OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming

Birger Landwehr, Peter Marwedel,
Rainer Dömer

Lehrstuhl Informatik XII
University of Dortmund

Report No. 484

April 1994

Abstract: In this report we describe an IP-model based high-level synthesis system. In contrast to other approaches, the presented IP-model allows solving all three subtasks of high-level synthesis (scheduling, allocation and binding) simultaneously. As a result, designs which are optimal with respect to the cost function are generated. The model is able to exploit large component libraries with multi-functional units and complex components such as multiplier-accumulators. Additionally, the model is capable of handling mixed speeds and chaining in its general form. Applying algebraic transformations helps to exploit underlying component libraries more efficiently than other HLS-systems[1].

[1]This work has been partially supported by ESPRIT project BRA 6855 (LINK).

Contents

Chapter 1

Introduction

During the recent years, there has been an ever-increasing demand to speed up the design cycles for the design of electronic systems. This demand is caused by time-to-market requirements for products in this area. At the same time, there has been an increasing need to achieve for correctness by construction.

Due to these driving forces, synthesis techniques are now being used for the design of many electronic products. Currently, logic synthesis and synthesis of finite state machines are the most widely employed techniques. Unfortunately, these techniques cannot be used for the design of efficient systems containing data paths. The design of such systems is the goal of the various approaches to high-level-synthesis. Despite significant efforts by researchers in the area, high-level synthesis is still hardly used in industry. A major reason for this is the inefficiency of current high-level synthesis systems. This inefficiency has several reasons. We believe, the two most important reasons are

1. the partitioning of algorithms for high-level synthesis into algorithms for solving the three subtasks (scheduling, allocation and binding) independently. However, these subtasks are related and therefore this approach potentially results in inferior designs.

2. library mapping – in contrast to its popularity for logic synthesis – is still very poor.

In this report, we will describe a method which potentially improves the efficiency of high-level synthesis significantly by avoiding these two sources of inefficiencies. The synthesis system based on our method is called OSCAR (optimum simultaneous scheduling, allocation and resource binding).

In the following chapter, we classify our work with respect to previous publications. A brief overview of the entire high-level system follows in chapter 3. The underlying IP-model as the basis of our synthesis method will be described in chapter 4. Thereafter, in chapter 5 we present a method for intelligent library component in combination with an efficient library management. Experimental results can be found in chapter 7. Chapter 8 concludes this report with a summary and future research topics.

Chapter 2

Related work

Almost all early approaches to high-level synthesis partitioned the problem into subproblems for scheduling, allocation and binding. This includes the work by Tseng and Siewiorek [ST83], Marwedel [Mar86] and others (see [MPC90] for a survey). This work helped finding solution methods for these subproblems. Later it was recognized, that these subproblems should be solved simultaneously in order to avoid suboptimal results.

The work of Gebotys is especially stimulating in this respect, because it is based on a formal integer programming (IP)-model, which has the potential of solving several subproblems concurrently. The approach to scheduling in this work is an improvement over an earlier model by Hwang [HLH91].

Early work by Gebotys into this direction [GE91] did not include binding. Recently, the two-index model in [GE91] has been extended into a three index model (see e.g. [GE93]). The three-index model has the potential of handling advanced features such as pipelining, mixed speeds and wiring optimization. Unfortunately, the latter is not described in [GE93].

Other approaches which are based on IP-models include the following:

Library mapping (see also Dutt [Dut88]) is emphasized in a paper by Achatz [Ach93], integrating scheduling and allocation. If only scheduling and allocation have to be integrated and if additional constraints are met, the 0/1 integer program used by Gebotys can be replaced by a more efficient general integer problem [WMGB93].

Allocation and assignment have been integrated for example by Rim and Jain [RJL92].

In addition to having the potential for solving subproblems concurrently, IP-models have the advantage of being formal models of high-level synthesis. This makes formally checking the correctness of high-level synthesis easier.

Now that a considerable amount of heuristics have been published, we believe that it is the time to investigate more formal models.

Currently available IP-models, however, do still have major limitations. For example, currently published IP-models do not allow chaining in its general form. For all existing algorithms, operations to be chained must be manually replaced by a single, more powerful operation before actual synthesis is started. This means, these algorithms cannot automatically decide whether or not two operations are to be chained.

Moreover, existing algorithms usually consider simple libraries containing mostly adders and multipliers. They are not capable of exploiting efficient complex components such as multiplier-accumulators and many of them are unable of selecting components with different speeds.

Chapter 3

Design flow in the OSCAR system

The OSCAR system (see fig. 3.1) essentially consists of two parts: the actual high-level synthesis kernel which is based upon an IP-model (left part) as well as the library component selection and management (right part).

The input specification of our target design consists of two separate descriptions: the behavioral description based on a VHDL subset and the design specification. The latter especially contains user defined timings and manual bindings of operations to control steps and/or component instances.

After reading the two input descriptions, the internal control- and data-flow graph will be created. Our data structure is based upon a version of the *Assignment Decision Diagram* [VC92][HCG92]) which is extended in order to handle additional timing information *(TADD)*.

The next major task consists in selecting suitable library components and to find an optimal coverage (in the sense of costs) concerning the data-paths. We have divided this task into 5 basic steps:

1. Extracting primary operators (by the *DFG Analyzer*): all arithmetical and relational operators in the TADD are extracted and saved in the *operation list*. Constants as arguments are saved together with the related operators as well. This simplifies the recognition of expressions which can be simply implemented by interconnect (e.g. consider that expression `x * 2` can be simply implemented by an offset in the wiring pattern).

2. Extracting secondary (derived) operators (by the *DFG Analyzer*): Each operation is checked whether it can be transformed to a simpler one. For example, a multiplication by 2 can be transformed to either (`a + a`) or (`a shl 1`). Obviously, both derived operations are simpler and cheaper (in their realization) than the original multiplication. Such applicable transformations are stored in the external *algebraic rule library*. For this example, the operation list contains now the *add* and *shift* operation together with the previously extracted multiplication.

3. Extracting algebraic expressions (by the *DFG Analyzer*): In the next step, we will try to find expressions in the TADD which can be implemented by single components. We call such expres-

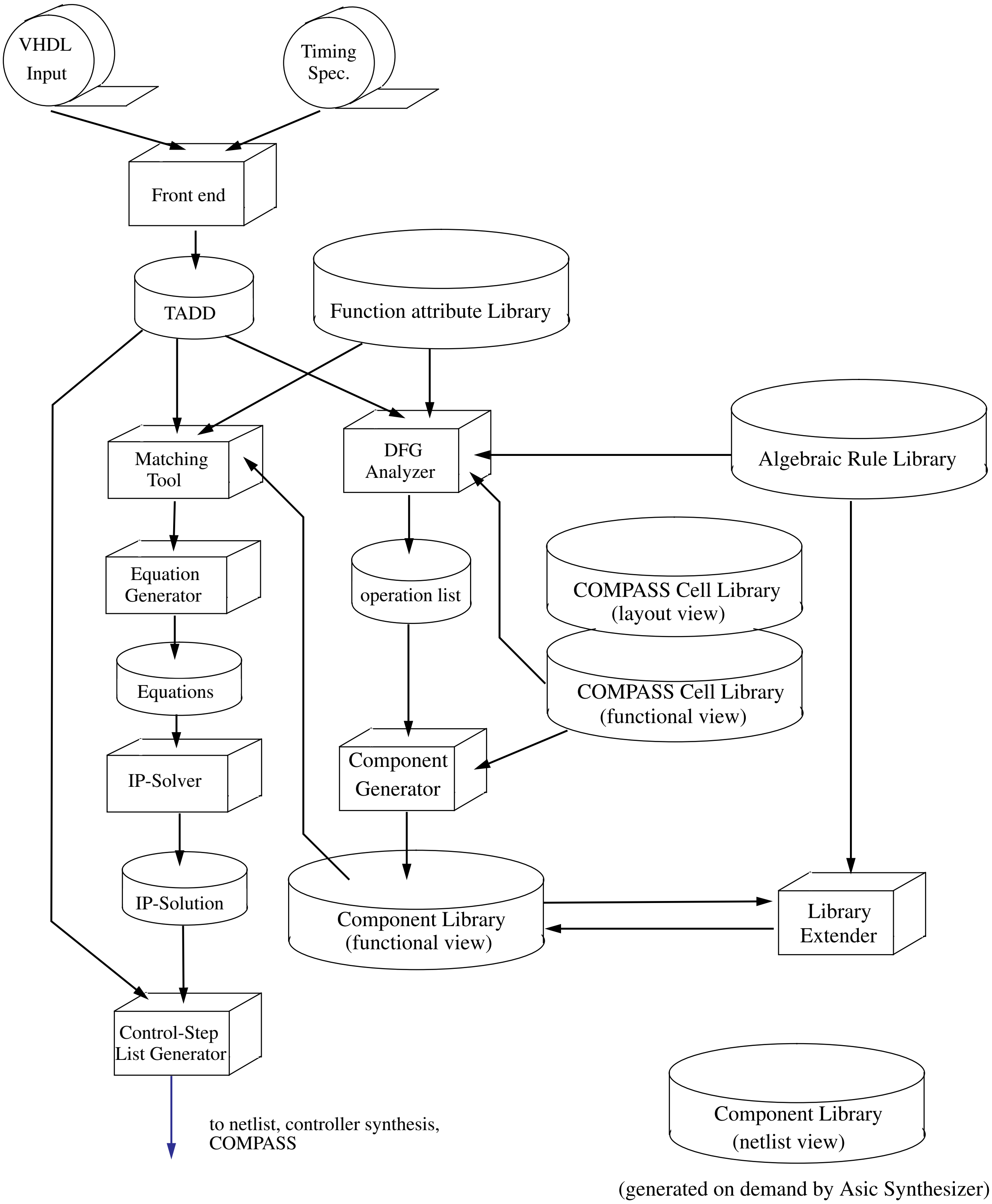


Figure 3.1: System overview

sions *macro-operations*. Such macro-operations will be also saved in the operation list.

4. Component generation: Cell libraries usually consist of data-path cells and simple standard cells like gates etc. Data-path cells are often realized by cell compilers which generate components of the required bit width on demand. We have divided this compilation process into two phases:

    (a) Phase I (by the *Component Generator*): In the first phase only the functional view of n-bit components is generated. This view is sufficient for component selection since only the actual behaviour is required at this time. The generation of the relating component netlists is task of phase II.

    (b) Phase II (by the underlying CAD tool *Compass*): The component netlists are only generated on demand in the case of a component instantiation. Consequently, we can move this phase to the end of the entire synthesis process.

5. Component library extension (by the *Library Extender*): The component library contains all those components which are capable to execute at least one operation of the operation list. Since components were specially generated for a certain operation, the utilization for other operations is generally poor. The utilization can be optimized by applying algebraic rules to the component's functional description.

In the next step an optimal matching between components and the TADD is determined. The *Matching Tool* can be considered as connecting link between the component selection part and the actual HLS part. It generates a list of possible and useful matches between available components and (macro-) operations.

Now, all required equations can be generated by the *equation generator* and passed to the IP-solver [Ber92]. The synthesis result – a list of integer variables – is transformed to the so-called control-step-list which is used as input for netlist generation and for the controller description. Then, the controller is synthesized and mapped to the target technology by the underlying CAD tool. Finally, all netlists of required components are generated (phase II of component generation).

Chapter 4

IP-model based high-level synthesis

4.1 Integrated Scheduling, Allocation and Binding

4.1.1 Binding model

High-level synthesis basically has to establish bindings between operations j, control steps i and resources k. Such bindings can be represented by binary decision variables. Triple-indexed variables are required for integrating scheduling, allocation and binding.

$$x_{i,j,k} = \begin{cases} 1, & \text{if operation } j \text{ will be started on resource instance } k \text{ at control step } i \\ 0, & \text{otherwise} \end{cases}$$

Throughout this text, we will use index k as a name for a particular component. K will be the index set (value range) of these names.

Each component k will be an instance of a component type contained in the component library. We will use index m to denote a certain component type and M to denote the set of names of all component types. Function *type* is assumed to return the component type of a certain component instance.

Furthermore, we will use index j to uniquely denote an operation contained in the TADD. J will be the set of all j's. More precisely, each j corresponds to an operation *instance*. Each j is of a certain type, e.g. a particular operation in the TADD may be an instance of operation type `add` or `multiply`. We will use g to denote a certain operation type and G to denote the set of all operation types. Function *optype*(j) is assumed to return the operation type of a certain operation instance j.

Table 4.1 contains the used mathematical symbols.

Variables $x_{i,j,k}$ have to be computed by the synthesis system. This will assign a control step i and a component instance k to each operation instance j.

