layout-information may be implemented. This matter is subject to discussion but beyond the scope of this report.

```
small-alu

<table>
<thead>
<tr>
<th>aspect</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>(* bits 10)</td>
</tr>
<tr>
<td>width</td>
<td>25</td>
</tr>
<tr>
<td>technology</td>
<td>&quot;nmos&quot;</td>
</tr>
<tr>
<td>layout</td>
<td>a-pointer</td>
</tr>
</tbody>
</table>
```

The physical behaviour of a module, at this moment, is described by aspects as there are dissipation and delay-table. Integers and expressions may be assigned to these aspects. The values of these aspects depend on what operation is performed. The dissipation however is given by a maximum value that may occur in a certain situation. The module must always be able to deal with this dissipation. The value of the delay time is important in the data path description and depends highly on what operation is performed. Therefore the information is written in a table with for each operation its particular delay time.

```
small-alu

dissipation

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ . 16)</td>
<td>75</td>
</tr>
<tr>
<td>(- . 17)</td>
<td></td>
</tr>
<tr>
<td>(and . 10)</td>
<td></td>
</tr>
<tr>
<td>(not . 8)</td>
<td></td>
</tr>
</tbody>
</table>

delay-table
```

The module interface aspects comprise information about ports and signals through which the module connected to the outside world. The input-list and the output-list give information about the input-ports and output-ports like the number of ports, the names and the bit width of these ports. The control-list tells more about the signals that must be put on the control ports to obtain a particular operation to be performed in a multi-operation module, for instance an alu. In the control-table information is contained about control-signals that tell a multi-operation module which operation to perform. A module may have one or more control ports, each a specific number of control lines wide. The value of control-table is a list of vectors that have as elements the settings of the control lines for a specific operation execution. It depends on the operation performed by the module, what input or output ports are used. For instance a monadic operation uses one input port only, whilst a dyadic operation uses two. Likewise, the used output ports are
in some applications operation-dependent. Therefore two tables input-table and output-table are defined in which for each operation the used input ports, respectively output ports are included.

The supply-list keeps track of the supply-connections to the outside world of a module.
As said before aspects may have parameterised values. A list of parameters that occur in a specific module description is kept in its aspect parameters. This list is included in the module description by the user. When a module is derived from a parameterised module description by fixing a parameter to a value, the name of the parameter is deleted from this aspect automatically. In the aspect parents a list is kept of parents of the module. This list points out a path by which this module is derived. It starts at the generic module template and so on.

```
small-alu

parameters
  (bits)
  (generic-module-template
      big-alu)

parents
```

Design administration data comprises information about the designer, the person who may be referred to when questions arise. The status of the module is kept, which tells if the module design is delivered or has reached only a preliminary state. This information is helpful for a user who is examining the reliability of a design and weather its advisable to use it or not. A module documentation may be written in the aspect documentation. At this point an extensive description of the module characteristics and its performance may be written in a text readable by humans. For special development functions, the aspect boolean-function is generated when the module is defined. The value of this aspect is a string that is used by a tool described in [Dobb87].

```
small-alu

designer

status

documentation

boolean-function

"your name"

"preliminary"

"a text ...

"lib-bfg-small-alu"
```

The set of aspects described here at this moment meets the requirements of the tools that use the module library. Furthermore it gives an impression on what kind of aspects are of importance. Similar kinds of modules are described in literature [Foo86], [Suck79], [Hosk79], [Nien79].

If necessary aspects may be deleted, or new aspects may be defined. How this is done, is described in the chapter library maintenance.
4. Interfaces

4.1 access interface

The access interface provides a set of functions through which object data is retrieved. This interface is for reading purposes only. It responds fast to user requests. The set of access functions is extendible to meet the needs of the user. Two groups of functions are distinguished, namely those that act on operations, prefixed \textit{lib-operation-}, and functions operating on modules, prefixed \textit{lib-module-}. The functions can be used interactively or by a program that is a part of the silicon compiler. The access functions are implemented such that they make no use of other access functions. The data structure is directly accessed by referring to the vectors.

In some access functions, sets of objects have to be searched through. The search mechanism in this module library is based on lists, that guide the search process. These lists contain sets of object names. The lists are walked through when a subset is investigated.

Operation sets and in a minor extent, module sets are subdivided into subsets. The operation objects are subdivided according to the hierarchy, described previously. The subsets are implemented in Lisp \textit{variables}. The use of variables is efficient in this case, because the set of operations is a limited one and the hierarchy in operation descriptions is well defined.

For modules it is more complex to use variables than for subsets of operations. The hierarchy in module descriptions is not as plain as in operation descriptions. A larger variety of access functions exists and administration of the variables would give a large amount of overhead. The return or efficiency is not compensating sufficiently for the investment done.

When information about the subsets is requested, direct access to the variables gives a faster response than going through the entire set of operation or module objects. The access user has no direct access to the variables, because they are for internal use only. The variables are updated whenever a new operation is created, or an existing one is deleted.

