OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming[1]

Birger Landwehr, Peter Marwedel, Rainer Dömer

University of Dortmund, Dept. of Computer Science XII, 44221 Dortmund, Germany

Abstract: This paper presents an approach to high-level synthesis which is based upon a 0/1 integer programming model. In contrast to other approaches, this 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. Furthermore, the model is capable of handling mixed speeds and chaining in its general form.

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 our paper, 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). OSCAR also respects timing constraints – a feature that was missing in early synthesis systems.

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 [13], Marwedel [10] and others (see [11] 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 [7]. Early work by Gebotys into this direction [4] did not include binding. Recently, the two-index model in [4] has been extended into a three index model (see e.g. [5]). 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 [5].
Other approaches which are based on IP-models include the following: Library mapping (see also Dutt [3]) is emphasized in a paper by Achatz [1], integrating scheduling and allocation. Allocation and assignment have been integrated for example by Rim and Jain [12].
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 is it 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. Our work aims at removing the limitations mentioned above.

3 The OSCAR System

3.1 Preprocessing

The input description of our high-level-synthesis system OSCAR is based on a subset of VHDL [8]. The input description is first compiled into a special form of data-flow graphs, namely Gajski's assignment decision diagrams (ADD) [14, 6]. These diagrams have been extended to include timing specifications. These extended ADDs are called *timed assignment decision diagrams* (TADDs).

3.2 Integrated Scheduling, Allocation and Binding

3.2.1 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.

$$x_{i,j,k} = \begin{cases} 1, & \text{if op. } j \text{ will be started on inst. } k \text{ at cs } i \\ 0, & \text{otherwise} \end{cases}$$

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 $\underline{type}$ 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 $\underline{optype}(j)$ is assumed to return the operation type of a certain operation instance j.
Table 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 example, if k cannot perform operation j, the corresponding decision variables are never generated in order to reduce the number of variables and relations.

3.2.2 Constraints

The following constraints are required to define the set of legal solutions:

$I \subset \mathbb{N}_0$	set of control steps
$i \in I$	control step $i \in I$
$J \subset \mathbb{N}_0$	set of ops in the TADD
$j \in J$	operation $j \in J$
$K \subset \mathbb{N}_0$	index set of resource instances
$k \in K$	resource instance $k \in K$
$M \subset \mathbb{N}_0$	index set of existing resource types
$m \in M$	resource type $m \in M$
$R(j)$	range of possible c-steps for op j
G_m	operation types executable by m
k_{max}	max number of all available instances
$\ell(j,k)$	latency of component k for operation j
$\underline{type}(k)$	component type of instance k
$G \subset \mathbb{N}_0$	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
$C(j) = \max_k C(j,k)$	max of delays for executing j on k

Table 1: Mathematical notation

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:

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

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:

$$j \text{ exec. on } k \iff \underline{optype}(j) \in G_{\underline{type}(k)} \qquad (2)$$

Note that relation "j executable on k" is more general than the corresponding implicit relation in [4]. 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

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

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

$$\forall k: \quad \text{if } type(k) = type(k+1) \text{ then } b_k \leq b_{k+1} \qquad (3)$$

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! This redundancy is not eliminated in other models [5].

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*[2] 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) 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. $\forall j \in J$:

$$\sum_{i \in R(j)} \sum_{\substack{k \in K \\ j \text{ exec on } k}} x_{i,j,k} + \sum_{\substack{y \in Y: \\ j \in y}} \sum_{i \in R(y)} \sum_{\substack{k \in K \\ y \text{ exec on } k}} x_{i,y,k} = 1 \quad (4)$$

If no complex components are employed, the right sum of (4) becomes 0. In this case the constraint reduces to the standard operation assignment constraint (1).

Additionally, constraint (5) restricts the assignment of macro-operations to at most one complex component:

$$\forall y \in Y: \sum_{i \in R(y)} \sum_{\substack{k \in K \\ y \text{ executable on } k}} x_{i,y,k} \leq 1 \qquad (5)$$

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

Before calculating the IP-model, a matching between the system and component behavior must be performed in a preprocessing phase.