All combinations of i, j and k for which no solution is feasible will not be used as subscripts of x. For

$I \subset I\!N_0$	the set of control steps
$i \in I$	control step $i \in I$
$J \subset I\!N_0$	the set of operations in the TADD
$j \in J$	operation $j \in J$
$K \subset I\!N_0$	index set of resource instances
$k \in K$	resource instance $k \in K$
$M \subset I\!N_0$	index set of existing resource types
$m \in M$	resource type $m \in M$
$R(j)$	range of possible control steps for operation j
G_m	operation types that can be executed by m
k_{max}	maximum number of all available instances
$\ell(j,k)$	latency of component k for operation j
$\underline{type}(k)$	component type of component instance k
$G \subset I\!N_0$	the set of operation types
$g \in G$	operation type $g \in G$
$\underline{optype}(j)$	operation type of operation instance j
$C(j,k)$	delay for executing j on k
$C(j) = \max_k C(j,k)$	maximum of delays for executing j on k

Table 4.1: Mathematical notation

example, if k cannot perform operation j, the corresponding decision variables are never generated in order to reduce the number of variables and relations.

4.2 Constraints

4.2.1 Operation assignment constraints

Each component type m is assumed to be able to execute a set G_m of operation types. E.g. a certain component type may be able to perform additions and multiplications while others are only able to perform either of the two.

Our first set of constraints now models the fact that each operation j should be started on exactly one resource instance of the appropriate type. Furthermore, each operation j should be started in a control step i which lies within the range $R(j)$ of feasible control steps. $R(j)$ is the range of control steps between the earliest (ASAP) and latest (ALAP) control step feasible for operation j. These conditions are modelled by the following relations:

$$\forall j \in J : \quad \sum_{i \in R(j)} \sum_{\substack{k \in K \\ j \text{ executable on } k}} x_{i,j,k} = 1 \tag{4.1}$$

For each j, the sum over k includes only instances for which the relation "j executable on k" holds. This ensures that operations will be mapped to appropriate components. Relation "j executable on k" can be defined as:

$$j \text{ executable on } k \iff \underline{optype}(j) \in G_{\underline{type}(k)} \tag{4.2}$$

Note that relation "j executable on k" is more general than the corresponding implicit relation in [GE91]. For each k, we have to know the corresponding type m before solving our synthesis problem. To model this knowledge, we are using a function called $\underline{type}$. For each potential instance k, function $\underline{type}$ has to return the corresponding resource type. Without loss of generality, we require $\underline{type}$ to be a monotone step function of k. Before synthesis, a sufficiently large number of potential instances is automatically computed for each type m.

In order to model the fact that for a certain potential instance k, the instance may be either selected or left out of the final design, we introduce decision variables b_k:

$$b_k = \begin{cases} 1, & \text{if instance } k \text{ is selected} \\ 0, & \text{otherwise} \end{cases}$$

A simple observation can be used to speed up the search for optimal designs: if $type(k) = type(k+1)$, then the solutions $b_k = 1, b_{k+1} = 0$ and $b_k = 0, b_{k+1} = 1$ are equivalent, except for renaming of resource instances. In order to generate only one of these equivalent solutions, we require that

$$\forall k : \quad \text{if } type(k) = type(k+1) \text{ then } b_k \leq b_{k+1} \tag{4.3}$$

without loss of generality.

Experimental results have shown that this constraint can reduce the execution time of the IP-solver by a factor of 10 - 50 even for small examples!

This redundancy is not eliminated in other models [Gebotys93].

Components which we have considered so far are able to perform certain functions. Results computed by these functions can be described in terms of expressions involving operators, input ports and constants. Up till now, high-level synthesis systems have only considered expressions involving a single operator, e.g `in_a + in_b`. In the following, we will call components computing those expressions *simple components.*

Recent component libraries, however, do contain components such as multiplier-accumulators (MACs), which compute expressions like `(in_a * in_b) + in_c`. These components correspond to complex gates in logic synthesis and we will therefore call these components *complex components.* Complex components allow very efficient implementations, but high-level synthesis systems in general are not capable of exploiting them.

One of the goals we set for OSCAR is to decide automatically whether to map sets of adjacent operations to several simple components or to a single complex component.

To this end, we define *macro-operations*[1] to be a set of adjacent operations which can be executed by at least one component type. Let Y be the set of all such macro-operations.

Constraint (4.4) ensures that either all operations contained in a macro-operation $y \in Y$ are assigned to separate simple components or to a single complex component.

$$\forall j \in J : \sum_{i \in R(j)} \sum_{\substack{k \in K \\ j \text{ executable on } k}} x_{i,j,k} \quad + \sum_{\substack{y \in Y: \\ j \in y}} \sum_{i \in R(y)} \sum_{\substack{k \in K \\ y \text{ executable on } k}} x_{i,y,k} = 1 \tag{4.4}$$

If no complex components are employed, the right sum of (4.4) becomes 0. In this case the constraint reduces to the standard operation assignment constraint (4.1).

Additionally, constraint (4.5) restricts the assignment of macro-operations to at most one complex component:

$$\forall y \in Y : \sum_{i \in R(y)} \sum_{\substack{k \in K \\ y \text{ executable on } k}} x_{i,y,k} \quad \leq \quad 1 \tag{4.5}$$

Before calculating the IP-model, a matching between the system and component behavior must be performed in a preprocessing phase. Figure 4.1 illustrates the matching between an DFG (elliptical wave filter [KWK85]) and a MAC represented by an expression tree.

The MAC can also be employed to perform only one of the two operations by applying neutrals to the input lines according to the selected operation. Due to this, a component library which consists of only this component is sufficient for synthesizing this benchmark.

The described technique of compiling operations to macro-operations in order to exploit complex components can be also applied for module sharing: a set of data-independent operations are allowed to be assigned to the same component instance at the same control step if the following presumptions are fulfilled:

[1] Note that macro-operations $y \in Y$ can be handled in the following constraints just as conventional operations $j \in J$

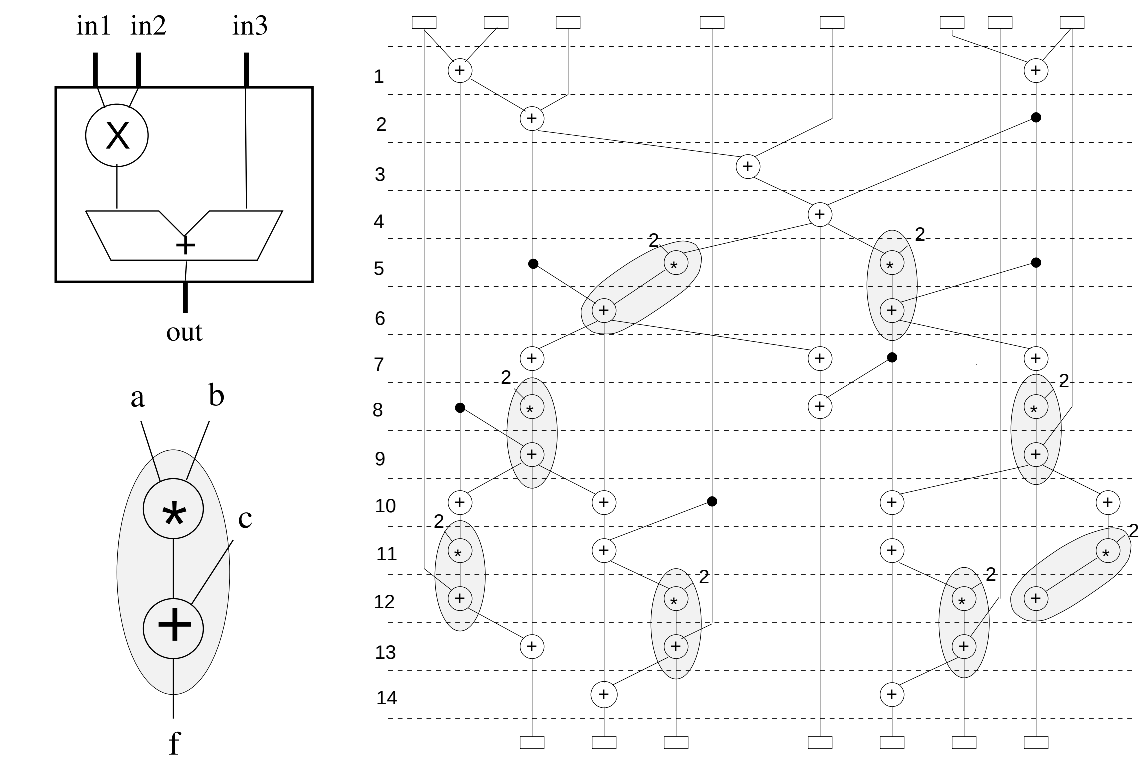


Figure 4.1: Matching module and system behavior (elliptical wave filter)

- the number of port lines must be at least as large as the sum of argument bitwidths
- a sufficient number of separation bits between the arguments must be inserted to avoid interactions between the operations
- all unused input lines of the module must be *don't care*-extendable[2]

These requirements are fulfilled by several function units like the n-bit adder shown in figure 4.2.

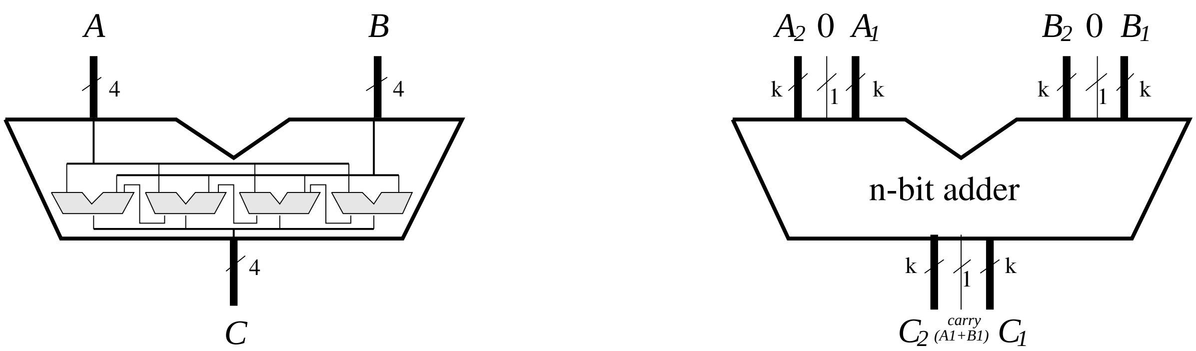


Figure 4.2: n-bit adder

Such components can be employed to perform two data-independent additions which are defined on

[2]the required information is specified in the *function attribute library*, see chapter 5.1, 5.3.

k bits (with $k + k + 1 \leq n$). One separation bit $s = 0$ has to be inserted between both argument pairs in order to avoid carry propagation.

While generating the netlist, the resulting output $C_1 x C_2$ (x represents the carry of $A_1 + B_1$) must be split into two bit vectors C_1 and C_2.

Another extension of the operation assigment constraint deals with alternative DFG versions. We obtain such alternative DFGs by applying algebraic transformations to an original DFG. The original DFG and all derived DFGs are equivalent and must be considered as mutual exclusive during synthesis (see chapter 5.3.2).

We define D as set of all original DFGs d in the TADD. The set $V(d)$ contains all alternative versions v of one DFG $d \in D$. The decision variable $u_{d,v}$ describes whether or not a certain version v of DFG d is selected in the final design:

$$u_{d,v} = \begin{cases} 1, & \text{if version } v \text{ of DFG } d \text{ is selected} \\ 0, & \text{otherwise} \end{cases}$$

For each DFG d, the mutual exclusiveness of alternative versions is formulated by constraint (4.6).

$$\forall d \in D, \forall v \in V(d), \forall j \in J : \sum_{i \in R(j)} \sum_{\substack{k \in K \\ j \text{ executable on } k}} x_{i,j,k} + \sum_{\substack{y \in Y: \\ j \in y}} \sum_{i \in R(y)} \sum_{\substack{k \in K \\ y \text{ executable on } k}} x_{i,y,k} = u_{d,v} \qquad (4.6)$$

Constraint (4.7) models the fact that exactly one version v of a certain DFG p must be selected.

$$\forall d \in D : \sum_{v \in V(d)} u_{d,v} = 1 \qquad (4.7)$$

4.2.2 Resource assignment constraints

We assume that all components are only able to start a limited number of operations. More precisely, we assume that component k is able to start a new operation j every $\ell(j,k)$ control steps. $\ell(j,k)$ is called the component *latency*. This restriction is modelled by the following relations:

$$\forall i \in I : \forall k \in K : \sum_{\substack{j \in J \\ j \text{ executable on } k}} \sum_{\substack{i'=i \\ i \in R(j)}}^{i+\ell(j,k)-1} x_{i',j,k} \leq b_k \qquad (4.8)$$

A naive approach would use 1 as the constant at the right hand side of this equation. With the current approach we avoid solutions in which operations are assigned to non-selected instances.

Summing up into forward direction (from $i' = i$ to $i + \ell(j,k) - 1$) has the advantage of replacing the two constraint sets (2) and (13) in [GE93] by a single constraint set.