The functions that retrieve information about subsets of operations directly access one of the variables. This decreases the time needed for searching in the entire set of operations in a substantial extent.

Access functions approach the module library from two sides.
Object names are given as argument of an access function, and its aspects are returned.
- Aspects of a specific value are given and objects that match this value are returned.

In the following, the functions to access the operation and module descriptions are described. A list of these functions, with their syntax and documentation is found in the appendix.

### 4.1.1 operations

The standard operation objects are implemented as vectors. As described before an operation has the aspects class, commutativity, associativity, and modules. For each aspect, a function is created, that provides retrieval of aspect values of single objects. Direct access to the vectors is applied to obtain the desired information. Four macro's return the values of the respective aspects. Two functions, concerning associativity and commutativity, return boolean values true or nil on call, depending on the fact whether the functions are associative respectively commutative or not. The aspect modules is treated differently. The argument to the access function is a list of operations. The result is a list of modules that perform all operations of the argument. It is the intersection of the aspect modules of all the operations included in the argument.

The class of an operation denotes whether an operation is an io-transfer operation or not, if it is monadic or dyadic, and so on. Some access functions return a subset of operations of a specific class. These functions need no argument, and they access the variables that are described before. Other access functions need as argument an operation. It is asked whether a specific operation is a member of a class or not. The values returned are nil or non-nil.

### 4.1.2 modules

Standard module objects are also implemented as vectors. The access functions are prefixed lib-module-. The standard module object has at this moment about 20 aspects. Access functions are defined, that directly return the value of each aspect of a module. Next, a special module access function is defined that needs two arguments, namely the aspect name and the module name. However, for reasons of time performance not only this function is implemented but also a specific access function for each aspect.

Some aspects, namely the input-table, the output-table, the control-table, and the delay-table contain tables of operation dependent information. Functions are implemented for retrieval of information that is specific to one operation. As argument of these functions, both a module name and an operation is needed in the argument. Besides the module concerned, also an operation has to be included in the argument.

For searching and management purposes, a list of all defined modules is kept in a
variable. This list is referred to by several functions that search through the set of defined modules. The value of this variable is accessed itself by an access function.

To improve the access performance, one subset of modules is defined. It is a set of modules that are not parameterised: static modules. These modules have integer values for aspects like length, width, area and so on. For direct cost calculation, integer values of these aspects are needed. Parameterised expressions are no integers and can therefore not be handled. The subset of not parameterised modules is kept in a variable. An access function is implemented to retrieve the contents of the variable. Furthermore, this variable is used by another access functions that returns a list of modules that match a certain specification. The function optionally searches in the entire set of modules, or only in the set of modules that have no parameters.

A specific set of modules is the family of multiplexers. A special function is developed in the access interface for retrieval of the names of specific multiplexers. It needs two arguments, \emph{ports}, and optionally \emph{bits}. When this function is called with only \emph{ports} as argument, multiplexers are returned, of which the number of input ports is \emph{ports} and of which the parameter \emph{bits} has not yet been set. On the other hand, when \emph{bits} is also included in the argument, a list of multiplexers is returned that have \emph{ports} input ports and precisely \emph{bits} wide input and output ports.

4.2 definition interface

The definition interface provides functions for implementation of operation and module descriptions. Two standard objects are developed, one for operations and one for modules. Based on these standards, objects may be defined by the user. Objects can be stored, updated and deleted. This facility enables the user to edit operation and module descriptions.

4.2.1 Operations

The definition of operations is done by a special definition function. This function creates a vector with 4 elements, one for each aspect. The values of \emph{class}, \emph{commutativity}, and \emph{associativity} are assigned according to the aspect-value list of the definition function. The value of the aspect \emph{modules} may not be given by the user, but is updated automatically every time a \emph{module} is defined that performs the particular operation.

The function used is

\begin{verbatim}
(lib-def-operation <operation> <aspect-value-list>)
\end{verbatim}

As an example the operation + is defined by:
(lib-def-operation '+
  '((class . (dyadic arithmetic additive))
    (commutativity . t)
    (associativity . t)))

Checks are made on the argument that is given by the user. The user is notified when he
tries to include the aspect modules in the aspect-value list. Furthermore a check is done
on the consistency of value of the aspect class, which must be a list. For instance an
operation is not allowed to be both dyadic and monadic.

The operation symbol has a global function definition, outside the module library.
Whenever an operation is defined, the operation is overloaded. Furthermore, the
hierarchy variables that are used in the access facility, are updated, accordingly. In the
definition phase some overhead work is done, but access is faster.

There is no parent or inheritance mechanism through which new operations can be
derived from others. The aspects of the operation description, that are not included in
the aspect-value-list get the nil value. Update of an operation description is done by
redefining the operation. The old definition is then overwritten.