4.2.3 Precedence constraints

Data dependency relations are explicitly represented in the TADDs. For data-dependent operations, the following constraints have to be met:

$$\forall j_1 \prec j_2 : \quad \forall i \in R(j_2) \cap (R(j_1) + C(j_1) - 1) :$$

$$\sum_{\substack{k \\ j_2 \text{ executable on } k}} \sum_{\substack{i_2 \le i - chain(j_1,j_2) \\ i_2 \in R(j_2)}} x_{i_2,j_2,k} + \sum_{\substack{k \\ j_1 \text{ executable on } k}} \sum_{\substack{i-(C(j_1,k)-1) \le i_1 \\ i_1 \in R(j_1)}} x_{i_1,j_1,k} \le 1 \tag{4.9}$$

$C(j_1, k)$ denotes the delay of operation j_1 on component instance k. This notation allows different execution times of a certain operation on different function units. It can be shown that this approach is sufficient to guarantee correct solutions even in the case of components with mixed speeds.

Parameter $chain(j_1, j_2)$ describes a possible assignment of both operations j_1, j_2 to the same control step presuming that suitable components are available. This parameter should be calculated for all operation pairs in a preprocessing step.

If *chain* is set to 0, data-dependent operations will be assigned to different control steps. In this case, the support of chaining is limited to manually created combined operations. This is the approach taken in [GE91].

If *chain* is set to 1, data-dependent operations are allowed to be assigned to the same control step. This case corresponds to chaining in the more traditional sense.

4.2.4 Chaining Constraints

In case that two or more data-dependent operations are assigned to the same control step the sum of real execution times must be less than the user defined cycle time $time_{cycle}$.

We define a new relation

$$\begin{array}{rcrcl} j_1 \prec\!\!\prec j_2 & \Longleftrightarrow & & & j_1 \prec j_2 \quad \wedge \\ & & \exists k_1 & : & j_1 \text{ executable on } k_1, \\ & & \exists k_2 \neq k_1 & : & j_2 \text{ executable on } k_2 : \\ & & & & time(j_1,k_1) + time(j_2,k_2) + \ell_{phy} \le time_{cycle} \end{array}$$

Constant ℓ_{phy} describes a system dependent latency caused by interconnect delays. Further, we define a new set *CHAINS*. Each $ch \in CHAINS$ is the longest chain $j_1 \prec \cdots \prec j_n$ ($1 \ldots n$ are local indices) in the data flow graph with:

1. operations j_1 to j_n are data dependent: $\forall j_i,\ i \in \{1, \ldots, n-1\} : j_i \prec j_{i+1}$
2. at least two operations $j_i, j_{i+1} \in ch$ can be executed in the same control step: $j_i \prec\!\!\prec\ j_{i+1}$
3. ch is maximal: $\nexists\, j_0 : j_0 \prec\!\!\prec\ j_1 \quad \wedge \quad \nexists\, j_{n+1} : j_n \prec\!\!\prec\ j_{n+1}$

$$
\begin{aligned}
CHAINS &:= \bigcup ch \\
ch &:= \{j_1, \ldots, j_n \mid j_i \in J \ \wedge\ \forall j_i,\ i \in \{1, \ldots, n-1\} : j_i \prec\!\!\prec\ j_{i+1}\}
\end{aligned}
$$

Constraint (4.10) restricts the maximum number of chained operations $j \in ch$ per control step. In combination with constraint (4.9) only coherent operator chains are allowed to be assigned to the same control step. The maximum length of each chain is restricted by $time_{cycle} - \ell_{phy}$

$$
\forall ch \in CHAINS:\ \forall i \in I : \quad \sum_{\substack{j \in ch: \\ i \in R(j)}} \ \sum_{\substack{k \in K \\ j \text{ executable on } k}} time(j,k) * x_{i,j,k} \quad \leq \quad time_{cycle} - \ell_{phy} \qquad (4.10)
$$

Figure 4.3 illustrates the effect of chaining for simple expression tree. A cycle time of $100ns$ and latency of $10ns$ restrict the total execution time of all chained operations to at most $90ns$. The left side of the figure represents two possible solutions with maximum number of chained operations in one control step.

In this example, ch_1 consists of $\{j_1, j_3, j_4\}$ because of $j_1 \prec j_3 \prec j_4$ and $j_1 \prec\!\!\prec\ j_3, j_3 \prec\!\!\prec\ j_4$. The other set ch_2 is represented by $\{j_2, j_3, j_4\}$. Set *CHAINS* consists of $\{ch_1, ch_2\}$.

In case of solution I, constraint (4.10) is fulfilled only for the first two elements of ch_1 and ch_2. The alternative solution shown underneath consists of the entire set ch_1. The additional assignment of operation $j_2 \in ch_2$ in the same control step would violate constraint (4.10).

4.2.5 Timing constraints

Models for timing constraints can be taken over from [GE91]. For example, if two operations j_1 and j_2 should be separated by T control steps, then the following relations should hold:

$$
\forall i : \qquad \sum_{\substack{k \\ j_1 \text{ executable on } k}} x_{i_1,j_1,k} + \sum_{\substack{k \\ j_2 \text{ executable on } k}} \ \sum_{\substack{i_2 \neq i+T \\ i_2 \in R(j_2)}} x_{i_2,j_2,k} \leq 1, \qquad i_1 \in R(j_1) \qquad (4.11)
$$

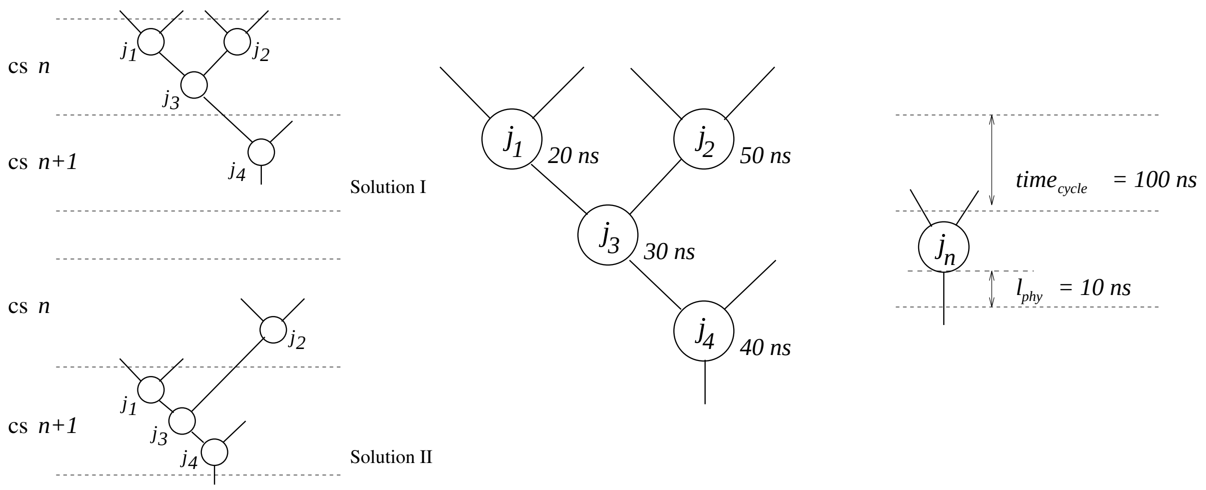


Figure 4.3: Operation chaining

Minimum timing constraints[3] can be represented by the following relation, respectively:

$$\forall i: \quad \sum_{\substack{k \\ j_1 \text{ executable on } k}} \sum_{\substack{i_1 \geq i \\ i_1 \in R(j_1)}} x_{i_1,j_1,k} + \sum_{\substack{k \\ j_2 \text{ executable on } k}} \sum_{\substack{i_2 \leq i+(C(j_1,k)-1)+T \\ i_2 \in R(j_2)}} x_{i_2,j_2,k} \leq 1 \qquad (4.12)$$

4.3 Cost function

One of the objectives of the current paper is to show how interconnect optimization can be integrated into a unified model for scheduling, allocation and binding. Therefore, the cost function has to include terms describing interconnect. The following cost function fulfills this requirement:

$$\sum_{m \in M} (c_m * \sum_{\substack{k \in K \\ \underline{type}(k)=m}} b_k) + \sum_{k_1,k_2} c_{k_1,k_2} * w_{k_1,k_2} \qquad (4.13)$$

with

$$w_{k_1,k_2} = \begin{cases} 1, & \text{if component } k_2 \text{ is connected to component } k_1 \\ 0, & \text{otherwise} \end{cases} \qquad (4.14)$$

The first term describes the cost of functional units. c_m is the cost per instance of component type m.

The second term describes the cost of the interconnect. c_{k_1,k_2} is the cost for interconnecting k_1 to k_2. Due to the lack of layout information, c_{k_1,k_2} is usually set to the bitwidth.

[3]in order to formulate maximum timing constraints exchange the $\geq$ resp. $\leq$ relations of the sum boundaries

A naive implementation for computing w_{k_1,k_2} would contain terms of the form $x_{i,j_1,k_1} * x_{i,j_2,k_2}$, which are quadratic in x. The corresponding quadratic assignment problem can be avoided by using a trick published by Rim, Jain and De Leone [RJL92]. The trick consists in defining w_{k_1,k_2} as:

$$\forall j_1 \prec j_2: \qquad w_{k_1,k_2} \geq \left(\sum_{i_1 \in R(j_1)} x_{i_1,j_1,k_1} + \sum_{i_2 \in R(j_2)} x_{i,j_2,k_2} \right) - 1 \tag{4.15}$$

$$w_{k_1,k_2} \geq 0 \tag{4.16}$$

If both variables x_{i,j_1,k_1} and x_{i,j_2,k_2} are set to 1, w_{k_1,k_2} also becomes 1. In the case that only one of both variables takes the value of 1, w_{k_1,k_2} becomes 0. Constraint (4.16) avoids negative values if both variables are set to 0. Consider that a variable w_{k_1,k_2} is only created if both j_1 and j_2 can be executed on instances k_1 resp. k_2. With this trick, the cost function is still a linear function in x.

Hence, algorithms for solving (linear) integer programming problems can be used to compute optimal bindings.

Chapter 5

Intelligent library component selection and management

In the last chapter we described the underlying IP-model in detail. However, the actual high-level synthesis – realized by the subtasks *scheduling, allocation, binding* – is only one part in the OSCAR system. The other part concerns the intelligent component selection and library management. In this chapter now, we deal with this important part.

5.1 Overview about required libraries

The selection and management of components requires a large amount of information about supported operations inclusively their attributes, rules for transforming these operations or entire expressions to equivalent ones and the availability of suitable components. The OSCAR system makes use of four libraries:

The *function attribute library (FAL)* contains attributes of supported operators and functions. Synthesis fails, if a used function is not defined in the FAL or has no synthesizable VHDL-body.

The *algebraic rule library* is employed to transform algebraic expressions to equivalent ones.

The *cell library* contains cells which are specified by their ports and their internal structure. In OSCAR, we need an additional representation of their behavior together with area and timing information. Both specifications, the structure as well as the functional specification are stored in two different views.

1. *Functional view*: The functional description contains the behavioral cell description inclusively several attributes like cell area, timing etc. Since this view is fundamental to generate both the functional and structural component description it must contain additional compiler information as well. This information refers to available bit widths and compiler directives.

2. *Layout view*: This kind of representation is very CAD tool dependent and has no importance for

the actual selection of library components. A description of this view is omitted in this report.

The *component library* contains all available components in the required bit widths. Since it is unnecessary to compile all potentially employed components of different bit widths in advance, we divided this library into two views as well :

1. *Functional View*: The functional description is based on implication rules [Mar90]. For example, an ALU may be capable to perform an addition on the ports `a` and `b` controlled by a certain control signal `c`. We describe its behavior by

   ```
   ALU (a, b, '0', f) -> a + b
   ```

   The left side of this implication is represented in the way of a conventional function call. The arguments (in this example `a`, `b`, `c = '0'`, `f`) correspond to the signals which are implemented as interconnects in the final netlist. The right side of the implication consists of the algebraic expression to be performed for this function call.

   Additional entries in this library inform about port names, area and the timing.

2. *Netlist view*: The compilation of bit slices will be performed on demand by the underlying CAD tool at the end of the synthesis process.

5.2 Dataflow-/ Controlflow Specification

For the internal representation of the data and control flow, Assign Decision Diagrams (ADD) are employed in our synthesis system (see figure 5.1). This data structure, introduced by CHAIYAKUL, GAJSKI et al. in [VC92][HCG92] combines the data- and control-flow and minimizes the syntactical variance in behavioral descriptions at same time. "Syntactical variance" is understood as the possiblity to describe a certain hardware structure by different behavioral descriptions.

Figure 5.1 shows the basic structure of an ADD. An ADD represents the maximum parallel form of a given behavioral description. This means, all operations, which can be performed in parallel occur on different data-flow graphs. However, different operators can be mapped to the same function units depending of the scheduling, allocation and binding steps in the synthesis process.