Deletion of an operation description is facilitated by a special function. When called, it is
first checked if the operation is referred to by any modules. If not, deletion is allowed
and is actually done. Else, deletion is not allowed and the user is signaled. Deletion is
only done by deleting the operation from all the appropriate variables that were described
before. Uninterning of the operation symbol is not allowed, because of the global
definition of an operation. It would destroy more information, than was brought in by
the module library.

4.2.2 modules
A module is created by call of one of the definition functions. There are two mechanisms
of defining modules. The first is a function that creates a vector by assigning values to
aspects of the vector. The function call looks like:
(lib-def-module <name> <aspect-value-list> [ <parent>] )
In the argument three parts are distinguished.

— the module name

— an aspect-value list

— an optional parent

The name identifies the module description. It is chosen by the user and it is known
systemwide.
The aspect-value-list contains aspect-value pairs. The aspects that are mentioned in the first part of the aspect-value pair are assigned the values of the second part of the pair. The order in which the aspect-value pairs appear in the list is not relevant. The definition function takes care of a proper handling. Not all aspects have to be mentioned in the list. Omitted aspects get the value that is found in the description of the parent. Some aspects are not allowed to be defined by the user, because they are recorded automatically. The aspects concerned are boolean-function and parents. The list parents is updated by appending the name of the parent, mentioned in the argument of the definition function.

This parent is optionally included in the argument. The parent is another module description itself. If this parent is not included in the argument, a default parent is used, described in the variable *generic-module-template*.

Update of existing module descriptions is possible by calling the same definition function. A module is then defined with itself as parent. The vector is then overwritten.

When the definition function is called it is first checked if the module identifier already exists. The user is then notified that this module description is overwritten. Next the existence of the parent is checked. It is also checked if the aspect-value list is consistent. The set of operations, that a module performs is the basis for several tables in the module description. It is checked if these tables are consistent with the set of operations. First it is checked if the input- and output-list provide enough input respectively output ports. Next the control-table is checked on the presence of a control vector for each operation. A number of other checks are made to take care that no inconsistent module descriptions are allowed to be implemented.

When no errors are found the aspect-value list is processed, so that the according vector elements get the appropriate values. The aspects that may not be user-defined are automatically assigned their appropriate values. Next, the vector is interned, and the appropriate variables, that contain the newly developed module description are updated. These variables are *lib-modules* and eventually *lib-no-parameter-modules*.

The second important mechanism through which modules are defined, is parameter fixing. Modules can have aspects with parameterised values. At this stage of implementation of the silicon compiler, two parameters are distinguished, ports and bits. Ports is a parameter for the number of input ports of a generic multiplexer module. Bits gives the bit width of a single input or output port. For both parameters a special parameter fixing function is developed, They are (lib-fix-mux-ports <name> <ports> <parent>), and (lib-fix-bits <name> <bits> <parent>).

When these functions are called, a new module is created. Again, it is a vector, initially
based on the parent. The parent has some aspects with parameterised expressions. These aspects are evaluated with the value of the parameter taken from the argument of the function call. Not all elements of a vector can be parameterised, and furthermore, each element is treated differently. A parameter fixing function therefore must be developed and handled with care and is aggravated on the standard module object as it exists at this moment.

In the implementation of this moment, before bits are fixed, ports must have been fixed. This is particularly done for the aspect input-list of the multiplexer description. When ports are fixed, a list results in which the parameter bits is still present. Only after ports are fixed, the function that fixes bits can be executed properly. When ports are fixed, the following aspects are processed. The first is the control-list, of which the number of control lines is determined. Next, the number of input ports of the input-list, that is available in the new module description, is determined. Finally, the normal actions that are taken when a new module is defined, are performed. These are variable update and interning of the new vector.

When bits are fixed, aspects as length, width, area and dissipation are evaluated. Usually, these aspects have simple expressions as content. Furthermore, the delay-table, the input-list and the output-list are updated. When a parameter is fixed, it is deleted from the list in the aspect parameters, an aspect of the module description.

Deletion of a module from the library system is done by calling the function (lib-delete-module <module-name>). This function first checks if the module is not used as a parent. Parents may not be deleted, because then inconsistencies in the modules that were derived from this parent, occur. If deletion is allowed, the appropriate action is taken. That means that the two variables that contain module names, are updated. Next the name of the module is deleted from the aspect modules of the operations that were performed by this module. Finally the name of the module is uninterned.

4.3 screen interface

The screen interface provides a user friendly means, which incorporates features of both the definition and the access interface. It is build to be used in an EMACS environment. It is for on-line use only, where the user can interactively retrieve object data and append, change, update or delete object data. It gives a clear and bright view on the system. Through a menu oriented approach the user enters the system.