The representation of an ADD consists of the following parts:

- Assignment Value Part: This graph consists of read nodes containing the input values and data flow graphs (represented as ovals in figure 5.1). Each data flow graph (DFG) corresponds to one basic block occuring in the input description. Nodes in the DFG correspond to operators (operator nodes), edges between operator node represent the data dependencies. The results of the single DFGs are used as inputs to the assignment decision nodes (ADNs). Each ADN is controlled by assigment conditions and writes the selected DFG result to the write node.

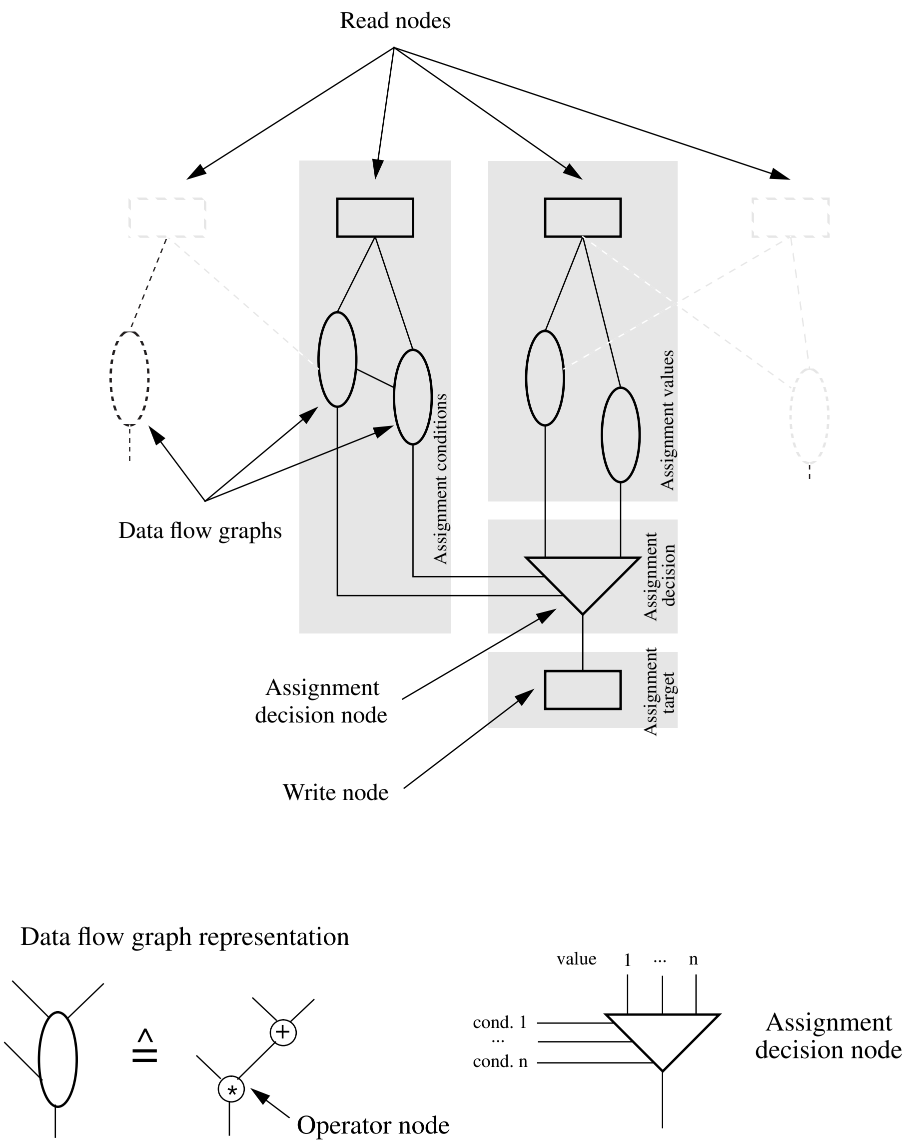


Figure 5.1: Assignment Decision Diagram

- Assignment Condition Part: The selection of a certain result of DFGs within the assignment value part depends on conditions occuring in the behavioral description. Such conditions can be formulated by statements like "IF <*condition*> THEN <*expr1*> ELSE <*expr2*>". The assignment condition part represents the evaluation of *condition* to select either the result of *expr1* or *expr2* by the ADN.

- Assignment Decision Part: The assigment decision nodes (ADNs) select one of several results coming from in the assignment value part. The selection is controlled by the mutually exclusive inputs from the assignment condition part.

- Assignment Target: The output of each ADN is passed into the write node and thus can be used as input to the ADD in subsequent control steps.

In addition to the information about the data and control flow, the ADD must be extended to include timing constraints. We call this extended representation *TADD*. Assuming that results of time constrained operations are saved in registers, we can insert additional edges between the read and write nodes. These edges are labelled with intervals representing the minimum and maximum timing.

5.3 Analysis of the data flow graph

In sections 5.1 and 5.2 we gave a short overview about the employed libraries in the OSCAR system and the CDFG representation as well. Now we want to follow the path in the design flow, beginning with the behavioral input description down to the final netlist.

5.3.1 Operator extraction

After parsing the behavioral description and the additional synthesis specification by the front end the TADD will be generated. The DFG analyzer extracts operators inclusively their bit widths from the DFG. We call such operators *primary operators*. By way of contrast, we obtain *secondary operators* by applying algebraic rules to primary or other secondary operators.

In the following, we will describe the component selection process on the basis of a simple example:

```
ARCHITECTURE behavior OF example IS
    signal a, b, f: bit_vector (15 downto 0);

BEGIN
    f <= (a * 2) + (b * 2);
END behavior;
```

The arithmetic expression shown in this example consists of the two operators `+` and `*` whose attributes are stored in the function attribute library. We briefly describe the attributes of the add-operation:

```
FUNCTION          +
ARGUMENTS         2
COMMUTATIVITY     TRUE
ASSOCIATIVITY     TRUE
NEUTRALS          X, 0
EXTENSIONS        DONT_CARE, DONT_CARE
RESULT_WIDTH      MAX (#1, #2) + 1
```

The first entry declares the operator name (`+`) followed by the number of arguments. The entries `COMMUTATIVITY` and `ASSOCIATIVITY` indicate that an expression potentially has alternative representation forms (e.g. `a + b = b + a`, `(a + b) + c = a + (b + c)`). Neutrals may be applied to the ports in order to perform the identity function `f = x`. Due to the commutativity in this case it is suffient to specify only one valid entry – others can be simply derived (here: applying the neutral "`0`": `f = x` $\Leftrightarrow$ `f = x + 0 = 0 + x`). In order to execute operations on components with larger bit widths than the relating arguments, additional inputs must be applied to the remaining port lines. Valid entries are the `SIGN-`/`ONE-`/`ZERO-`/`NO-` extension as well as `DONT_CARE`. `NO`-extension means that the bit widths of the component and the corresponding operator *must* be identical. `RESULT_WIDTH` is required to determine the bit width of the result dependent of its argument bit widths. Due to the carry bit, the result width of an addition is always `MAX (#1, #2) + 1` (`#`n means the bit width of the n-th argument).

After extracting the two primary operators (`"+"` and `"* 2"`) from the DFG the *operation list* consists of the following entries:

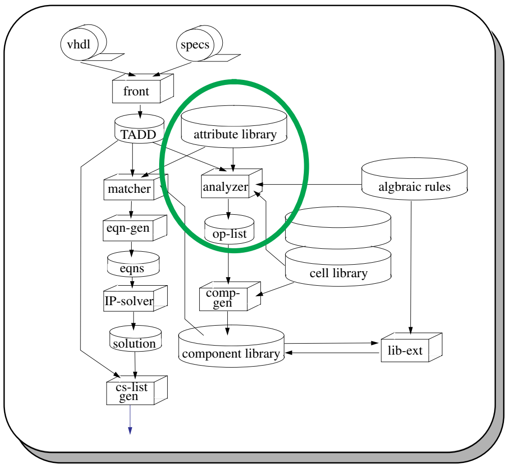
```
( * [15:0] 2 [15:0] )
( + [15:0] [15:0] [15:0] )
```

Each entry consists of the operator identifier, the bit width of the calculated result and the bit widths of the arguments. Constants, belonging to the analyzed operation will be also saved in the operation list.

The operation list may also contain entire expressions. In this case, arguments represent the subexpressions.

5.3.2 Improving the applicability of components by algebraic transformations

We essentially exploit algebraic transformations in order to extend the applicability of available function units. Due to this, we can employ restricted component libraries even in cases in which simple



System Overview

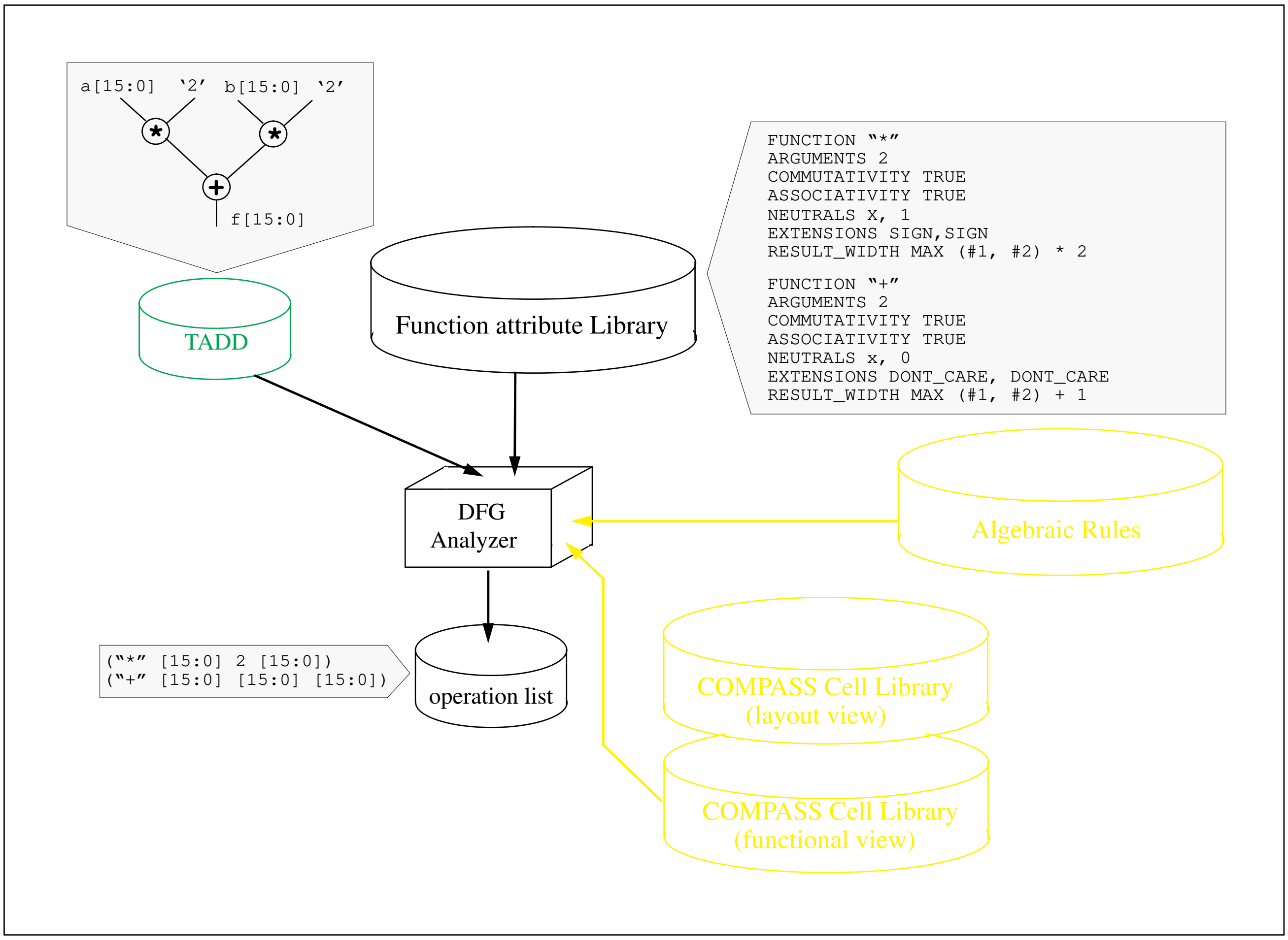


Figure 5.2: Extracting *primary* operators

component selection methods could fail or lead to inferior results. Algebraic rules are represented in the following format:

(<*left-expr*>) `->` (<*right-expr*>)

Both the left and the right side of each rule are represented in prefix notation.

Continuing or example, we are now able to replace `(a * 2)` (resp. `(b * 2)`) using the algebraic transformations:

```
(* a 2) -> (+ a a)
(* a 2) -> (shl a 1)
```

Consequently, the operation list can be extended by the entries `("+" [15:0] [15:0] [15:0])` and `(shl [15:0] 1 [15:0])` (fig. 5.3).

5.3.3 Handling complex expressions

Up to now, we treated complex expressions by dividing them into simple operations. However, libraries frequently contain complex components like MACS, so it seems to be sensible to employ such components in their entirety.