A small number of functions are forming the interface at this moment. The functions are described one by one. Through one function it is possible to interactively define a module description: (lib-def-module-on-screen [ <parent> ]). When this function is called, a buffer is created in which a call of the module definition
function \((\text{lib-def-module } \langle\text{arg}\rangle)\) is written. For the aspects and values, defaults based on the \(*\text{generic-module-template}\*\) are written. The user now may edit this text to obtain the required module description. The screen interface makes use of these checks in an extensive way. The user is notified when mistakes are made. The aspect-value list of an object is given aspect by aspect. Whenever an aspect value is defined in the interactive session, other aspects that depend on this value, are calculated and printed on screen. The user is allowed to change this precalculated, but the possibility to make errors is reduced. When the module definition is evaluated, the checks on the arguments to these functions, described in the definition interface, are performed.

Similarly, an operation description may be edited. The function is called by:
\((\text{lib-def-operation-on-screen} \ [\ <\text{operation}> \ ])\).
If the argument is given by the user, an existing operation description can be updated, otherwise a new operation definition is created.

For access on screen several functions are implemented, based on functions of the access interface. One is \((\text{lib-screen-aspect } \langle\text{aspect}\rangle \ [\ <\text{module-list}> \ ])\). The names of the modules and the values of the \langle\text{aspect}\rangle in the argument is printed in a buffer. Only the modules, included in the \langle\text{module-list}\rangle are printed. Default, all modules are accessed, but a list or a regular expression that returns a list, may be given as the second argument.

Another function, \((\text{lib-screen-module } \langle\text{module}\rangle)\), prints the aspects and values of one module in a buffer. This facilitates a read-only view of the description of \langle\text{module}\rangle. Aspects, with value nil are not printed.

To have a view over a specific object vector, the function \((\text{lib-screen-vector } \langle\text{vector}\rangle)\) is called. It creates a buffer, in which the entire operation or module vector is printed. The aspect names are not included, only the values in the order that is given by the respective standard objects.

The screen interface may be expanded in the future, according to the wishes of the users. When the module library is implemented on another system than the HP-EMACS, function descriptions may have to be adapted.
5. **Library maintenance**

5.1 **file organisation**

The module library as described in this report is build up out of several parts. A vector data structure was described and interfaces that operate on this data structure. The definition interface comprises the functions mentioned in the relevant chapter. Besides these functions many supporting macro's and functions are defined. The entire set is found back in the file libdef.

The aspect-value lists in the definition functions ask for a special syntax. This syntax is described in the files libsyntop and libsyntmod. The access macros and functions are all included in the file libaccess.

The definition and access of objects are supported by a number of variables. These variables are defined in the file libvar. Also some initial values are assigned to these variables in the same file. One special variable, the *generic-module-template*, is contained in the file libgen. This construction is forced by the order in which files have to be loaded. The screen interface functions are all comprised by the file libscreen.

During the development of the module library, an arbitrary set of operations and modules are used. The operation definitions are included in the file libops, whilst the module definitions are comprised by the file libmods.

5.2 **object and variable dump**

One feature that has not been described yet is the possibility to dump object descriptions. The definition of objects is rather time consuming. When the necessary objects are defined, and no changes in the set of objects are likely to occur, the object vectors may be dumped to a special file. By calling the function (lib-dump <file>) all objects are dumped. When the user starts a new session with the module library, a fast load of all object descriptions and variables, is facilitated. This is done by directly creating the object vectors, without the intermediate step of definition. The functions that provide this dump facility are included in the file libdump.

5.3 **system startup**

The system is started up by loading the file libload. This file takes care of loading all files mentioned, except the files libops and libmods. This is done because the user might want to define his own set of operations and modules, is likely to be the case in different design projects.
Finally, a menu is developed, through which a user can enter the module library. The menu takes care of the appropriate actions and warns the user for errors.

5.4 change of standard objects

Changes in the library system may be applied. As described previously, definition and update of object descriptions, based on standard objects, is provided for.

It is likely to occur that the standard operation or the standard module object has to be changed in the future, when intensive use of the module library points at some omissions. This is a major change in the library system and must be done carefully.

The first step is to change the variables that describe the standard objects. These are respectively *lib-operation-aspects* or *lib-module-aspects*. The next step is to adapt the *generic-module-template* accordingly. Furthermore the definition functions (lib-def-operation), (lib-def-module), (lib-fix-mux-ports) and (lib-fix-bits), in which aspects are treated separately, must be changed as far as necessary.

Changes in the former functions will have their impact on the screen interface functions. No rules can be given at this moment according to which adaptation of the screen functions must be applied. The functions in the screen interface must be checked on referring to other changed functions.

A consequence of larger extent is that the operation or module definitions that were made, have to be done again. It is important to know, that the call of the definition functions doesn't have to change. This holds, unless of course, aspects mentioned in the aspect-value-list of the function, are deleted from the standard object, or the new aspect must be included in the description. If this is not the case, the definition function provides a transparent feature for renewed definition of objects.
6. Conclusions

The module library is developed to support several tools in the silicon compiler. Not all tools are developed yet, but at this stage the demand graph constructor [Stok86], the cost calculator, [Enge87] and the hardware synthesis system [Woud87] make use of the module library. In the future, the module library will contain all the features and the data that is needed during the design, from the high level input description, up to the layout generation.

An object oriented approach of the module library is realised, which facilitates extendibility, and flexibility. An inheritance mechanism is implemented, in which modules at a high abstractional level are described. From these abstractions, more detailed module descriptions can be derived.

Fast retrieval times are obtained. In the appendix some numerical results are given of access functions that are developed for use in the several tools. The definition of operations and modules is provided for in a user friendly way. The internal datastructure is hidden from the user. The EMACS screen facility contains a small set of functions, for access and definition of objects. Definition of objects must be controlled more than is done at this moment. A better protection against inconsistencies in the module definition is needed. This set may well be extended in the future.

During the library development an imaginary set of modules was used. A realistic set of modules has to be defined, to be able to obtain realistic results with the silicon compiler.
References


[Hosk79] Descriptive databases in some design/manufacturing environment, E.M. Hoskins,
Proceedings of the

[Lewi86] Integrated circuit procedural language,
J.A. Lewis, A.A. Berlin, A.J. Kuchinsky, P.K. Yip,
HP-journal, June 1986, p. 4-10.

[Mcle85] Effective data management for VLSI design,
Paul McLellan,
Proceedings of the
22nd ACM/IEEE Design Automation Conference 1985,
p. 652-657.

[Nien79] Component library for an integrated DA system,
K.Y. Nieng, D.A. Beckley,
Proceedings of the
16th ACM/IEEE Design Automation Conference, 1979,
p. 437-444.

[Robe81] A vertically organised computer-aided design database,
K.A. Roberts, T.E. Baker, D.H. Jerome,
Proceedings of the
18th ACM/IEEE Design Automation Conference, 1981,
p. 595-602.

[Schu85] ICHABOD, A database manager,
H.B. Schutzman,
Proceedings of the
22nd ACM/IEEE Design Automation Conference, 1985,
p. 569-576.

[Sout83] MacPitts: An approach to silicon compilation,
J.R. Southard,
Computer (IEEE), vol 16, no 12,
p. 74-82, December 1983.

[Stok86] From IDEA to IC, the higher levels of a silicon compiler,
L. Stok

[Suck79] A design aids database for digital components,
D.J. Sucker, D.F. Wann,
Proceedings of the
16th ACM/IEEE Design Automation Conference 1979,
p. 414-420.

LISP, P.H. Winston, B.K.P. Horn, Addison-Wesley 1984.


Appendices

access functions

lib-find-module:
(lib-find-module <aspect-value-list>)
returns a list of modules of which the values of the aspects in the argument
match with the <aspect-value-list>.

lib-module-all:
(lib-module-all)
returns a list of provided modules. They are contained in the variable
*lib-modules*.

lib-module-area:
(lib-module-area <module>)
returns an integer number, giving the area of <module>.

lib-module-aspect:
(lib-module-aspect <aspect> <module>)
returns the value of <aspect> of <module>.

lib-module-boolean-function:
(lib-module-boolean-function <module>)
returns <boolean-function> of <module>.

lib-module-control:
(lib-module-control <module> <operation>)
returns the control-vector, found in the <control-table> of <module>, that
belongs to <operation>.

lib-module-control-list:
(lib-module-control-list <module>)
returns <control-list> of <module>.

lib-module-control-table:
(lib-module-control-table <module>)
returns <control-table> of <module>.

lib-module-delay:
(lib-module-delay <module> <operation>)
returns <delay> of <module>, that belongs to <operation>.
lib-module-delay-table:
(lib-module-delay-table <module>)
returns the table with <delay> information of <module>.

lib-module-designer:
(lib-module-designer <module>)
returns <designer> of <module>.

lib-module-dissipation:
(lib-module-dissipation <module>)
returns <dissipation> of <module>.

lib-module-documentation:
(lib-module-documentation <module>)
returns <documentation> of <module>.

lib-module-input:
(lib-module-input <module> <operation>)
returns <input> of <module>, that belongs to <operation>.

lib-module-input-list:
(lib-module-input-list <module>)
returns <input-list> of <module>.

lib-module-input-table:
(lib-module-input-table <module>)
returns the table with <input> information of <module>.

lib-module-layout:
(lib-module-layout <module>)
returns <layout> of <module>.

lib-module-length:
(lib-module-length <module>)
returns <length> of <module>.

lib-module-no-parameters:
(lib-module-no-parameters)
returns a list of modules that are not parameterized. They are contained in
the variable *lib-no-parameter-modules*.

lib-module-operations:
(lib-module-operations <module>)
returns a list of operations, performed by module.

lib-module-output:
(lib-module-output <module> <operation>)
returns <output> of <module>, that belongs to <operation>.

lib-module-output-list:
(lib-module-output-list <module>)
returns <output-list> of <module>.

lib-module-output-table:
(lib-module-output-table <module>)
returns the table with <output> information of <module>.