Figure 5.4 represents the analysis process taking information about available components into account. The exemplary component library contains an AMD 2901 like ALU which is capable to perform an add- and a shift-operation in sequence.

Generally, algebraic rules are not restricted to simple operations. Also complex expressions corresponding to complex components can be described. Rules of this type, like the distributive law

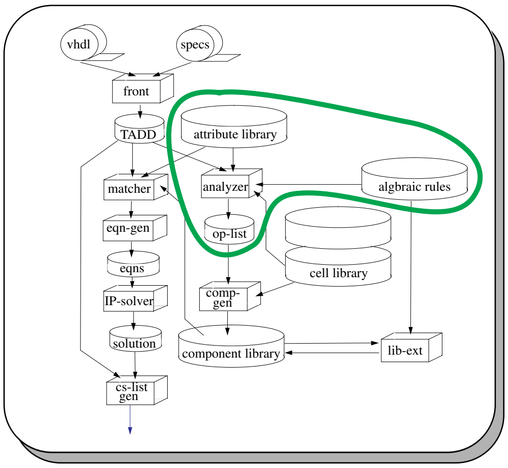
```
(a * c) + (b * c) -> (a + b) * c
```

are necessary to perform component selection exceeding simple operations. In combination with the rule `a * 2 -> a shl 1`, the expression `(a * 2) + (b * 2)` can be transformed into `(a + b) shl 1` which can be computed now by a single complex component.

At the end of the DFG analysis, the operation list contains all primary operators, operators derived by algebraic transformations (secondary ops.), and finally expressions executable in their entirety on complex components (macro-operations).

Continuing our example (see fig. 5.4) the final operation list consists of the following entries:

```
( * [15:0] 2 [15:0] )
( + [15:0] [15:0] [15:0] )
( shl [15:0] [15:0] 1 )
```



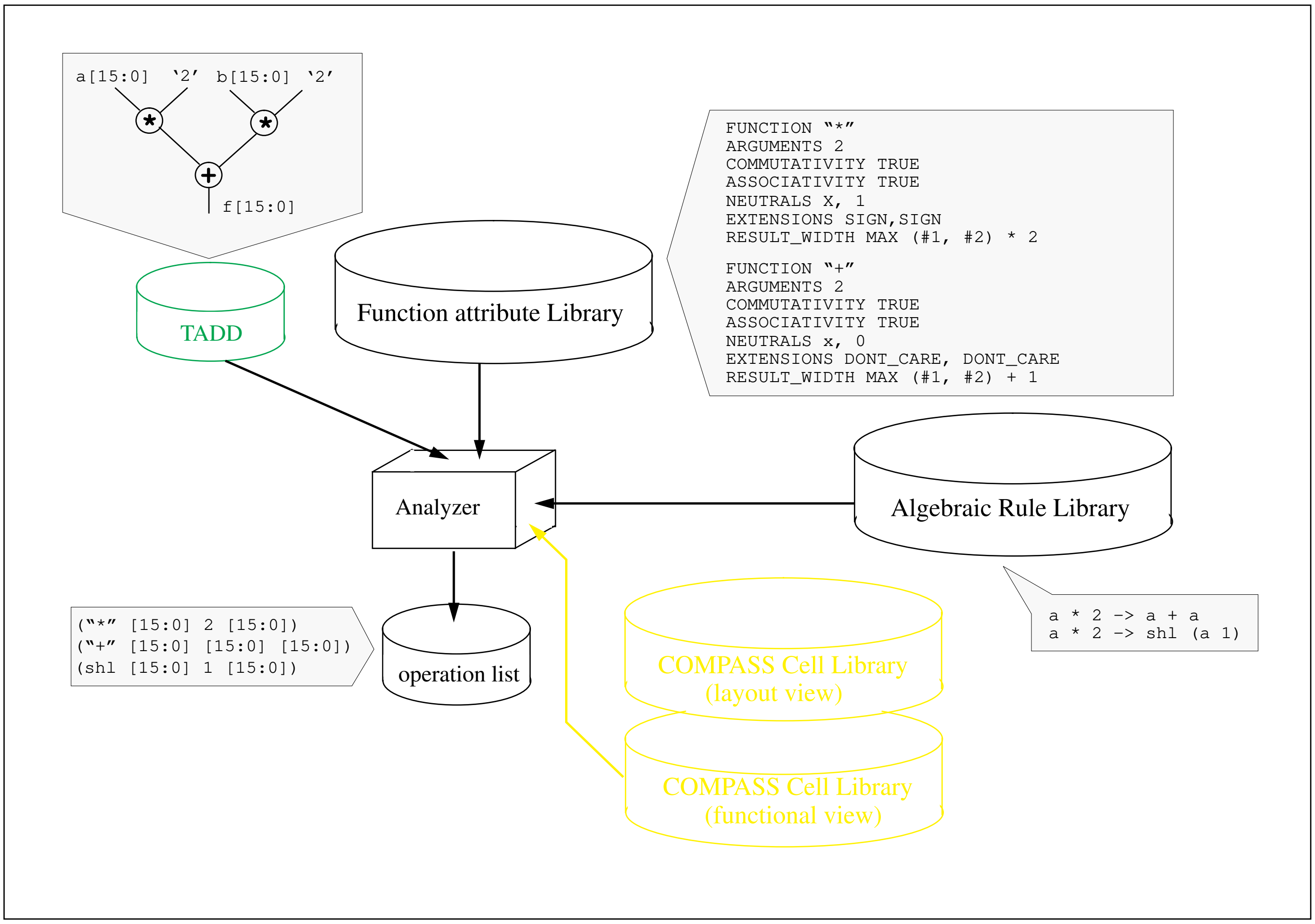
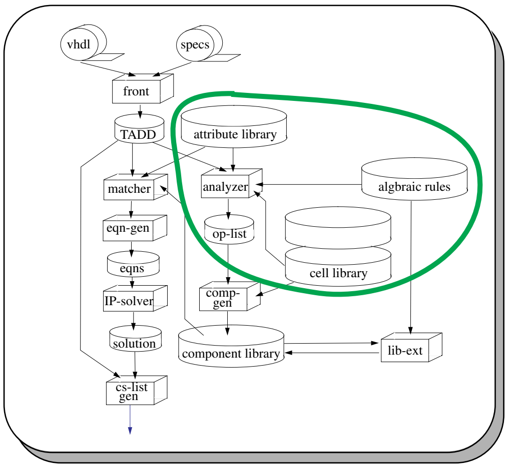


Figure 5.3: Extracting *secondary* operators



System Overview

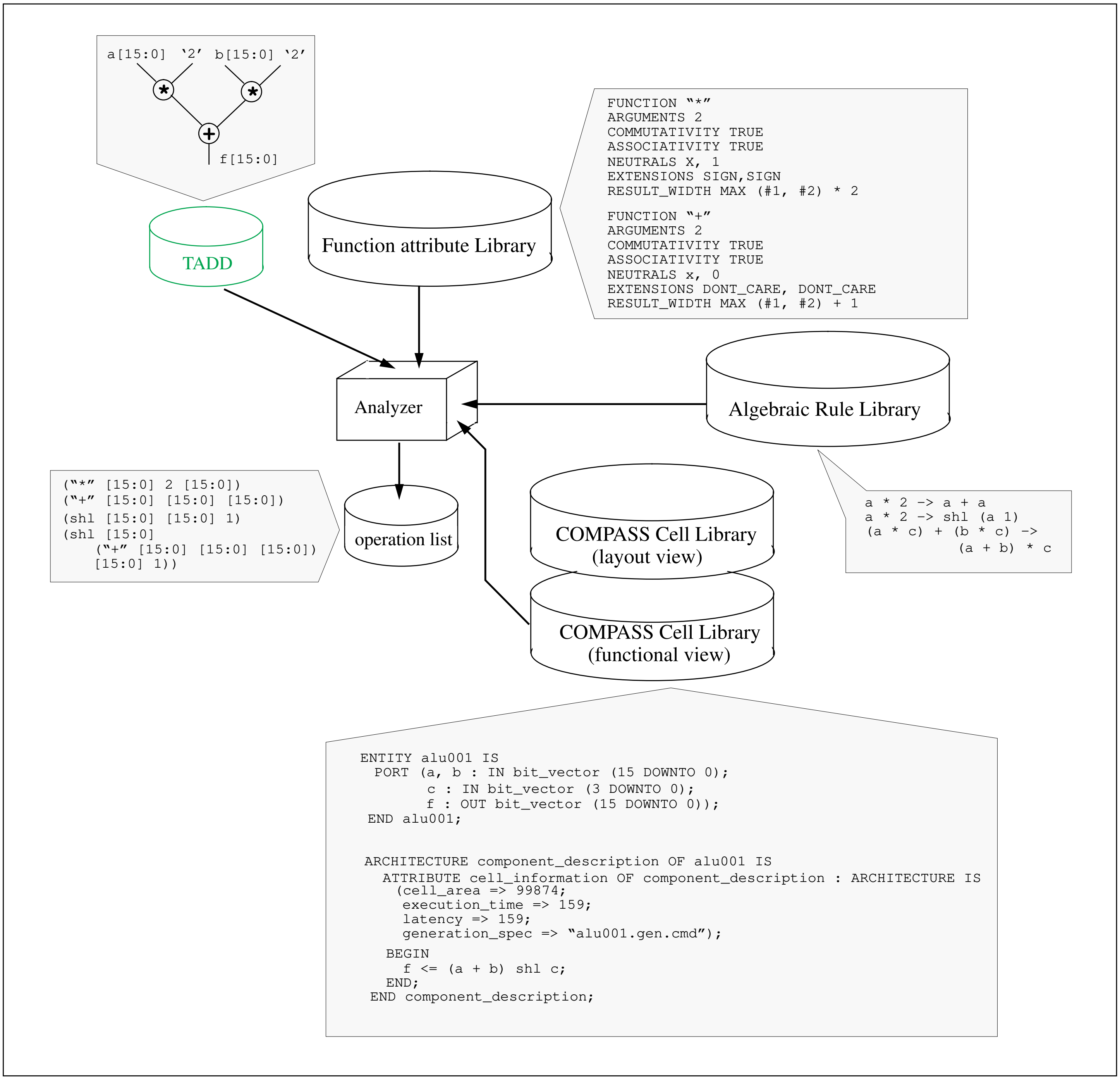


Figure 5.4: Selecting complex components

5.3.4 Implication rule based component representation

The component library contains the complete component specification including the interface declaration as well as the behavioral description. Due to this, a complete structural component description, based on a layout or netlist representation can be obtained. Consequently, the component library is well suited as interface to arbitrary component libraries, since all information about ports, behavior, timing and areas are included.

We give a short overview of the internal representation on the basis of a behavioral VHDL component description (see fig. 5.5).

```
COMPONENT       addsub6x16_1
CELL_NAME       vdp1asb001
AREA            444416
PORTS           (a [15:0], b [15:0], c [1], f [15:0])
ACTIVATION      ( (a [15:0], b [15:0], '0', f [15:0]) )
                    -> (= f [15:0] (+ a [15:0] b [15:0]) 140 140)
                ( (a [15:0], b [15:0], '1', f [15:0]) )
                    -> (= f [15:0] (- a [15:0] b [15:0]) 140 140)
```

The first two entries "`COMPONENT`" and "`CELL_NAME`" represent the component name and its underlying cell. The entry "`AREA`" measures the required area in λ^2. The port declaration consists of the port name together with the corresponding bit width. The functional description of each component is specified by a set of so-called implication rules: a certain application of (control) signals to the inports implies a function to be performed.

This way of representing component behavior by implication rules is described in the following sections.

Functional component description based on implication rules

The following VHDL description represents the behavior of a simple ALU:

```
ENTITY alu IS PORT (a, b :  IN bit_vector (15 Downto 0);
                   c :  IN bit;
                   f :  OUT bit_vector (15 Downto 0));
END alu;

ARCHITECTURE component_descriptions OF alu IS
BEGIN
           WITH c SELECT
           f <= a + b WHEN '0',
           a - b WHEN '1';
END component_description;
```

Figure 5.5: VHDL-description of an adder-subtractor

Its behavior can by simply described by the following implications rules:

```
alu (a, b, '0', f)   ->   f = a + b
alu (a, b, '1', f)   ->   f = a - b
```

<————— 'can be implemented by'

Signals on the left side correspond to the port declaration occuring in the ENTITY description. Arguments on the right side correspond to the port names and are used as free variables in the sense of transformational reasoning.

For each implication of the form <*activation*> `->` <*expression*> the relation "*can be implemented by*" is defined, which is determines the required control signals for a given operation. In this example, the behavior of the component is represented by two implication rules selecting either the addition or the subtraction.

This representation of multi-functional units can be simply extended to complex components.