lib-module-parameters:
(lib-module-parameters <module>)
returns <parameters> of <module>.

lib-module-parents:
(lib-module-parents <module>)
returns <parents> of <module>.

lib-module-status:
(lib-module-status <module>)
returns <status> of <module>.

lib-module-supply-list:
(lib-module-supply-list <module>)
returns <supply-list> of <module>.

lib-module-technology:
(lib-module-technology <module>)
returns <technology> of <module>.

lib-module-width:
(lib-module-width <module>)
returns <width> of <module>.

lib-mux-modules:
(lib-mux-modules <ports> [ <bits> ])
when <bits> is nil, a list of multiplexers is returned that have <ports> input ports, and that have <bits> still as a parameter.
In the case that <bits> is an integer, a list of multiplexers is returned
that have `<ports>` input ports and `<bits>` as bitwidth of these ports.

**lib-operation-additive:**
(lib-operation-additive)
returns a list of operations that are dyadic, arithmetic, and additive.

**lib-operation-all:**
(lib-operation-all)
returns a list of all operations that are defined in the module library by use of the definition function (lib-def-operation `<arg>`).

**lib-operation-arithmetic:**
(lib-operation-arithmetic)
returns a list of arithmetic operations.

**lib-operation-arithmetic-p:**
(lib-operation-arithmetic-p `<operation>`)  
returns non-nil if `<operation>` is an arithmetic operation.

**lib-operation-associative:**
(lib-operation-associative)
returns a list of associative operations.

**lib-operation-associative-p:**
(lib-operation-associative-p `<operation>`)  
This function returns non-nil if `<operation>` is associative.

**lib-operation-associativity:**
(lib-operation-associativity `<operation>`)  
returns the value of the aspect `<associativity>` of `<operation>`.

**lib-operation-boolean:**
(lib-operation-boolean)
returns a list of boolean operations.

**lib-operation-boolean-p:**
(lib-operation-boolean-p `<operation>`)  
returns non-nil if `<operation>` is a boolean operation.

**lib-operation-class:**
(lib-operation-class `<operation>`)  
returns the value of the aspect `<class>`, a list with the class-elements of `<operation>`.
lib-operation-commutative:
(lib-operation-commutative)
returns a list of commutative operations.

lib-operation-commutative-p:
(lib-operation-commutative-p <operation>)
This function returns non-nil if <operation> is commutative.

lib-operation-commutativity:
(lib-operation-commutativity <operation>)
returns the value of the aspect <commutativity> of <operation>.

lib-operation-dyadic:
(lib-operation-dyadic)
returns a list of operations, that are dyadic.

lib-operation-dyadic-arithmetic:
(lib-operation-dyadic-arithmetic)
returns a list of operations, that are dyadic and arithmetic.

lib-operation-dyadic-boolean:
(lib-operation-dyadic-boolean)
returns a list of operations, that are dyadic and boolean.

lib-operation-dyadic-p:
(lib-operation-dyadic-p <operation>)
returns non-nil if <operation> is a dyadic operation.

lib-operation-io-transfer:
(lib-operation-io-transfer)
returns a list of io-transfer operations.

lib-operation-io-transfer-p:
(lib-operation-io-transfer-p <operation>)
This function returns non-nil if <operation> is of class io-transfer.

lib-operation-modules:
(lib-operation-modules <operations> [ <param> ])
returns a list of modules that perform the whole set of operations mentioned in the list <operations>. If <param> is nil the modules are taken from the subset of modules, that have no parameters. Otherwise the whole set of modules is searched in.
lib-operation-monadic:
(lib-operation-monadic)
returns a list of operations that are monadic.

lib-operation-monadic-arithmetic:
(lib-operation-monadic-arithmetic)
returns a list of operations that are monadic and arithmetic.

lib-operation-monadic-boolean:
(lib-operation-monadic-boolean)
returns a list of operations that are monadic and boolean.

lib-operation-monadic-p:
(lib-operation-monadic-p <operation>)
returns non-nil if <operation> is a monadic operation.

lib-operation-multiplicative:
(lib-operation-multiplicative)
returns a list of operations that are dyadic, arithmetic, and multiplicative.

lib-operation-non-io-transfer:
(lib-operation-non-io-transfer)
returns a list of non-io-transfer operations.

lib-operation-p:
(lib-operation-p <operation>)
returns non-nil if <operation> is an operation, defined by use of the
definition function (lib-def-operation <arg>).

lib-operation-relational:
(lib-operation-relational)
returns a list of relational operations.

lib-operation-relational-p:
(lib-operation-relational-p <operation>)
returns non-nil if <operation> is an relational operation.
The definition functions have an aspect-value list as argument. The syntax according to which these lists have to be given is described below.

Operation definition
Operations are defined by the function (lib-def-operation <operation-name> <aspect-value-list>). The syntax of <aspect-value-list> is described below.

aspect-value-list:

aspect-value-pair:

class:

class-element:

commutativity:

associativity:
module definition syntax

Modules are defined by the function
(lib-def-module <module-name> <aspect-value-list>). The syntax of <aspect-value-list> is described below.

aspect-value-list :

parent :

aspect-value-pair :

- documentation
- parameters
- length
- width
- dissipation
- input-list
- output-list
- control-list
- input-table
- output-table
- control-table
- delay-table
- supply-list
- designer
- status
- technology
- layout
documentation:

parameters:

length:

width:

dissipation:

input-list:

output-list:

control-list:

input-table:

output-table:

control-table:

delay-table:
supply-list:
- symbol

status:
- preliminary
- delivered

layout:
- symbol

designer:
- string

technology:
- string

parameter:
- symbol

interface-list:
- interface

interface:
- symbol
- bitwidth

bitwidth:
- regular-expression

operation-delay:
- operation
- regular-expression

operation-control:
- operation
- boolean-vector

operation-inputs:
- operation
- inputs

operation-outputs:
- operation
- outputs
boolean-vector:

inputs:

outputs:
timing analysis of access functions

In this chapter a series of timing analysis is made of the access functions as they are provided by several versions of the module library.

module library based on vectors

The seconds preceding the function call is the time needed for executing the function 1000 times.

time (s)   time

<table>
<thead>
<tr>
<th>time</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.3326</td>
<td>(lib-module-width 'lib-add1))</td>
</tr>
<tr>
<td>16.98265</td>
<td>(lib-module-technology 'lib-add1))</td>
</tr>
<tr>
<td>22.38244</td>
<td>(lib-module-supply-list 'lib-add1))</td>
</tr>
<tr>
<td>16.29935</td>
<td>(lib-module-status 'lib-add1))</td>
</tr>
<tr>
<td>17.46597</td>
<td>(lib-module-parents 'lib-add1))</td>
</tr>
<tr>
<td>10.39958</td>
<td>(lib-module-parameters 'lib-add1))</td>
</tr>
<tr>
<td>21.33248</td>
<td>(lib-module-output-table 'lib-add1))</td>
</tr>
<tr>
<td>13.26614</td>
<td>(lib-module-output-list 'lib-add1))</td>
</tr>
<tr>
<td>33.81532</td>
<td>(lib-module-output 'lib-add1 ')</td>
</tr>
<tr>
<td>10.96623</td>
<td>(lib-module-operations 'lib-add1))</td>
</tr>
<tr>
<td>3.133208</td>
<td>(lib-module-no-parameters)</td>
</tr>
<tr>
<td>18.34927</td>
<td>(lib-module-length 'lib-add1))</td>
</tr>
<tr>
<td>17.08265</td>
<td>(lib-module-layout 'lib-add1))</td>
</tr>
<tr>
<td>15.06606</td>
<td>(lib-module-input-table 'lib-add1))</td>
</tr>
<tr>
<td>21.46581</td>
<td>(lib-module-input-list 'lib-add1))</td>
</tr>
<tr>
<td>27.68223</td>
<td>(lib-module-input 'lib-add1 '+))</td>
</tr>
<tr>
<td>17.98261</td>
<td>(lib-module-documentation 'lib-add1))</td>
</tr>
<tr>
<td>13.73278</td>
<td>(lib-module-dissipation 'lib-add1))</td>
</tr>
<tr>
<td>23.83238</td>
<td>(lib-module-designer 'lib-add1))</td>
</tr>
<tr>
<td>12.83282</td>
<td>(lib-module-delay-table 'lib-add1))</td>
</tr>
<tr>
<td>23.99904</td>
<td>(lib-module-delay 'lib-add1 '+))</td>
</tr>
<tr>
<td>21.5658</td>
<td>(lib-module-control-table 'lib-add1))</td>
</tr>
<tr>
<td>13.96611</td>
<td>(lib-module-control-list 'lib-add1))</td>
</tr>
<tr>
<td>32.89869</td>
<td>(lib-module-control 'lib-add1 '+))</td>
</tr>
<tr>
<td>15.94936</td>
<td>(lib-module-boolean-function 'lib-add1))</td>
</tr>
<tr>
<td>115.812</td>
<td>(lib-module-area 'lib-add1))</td>
</tr>
<tr>
<td>3.066544</td>
<td>(lib-module-all))</td>
</tr>
<tr>
<td>18.86591</td>
<td>(lib-module-documentation 'lib-adder1))</td>
</tr>
<tr>
<td>19.2159</td>
<td>(lib-module-operations 'lib-adder1))</td>
</tr>
<tr>
<td>10.91623</td>
<td>(lib-module-parameters 'lib-adder1))</td>
</tr>
</tbody>
</table>
[20.13253 (lib-module-parents 'lib-adder1))
12.73282 (lib-module-length 'lib-adder1))
20.69917 (lib-module-width 'lib-adder1))
41.715 (lib-module-area 'lib-adder1));aref
122.3118 (lib-module-area 'lib-adder1));calculated
13.39946 (lib-module-dissipation 'lib-adder1))
22.11578 (lib-module-delay-table 'lib-adder1))
22.4491 (lib-module-input-list 'lib-adder1))
13.89944 (lib-module-output-list 'lib-adder1))
15.36605 (lib-module-input-table 'lib-adder1))
22.24911 (lib-module-output-table 'lib-adder1))
23.71572 (lib-module-control-table 'lib-adder1))
24.58235 (lib-module-supply-list 'lib-adder1))
22.84909 (lib-module-boolean-function 'lib-adder1))
17.18265 (lib-module-designer 'lib-adder1))
17.53263 (lib-module-status 'lib-adder1))
17.91595 (lib-module-technology 'lib-adder1))
28.68219 (lib-module-layout 'lib-adder1))
26.99892 (lib-module-output 'lib-adder1 'x+'))
3.383198 (lib-module-no-parameters))
28.64885 (lib-module-input 'lib-adder1 'x+'))
27.78222 (lib-module-delay 'lib-adder1 'x+'))
27.46557 (lib-module-control 'lib-adder1 'x+'))
3.049878 (lib-module-all))
6.58307 (lib-operation-associative-p '+))
14.49942 (lib-operation-commutative-p '+))
8.316334 (lib-operation-p '+))
14.96607 (lib-operation-monadic-p '+))
6.716398 (lib-operation-io-transfer-p '+))
14.68275 (lib-operation-dyadic-p '+))
7.116382 (lib-operation-boolean-p '+))
13.49946 (lib-operation-associative-p '+))
6.849726 (lib-operation-arithmetic-p '+))
13.49946 (lib-operation-relational-p '+))
136.9112 (lib-operation-modules '(+) t))
38.2318 (lib-operation-commutativity '+))
47.04812 (lib-operation-associativity '+))
40.89837 (lib-operation-class '+))
3.283202 (lib-operation-relational))
3.364532 (lib-operation-monadic-boolean))
3.349866 (lib-operation-monadic-arithmetic))
3.299868 (lib-operation-dyadic-boolean))
3.049878  (lib-operation-dyadic-arithmetic))
3.066544  (lib-operation-associative))
3.049878  (lib-operation-commutative))
3.049878  (lib-operation-boolean))
3.049878  (lib-operation-associative))
3.066544  (lib-operation-arithmetic))
3.066544  (lib-operation-all))
151.5939  (lib-operation-modules '(or t))
7.33304  (lib-operation-associative-p 'or))
14.38276  (lib-operation-commutative-p 'or))
8.333  (lib-operation-p 'or))
15.94936  (lib-operation-monadic-p 'or))
7.16638  (lib-operation-io-transfer-p 'or))
14.23276  (lib-operation-dyadic-p 'or))
7.016386  (lib-operation-boolean-p 'or))
14.2161  (lib-operation-associative-p 'or))
7.099716  (lib-operation-arithmetic-p 'or))
14.49942  (lib-operation-relational-p 'or))
38.88178  (lib-operation-commutativity 'or))
39.34843  (lib-operation-associativity 'or))
46.96479  (lib-operation-class 'or))