The complex component used in our example is capable to compute each of the following expressions by applying neutrals to several inports:

```
alu (a, b, c, f)    ->    f = (a - b) shl c
alu (a, b, 0, f)    ->    f = a - b
alu (a, 0, c, f)    ->    f = a shl c
```

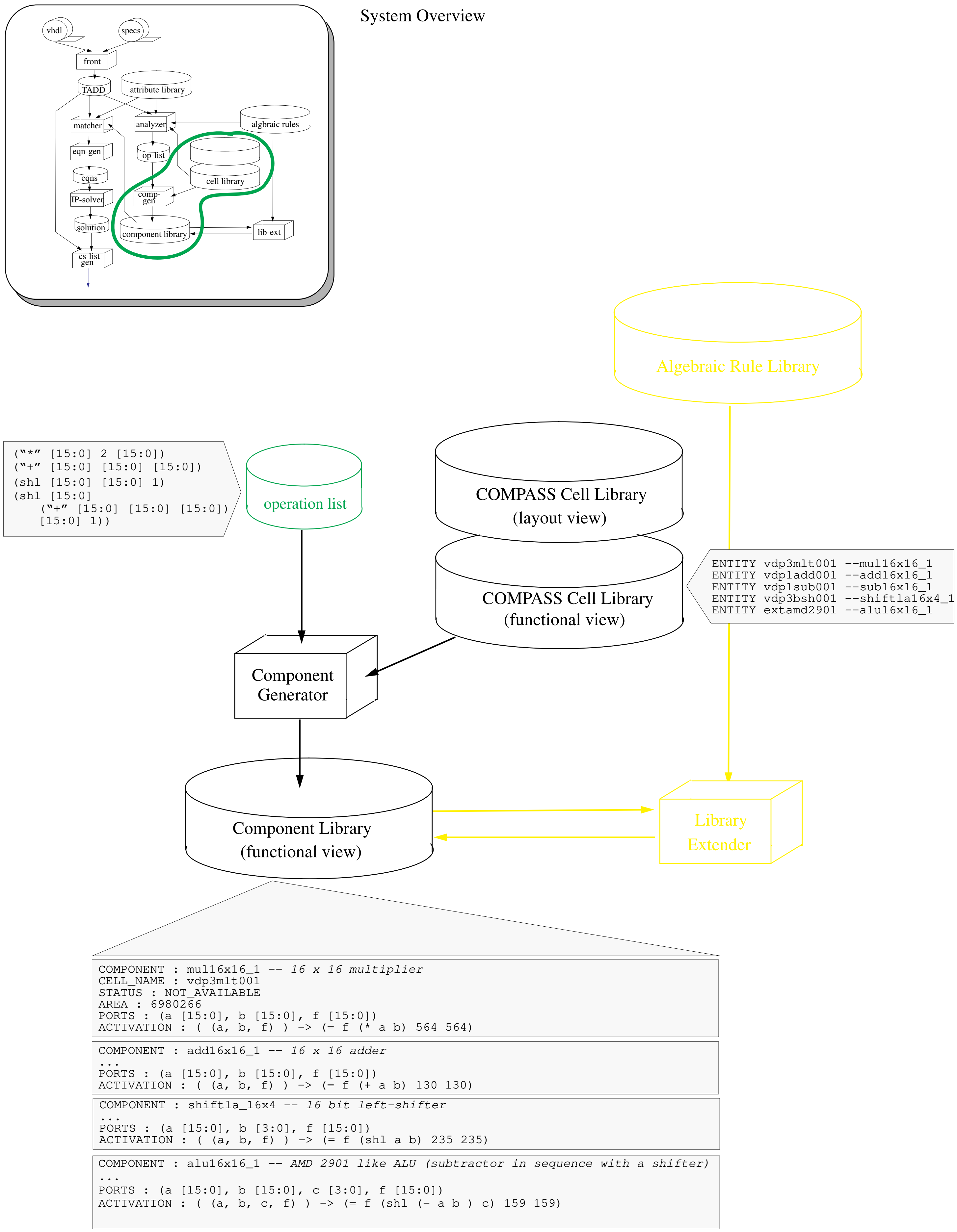


Figure 5.6: Functional component descriptions

Representation of multi cycle operations

Function units, which require more than one cycle to load data and to perform the selected operation can be described by implications as well. In this case, one component activation consists of several entries corresponding to single cycles.

The implication below represents a registered adder, which requires two cycles to load data into an internal register and perform the add operation.

```
radd (  (a, b, '0', X)                  – load registers, control signal: '0'
        (X, X, '1', f) )                – add registers, control signal: '1'
                         -> f = a + b
```

In the first cycle, the internal register is loaded with the external input values, appearing at ports `a` and `b`. Since the internal register is not required for synthesis, it may be hidden to the component description. In the second control step, both registered values are added and passed to the outport `f`.

5.3.5 Extending the applicability of components

Up till now, we applied algebraic transformations to the DFGs in order to extend the operation list by equivalent (macro-)operations. In order to improve the component utilization, we apply algebraic rules once more to the activation entries in the component library.

We want to describe this procedure on the basis of the ALU the following port and activation specification.

```
PORT:        a, b, c, f
ACTIVATION:  alu (a, b, c, f) -> f = (a - b) shl c
```

The algebraic rule library contains the entries:

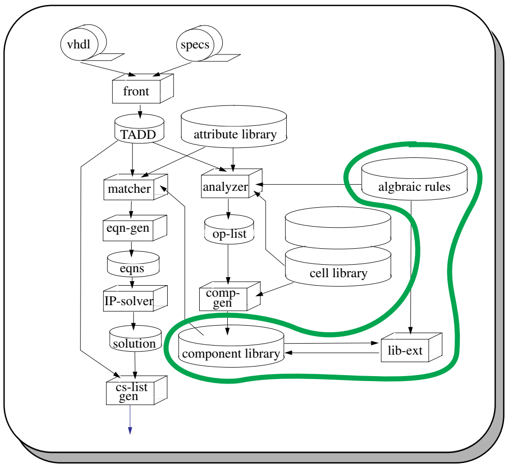
```
a shl 1       ->     a * 2
(a - b) * c   ->     (a * c) - (b * c)
```

Now, we can extend the activation entry by applying the two derived rules

```
(a - b) * 2     ->     (a * 2) - (b * 2)
(a - b) shl 1   ->     (a shl 1) - (b shl 1)
```

The generation of the additional activation entries is the task of the *library extender* which compares the right side of each component implication (function to be performed) with the left side of each rule.

The following activations can be generated by applying the specified rules.



System Overview

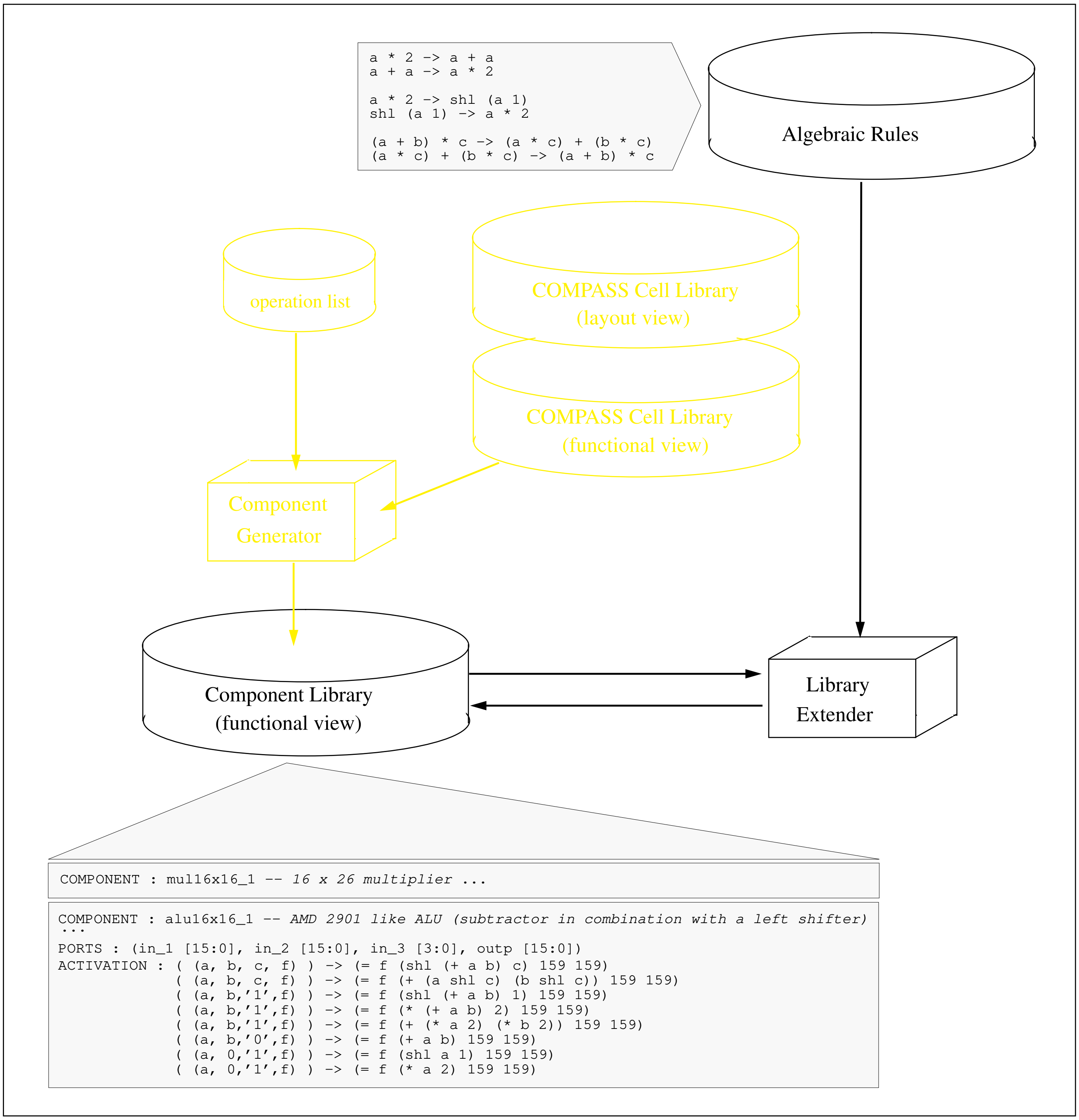


Figure 5.7: Extending the applicability of components

```
PORT:          a, b, c, f
ACTIVATION :   alu (a, b, c, f)    ->     f = (a - b) shl c
               alu (a, b, c, f)    ->     f = (a shl 1) - (b shl 1)
               alu (a, b,'1',f)    ->     f = (a - b) shl 2
               alu (a, b,'1',f)    ->     f = (a - b) * 2
               alu (a, b,'1',f)    ->     f = (a * 2) - (b * 2)
               alu (a, b,'0',f)    ->     f = (a - b)
               alu (a, 0,'1',f)    ->     f = (a shl 1)
               alu (a, 0,'1',f)    ->     f = (a * 2)
```

At the end of of the component section task, we have a complete component library allowing optimal matches between available function units and DFG operations. The next chapter deals with the link the actual high-level synthesis and the final steps to the netlist.

Chapter 6

Combining high-level synthesis and intelligent component selection

6.1 Interface to high-level synthesis

6.1.1 Component matching

The connecting link between the actual IP-model based HLS-part and the component selection part is realized by the *matching tool.* It performs a tree-matching between the TADD and the right sides of implication rules in the component library. The FAL contributes the required function attributes like commutativity, neutrals etc.

The matching tool creates for each operation a list of suitable components. In case that entire expressions can be mapped to single components, macro-operations are built. The matching process fails if at least one operation cannot be mapped.

6.1.2 Equation generation

Now, all equations can be generated for the constraints described in chapter 4. For this, the equation generator demands the information from the TADD and the function attribute library (operation assignment constraints, precedence constraint, chaining constraint), the list of matched components by the matching tool (resource assignment constraint) and finally timing-information from the design specification (timing constraint).

6.1.3 Generating the control-step-list

The output of the IP-solver representing the synthesis result in combination with the TADD is used to generate the control-step-list which is required for the netlist generation and controller synthesis in the subsequent steps.

For each (macro)-operation, the list contains the assigned control step, the component instance as well as a certain condition on which the operation will be performed.

6.2 Postprocessing

6.2.1 Netlist generation

The final netlist is generated by a traversal of the control-step-list. At this time, the component instances exist only in form of port declarations. The complilation of the data-path cells as well as controller synthesis is performed by the underlying CAD system (COMPASS).

6.2.2 FSM-description generation

The controller is specified on the basis of the control step list. Since this representation is independent of the underlying statemachine synthesis, arbitrary controller description can be generated. The states of the controller correspond to the control steps, the state transistions to the conditions. The output function of the state machine correspond to the component activations.

In a subsequent step, the controller description is synthesized and mapped to the target technology by the underlying COMPASS synthesis tool.