module library based on property lists

time (s) function

18.13261  (lib-operation-relational-p 'or))
4.633148  (lib-operation-multiplicative-p 'or))
11.33288  (lib-operation-p 'or))
14.08277  (lib-operation-monadic-p 'or))
18.13261  (lib-operation-logic-p 'or))
4.316494  (lib-operation-io-transfer-p 'or))
18.06594  (lib-operation-dyadic-p 'or))
7.433036  (lib-operation-commutative-p 'or))
18.13261  (lib-operation-arithmetic-p 'or))
4.616482  (lib-operation-additive-p 'or))
235.1573  (lib-operation-modules '(or)))
11.33288  (lib-op-class 'or))
8.399664  (lib-module-width 'alu-bit))
11.36621  (lib-module-type 'alu-bit))
12.48283  (lib-module-technology 'alu-bit))
11.5662  (lib-module-supply-list 'alu-bit))
13.74945 (lib-module-date-of-last-update 'add1))
17.93262 (lib-module-control-vector 'add1) '\=*))
12.5495  (lib-module-control-table 'add1))
12.73282 (lib-module-control-list 'add1))
12.24951 (lib-module-boolean-function 'add1))
24.13237 (lib-module-area 'add1))
293.9716 (lib-operation-relational)
291.2884 (lib-operation-multiplicative)
296.4715 (lib-operation-monadic-logic)
304.0712 (lib-operation-monadic-arithmetic)
298.1881 (lib-operation-logic)
311.0042 (lib-operation-dyadic-logic)
300.5713 (lib-operation-dyadic-arithmetic)
292.8549 (lib-operation-arithmetic)
249.59  (lib-operation-all)
289.6384 (lib-operation-additive)