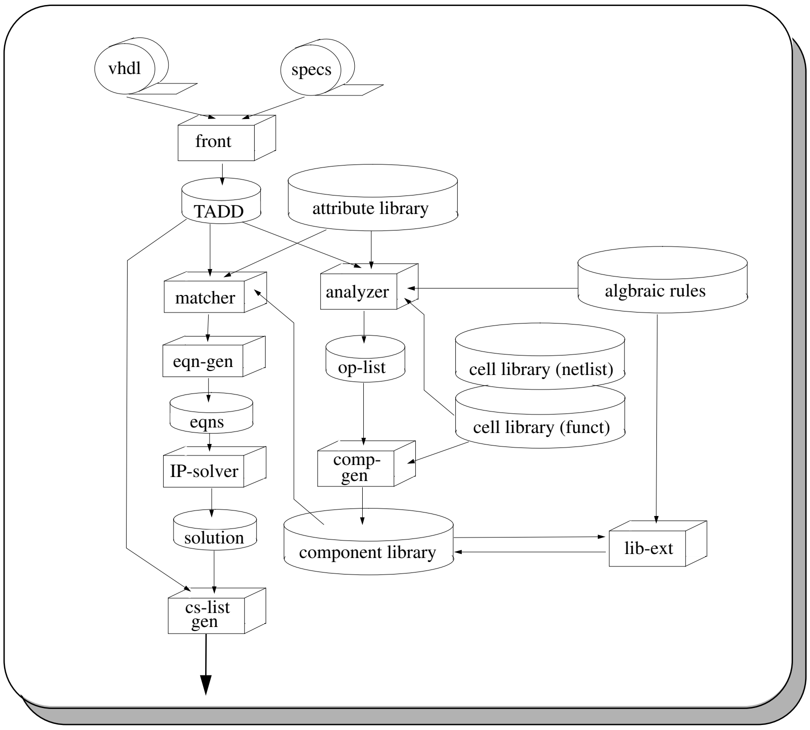
6.2.3 The role of COMPASS in the HLS-System

Our HLS-system expoits capabilities of COMPASS to map modules of the RT level to cells contained in the supplied cell library. The netlist and statemachine descriptions are passed to COMPASS via the *Logic* and *ASIC Synthesizer*.

The Logic Synthesizer reads the controller description and performs a synthesis process to obtain a gate level netlist. The ASIC Synthesizer assumes two tasks in our system: at first, it is used as front end to import the structural VHDL description delivered by the netlist generator. Secondly, it synthesizes all data path components, which are specified previously by the implications. The result of synthesizing these components consists of a netlist of slices.

After controller and component synthesis, three groups of netlists have been generated:

- gate netlist of the statemachine,
- netlists of instanciated components, and
- global netlist of the synthesized hardware which contain the controller and instanciated components



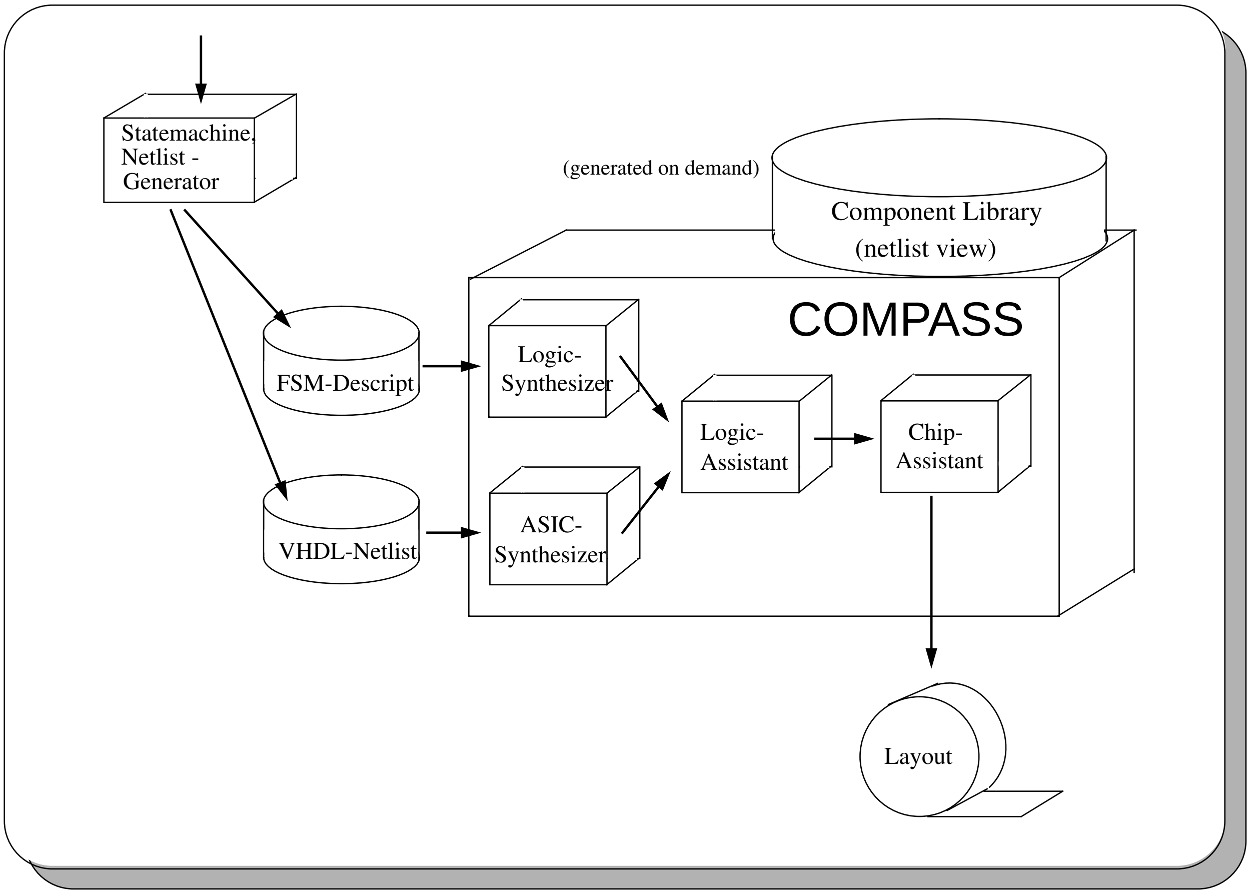


Figure 6.1: Employing commercial CAD-tools

The Logic Assistant merges all netlists yielding a complete structural description of the synthesized hardware. The final netlist can be passed in subsequent steps to floorplanning, placing and routing tools.

Chapter 7

Experimental results

We have applied our synthesis system to several benchmarks. All calculated results are optimal. The execution times have been measured on a Sparc 10 using the mixed IP-solver [Ber92].

In the following, we present experimental results for the 5th-order Elliptical Wave Filter [KWK85] and the Differential Equation Solver benchmark.

7.1 5^{th}-Order Elliptical Wave Filter

Table 7.1 shows the results of the EWF employing adders, multipliers and multi-functional units with different delays, latencies and costs. All delays are measured in control steps. The entry 2:1 means a pipelined function unit with a delay of two cycles. The specified costs represent the size relations between the particular component types. For each component type an upper bound of required instances was calculated by OSCAR, e.g. 2 out of 7 adders have been allocated for the first example.

FU	+	*	{+,*}		+	*	{+,*}		+	*	{+,*}	
delay	1	1	1		1	2	2		1	2:1	2:1	
costs	20	30	40	time	20	30	40	time	20	30	40	time
cstep 14	2/7	1/4	1/9	34s	-	-	-		-	-	-	
15	2/3	0/2	1/4	13s	-	-	-	-	-	-	-	-
16	2/3	1/1	0/3	170s	-	-	-	-	-	-	-	-
17	2/2	1/1	0/3	764s	4/7	2/4	0/8	13s	3/7	2/4	0/8	5s
18	⋮	⋮	⋮	⋮	2/4	2/2	0/6	33s	3/3	1/2	0/5	81s
19					2/2	2/2	0/4	1185s	2/3	1/1	0/4	349s

Table 7.1: Effect of different speeds of components

Table 7.2 presents results with chained (left part) and unchained (right part) operations. Chaining was applied to +,* -operations which allows to perform both operations together within one control step. In this example, all component delays are specified in ns.

FU		+	*	+	*
delay		$600ns$	$300ns$	$600ns$	$300ns$
costs		20	10	20	10
cstep	11	4/8	1/4	-	-
	12	3/4	1/1	-	-
	13	3/3	1/1	-	-
	14	⋮	⋮	3/7	2/4
	15			3/3	1/2
	16			2/3	1/1

Table 7.2: Enabling vs. disabling chaining of add/mult-operations

For this example, the application of chaining yields a reduction by 3 control steps of the mimimum schedule.

Table 7.3 presents synthesis results based on a complex component library. We employed MACs (see also figure 4.1) which are able to perform the addition and multiplication (separately or as macro-operation) within one control step.

FU		+	*	{*,+}	
delay		1	1	1	
costs		20	10	25	time
cstep	11	2/4	0/4	2/4	70s
	12	2/4	0/4	1/4	975s
	13	2/4	0/4	1/4	9519s
	14	2/4	0/4	1/4	9464s
	15	1/4	0/4	1/4	666s
	16	1/4	0/4	1/4	3195s

Table 7.3: Employing complex component libraries

7.2 Differential Equation Solver

The results calculated for the Diff-Eq benchmark are given in the following tables.

FU		+	-	*		+	-	*	
delay		450	450	700		450	450	700	
costs		20	20	30	time	20	20	30	time
cstep	3	1/2	2/2	3/4	1s	-	-	-	-
	4	1/1	1/2	2/3	1s	1/2	1/1	2/4	1s
	5	1/1	1/2	2/2	9s	1/1	1/1	2/2	3s
	6	1/1	1/2	2/2	187s	1/1	1/1	2/2	72s
	7	1/1	1/2	1/2	43s	1/1	1/1	1/2	14s

Table 7.4: Enabling vs. disabling chaining of sub/sub-operations

FU		+	-	*	{+,-}	{+,*}	
delay		1	1	1	1	1	
costs		20	20	30	25	40	time
cstep	4	0/2	0/1	1/4	1/3	1/6	3s
	5	0/2	0/1	2/2	1/2	0/3	128s
	6	0/1	0/1	2/2	1/1	0/3	3569s

FU		+	-	*	{+,-}	{+,*}	
delay		1	1	2	1	2	
costs		20	20	30	25	40	time
cstep	6	0/2	0/1	2/4	1/3	1/5	17s
	7	0/2	0/1	1/3	1/3	1/4	27s
	8	0/2	0/1	2/2	1/3	0/3	35s
	9	0/1	0/1	2/2	1/1	0/3	948s

FU		+	-	*	{+,-}	{+,*}	
delay		1	1	2:1	1	2:1	
costs		20	20	30	25	40	time
cstep	6	0/2	0/1	2/4	1/3	0/5	19 s
	7	0/1	0/1	2/2	1/1	0/3	607 s
	8	0/1	0/1	1/2	1/1	0/3	257 s
	9	0/1	0/1	1/1	1/1	0/2	121 s

Table 7.5: Effect of different speeds of components

Results of interconect minimization are given in table 7.6.

FU		+	-	*			+	-	*		
delay		1	1	1			1	1	1		
costs		20	20	30	interconnect	time	20	20	30	interconnect	time
cstep	4	1/1	1/1	2/2	5	1s	1/1	1/1	2/2	5	1s
	5	1/1	1/1	2/2	4	8s	1/1	1/1	2/2	5	3s
	6	1/1	1/1	2/2	4	149s	1/1	1/1	2/2	5	72s
	7	1/1	1/1	1/2	4	39s	1/1	1/1	1/2	4	14s

Table 7.6: Enabling vs. disabling interconnect minimization

7.3 Optimization of the Differential Equation Solver

In the previous section we gave experimental results for the differential equation solver. All calculated results are correct by construction and optimal with regard to a given objective function.

In section 5.3.3 we have proposed algebraic rules for a more efficient exploitation of the underlying component library. We have also shown that transformations can be used to yield cheaper and/or faster hardware realizations.

A typical example was the transformation rule `a * 2 -> a shl 1` such that the multiplication can be implemented by an offset in the wiring pattern. This special realization causes no costs and delay.

```
entity diffeq is
    port (<port declaration>);
end diffeq;

architecture behavior of diffeq is
begin
    process
        <variable declaration>
        <initialization>
        while x < a loop
            u1 := u * dx;
            u2 := 5 * x;
            u3 := 3 * y;
            y1 := u * dx;
            x  := x + dx;        considered basic block
            u4 := u1 * u2;
            u5 := dx * u3;
            y  := y + y1;
            u6 := u - u4;
            u  := u6 - u5;
            <output of x, y, u>
        end loop;
    end process;
end behavior;
```

The differential equation solver is a convenient example to present more of these tricks. We will show that the number of multiplications can be decreased by 2 with only two additional add- and one increment operation.

In the following, we only want to consider the basic block of this example. It consists of the expressions[1]:

expr$_1$ ::=
```
u' = (u - (u * dx) * (5 * x)) - (dx * (3 * y))
u' = u * (1 - 5 dx * x) - (3 dx * y)
```
[5 mult, 3 add]

expr$_2$ ::= `y1 = u * dx`

expr$_3$ ::= `x = x + dx`

expr$_4$ ::= `y = y + y1`

Obviously, expressions 2 - 4 cannot be simplified excepting that expr$_2$ is a subexpression of expr$_1$. However, expr$_1$ looks very promising. Before transformation it contains 5 multiplications and 2 additions. Since especially multiplications are characterized by expensive implementations, we want to concentrate on this operator at the beginning.

The algebraic rule library contains a set of applicable rules – denoted as R_1 to R_8.

$$
\begin{array}{llll}
R_1: & c * x & \rightarrow & (2^k - r) * x \\
R_2: & c * x & \rightarrow & (2^k + r) * x \\
R_3: & 2^k * x & \rightarrow & \text{x shl k} \qquad \textit{implementation}: \begin{cases} \text{hardwired}, & \text{if } k \text{ is const.} \\ \text{shift-register}, & \text{otherwise} \end{cases}
\end{array}
$$

Our first aim is to simplify products by splitting one factor into a power of two and a remainder. We can realize, that $5dx$ and $3dx$ can by simplified by these transformations. Obviously, the two products contain the same subexpression $t (= dx \text{ shl } 2)$ which can be simply implemented by interconnect.

$$
\begin{array}{lllll}
5dx & \rightarrow & 4dx + dx & \rightarrow & dx \text{ shl } 2 + dx \\
3dx & \rightarrow & 4dx - dx & \rightarrow & \underbrace{dx \text{ shl } 2}_{=::\, t} - dx
\end{array}
$$

After applying these transformations, expr$'_1$ consists now of only 3 multiplications and 4 additions.

$$\text{expr}'_1 ::= \texttt{u * } \underbrace{\texttt{(1 - (t + dx) * x)}}_{=::\, s} \texttt{ - ((t - dx) * y)} \qquad [3 \text{ mult}, 4 \text{ add}]$$

However, expr$_1$ contains still one subexpression s which can be simplified. $(1 - (t + dx) * x)$ can be transformed to $-((t + dx) * x) + 1$.

[1]Consider that u' represents the last value of variable u within this basic block since it appears at the right and left side of expr$_1$

$$\begin{array}{llll} R_4: & \text{-}expr & \rightarrow & \mathrm{inc}(\overline{expr}) \\ R_5: & \text{-}expr & \rightarrow & (\overline{expr}) + 1 \\ R_6: & x + 2^k & \rightarrow & \mathrm{inc}(x)_{[msb:k]} \\ R_7: & 1 - expr & \rightarrow & \mathrm{inc}(\overline{expr}) + 1 \\ R_8: & 1 - expr & \rightarrow & \mathrm{inc}(\overline{expr})_{[msb:1]} \end{array}$$

At first, we eliminate the sign by applying R_4 resp. R_5 such that s becomes to $\overline{((t+dx)*x)} + 2$. Now, R_6 can be applied in order to replace the addition. This is a valid transformation since $x + 2^k$ is equivalent to the increment operation which is applied to a slice (most significiant bit (MSB) to bit k) of x. R_8 represent the application of rules R_5, R_6, R_7 in one step.

Finally, expression expr_1'' contains only 3 multiplications (instead of 5), 3 additions and one additional increment.

expr_1'' ::= `u * (inc (`$\overline{\texttt{x * (t + dx)}}$`)[msb:1]) - ((t - dx) * y))` [3 mult, 3 add, 1 inc]

This example has shown that algebraic transformation can be especially applied to expressions which contain constants. In this example, we simplified the products $3dx$ and $5dx$ as well as the expression $1 - ((t + dx) * x)$.

However, algebraic transformations must be applied carefully. Even if they possibly mimimize the hardware costs for certain expression they can also increase the global hardware costs at the same time. A simple example makes this clear:

```
x := a + 1;
y := b + c;
```

Considering the first expression, we could apply the transformation `a + 1 -> inc (a)` to replace the add- by an increment operation. The second expression, however, contains an addition in any case which can not be replaced. This means that the transformation increases the global costs because an adder and an incrementer would be demanded. A local consideration of the first expression, however, would suggest the transformation.

In order to avoid such inferior results we must not replace the original operations but generate *additional* alternative solutions. Both the original and the transformed expression are considered as mutual exclusive in the IP-model such that only one them will be realized.

Figure 7.1 represents synthesis results for the DiffEq (top) and EWF (bottom) benchmarks. For each control step (x-axes) the light gray bars represent the number of required function units for the non-optimized version. The dark gray bars represent the optimized benchmark.

We can realize that for the DiffEq always 1 multiplier (instead of 2) is required. In order to yield the same solution by the original version we have to constrain the number of control steps to at least 10.

Similar results are calculated for the EWF benchmark. We are capable to generate results even for 11 control steps whereas the original version requires at least 14 control steps for an inferior result.

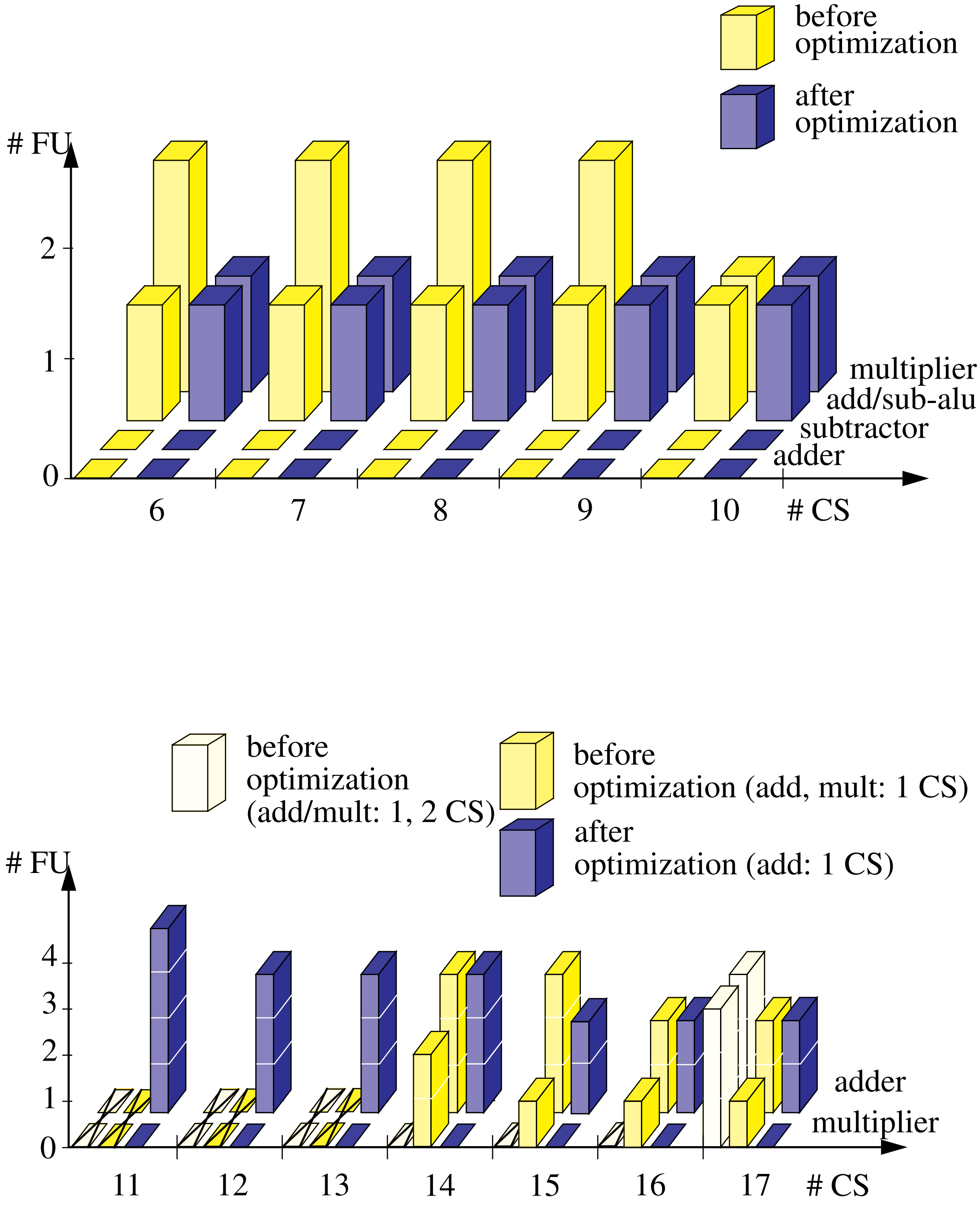


Figure 7.1: Results of the DiffEq and EWF

Chapter 8

Conclusion and future research

In this report we presented a new IP-model in combination with an intelligent method for component selection and management. We have extended existing approaches to IP-based high-level synthesis into several directions:

The presented IP-model extends previous models for scheduling and allocation to wiring optimization. In addition, is is capable of handling complex libraries with multi-functional components with mixed speeds. In contrast to traditional models, libraries may contain complex components such as multiplier-accumulators. Component libraries can be exploited more efficiently by applying algebraic transformations. Additionally, alternative data-path versions are supportedi. This allows to generate alternative versions and automatically determining the optimal data-path by the IP-solver. Additionally, the model supports the assigning of several operations operating on short bit vectors to components with a large bit width. Finally, the model supports chaining in its general form.

We have shown that acceptable runtimes can be achieved for standard benchmarks.

Our future research is concerned with methods for clever applications of algebraic transformations. We believe that genetic algorithms are practical due to the following facts:

1. expressions can be suitably encoded by *chromosoms*. Each chromosom consists of a sequence of *genes* which correspond to single subexpressions or function arguments.

2. genetic operators are able to modify the structure and the gene-information of chromosoms. The encoded algebraic expression can be modified by the following genetic operators. Each application of genetic operators must ensure the equivalence to the original expression.

   - *Mutation*: replace one subexpression by an other subexpression
   - *Insertion / deletion*: Insert resp. delete a redundant subexpression
   - *Translocation*: move parentheses in case of an associative function
   - *Inversion*: change the order of arguments in case of a commutative function

- *Crossing over*: exchange two subexpressions which are located on different chromosoms of the same population. All members of same population represent the same algebraic expression.

Details of the employed genetic algorithm will be decribed in a later report.

Bibliography

[Ach93] H. Achatz. Extended 0/1 LP formulation for the scheduling problem in high-level synthesis. *EURO-DAC'93*, 1993.

[Ber92] M.R.C.M. Berkelaar. Unix^{tm} manual page of lp_solve. *Eindhoven University of Technology, Design Automation Section*, 1992.

[Dut88] N. D. Dutt. GENIUS: A generic component library for high level synthesis. *Technical Report 88-22, U.C. Irvine*, 1988.

[GE91] C. H. Gebotys and M. I. Elmasry. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. *28th Design Automation Conference*, pages 2–7, 1991.

[GE93] C. H. Gebotys and M. I. Elmasry. Global optimization approach for architectural synthesis. *IEEE Transactions on CAD*, 1993.

[HCG92] T. Hadley, V. Chaiyakul, and D. D. Gajski. A data structure for interactive synthesis. *Technical Report 92-06, Dept. of Information and Computer Science, University of Irvine*, 1992.

[HLH91] C.-T. Hwang, J-H. Lee, and Y-C. Hsu. A formal approach to the scheduling problem in high-level synthesis. *IEEE Transactions on CAD*, 1991.

[KWK85] S. Y. Kung, H. J. Whitehouse, and T. Kailath. *VLSI and Modern Signal Processing.* Prentice Hall, 1985.

[Mar86] P. Marwedel. A new synthesis algorithm for the MIMOLA software system. *23rd Design Automation Conf.*, pages 271–277, 1986.

[Mar90] P. Marwedel. Matching system and component behaviour in MIMOLA synthesis tools. *Proc. 1st EDAC*, pages 146–156, 1990.

[MPC90] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proc. of the IEEE, Vol. 78*, pages 301–318, 1990.

[RJL92] M. Rim, R. Jain, and R. De Leone. Optimal allocation and binding in high level synthesis. *Proceedings of the 29th Design Automation Conference*, 1992.

[ST83] D.P. Siewiorek and C.J. Tseng. Facet: A procedure for the automated synthesis of digital systems. *20th Design Automation Conf.*, pages 490–496, 1983.

[VC92] L. Ramachandran V. Chaiyakul, D. D. Gajski. Minimizing syntactic variance with assignment decision diagrams. *Technical Report 92-34, Dept. of Information and Computer Science, University of Irvine*, 1992.

[WMGB93] T.C. Wilson, N. Mukherjee, M. K. Garg, and D. K. Banerji. An integrated and accelerated ilp solution for scheduling, module allocation, and binding in datapath synthesis. *6th International Conference on VLSI Design*, 1993.