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 cycle if the following presumptions are fulfilled: 1.) The number of port lines must be at least as large as the sum of argument bitwidths. 2.) A sufficient number of separation bits between the arguments must be inserted to avoid interactions between the operations. 3.) All unused input lines of the module must be *don't care*-extendable. These requirements are fulfilled by several function units like an n-bit adder. Such components can be employed to perform two data-independent additions which are defined on 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.

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: $\forall i \in I : \forall k \in K$:

$$\sum_{\substack{j \in J \\ j \text{ executable on } k}} \sum_{\substack{i'=i \\ i \in R(j)}}^{i+\ell(j,k)-1} x_{i',j,k} \leq b_k \qquad (6)$$

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 [5] by a single constraint set.

3. Precedence constraints

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

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

$$\sum_{\substack{k \\ j_2 \text{ executable on } k}} \sum_{\substack{i_2 \leq i - chain(j_1,j_2) \\ i_2 \in R(j_2)}} x_{i_2,j_2,k} \quad +$$

$$\sum_{\substack{k \\ j_1 \text{ executable on } k}} \sum_{\substack{i-(C(j_1,k)-1) \leq i_1 \\ i_1 \in R(j_1)}} x_{i_1,j_1,k} \leq 1 \qquad (7)$$

$C(j_1,k)$ 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 [4]. 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. 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

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

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: $j_i \lll j_{i+1}$
3.) ch is maximal:
$\nexists\, j_0 : j_0 \lll j_1 \wedge \nexists\, j_{n+1} : j_n \lll j_{n+1}$

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

Constraint (8) restricts the maximum number of chained operations $j \in ch$ per control step. In combination with constraint (7) only coherent operator chains are allowed to be assigned to the same control step. The maximum length of each chain is restricted by $time_{cycle} - \ell_{phy}$. $\forall ch \in CHAINS : \forall i \in I :$

$$\sum_{\substack{j \in ch: \\ i \in R(j)}} \ \sum_{\substack{k \in K \\ j \text{ exec on } k}} time(j, k) * x_{i,j,k} \leq \ time_{cycle} - \ell_{phy} \qquad (8)$$

Figure 1 illustrates the effect of chaining guided by a 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.

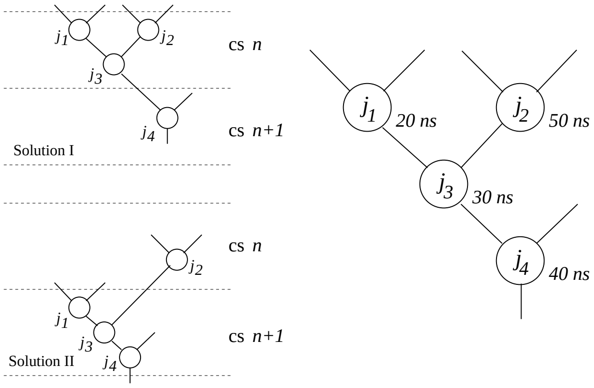

Figure 1: Operation chaining

In this example, ch_1 consists of $\{j_1, j_3, j_4\}$ because of $j_1 \prec j_3 \prec j_4$ and $j_1 \lll j_3, j_3 \lll j_4$. 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 (8) 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 (8).
5. Timing constraints
Models for timing constraints can be taken over from [4]. For example, if two operations j_1 and j_2 should be separated by T control steps, then the following relations should hold: $\forall i : i_1 \in R(j_1)$

$$\sum_{\substack{k \\ j_1 \text{ exec on } k}} x_{i_1,j_1,k} + \sum_{\substack{k \\ j_2 \text{ exec on } k}} \ \sum_{\substack{i_2 \neq i+T \\ i_2 \in R(j_2)}} x_{i_2,j_2,k} \leq 1 \qquad (9)$$

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

$$\begin{array}{ll} \forall i : & \displaystyle\sum_{\substack{k \\ j_1 \text{ exec on } k}} \ \sum_{\substack{i_1 \geq i \\ i_1 \in R(j_1)}} x_{i_1,j_1,k} \quad + \\ & \displaystyle\sum_{\substack{k \\ j_2 \text{ exec on } k}} \ \sum_{\substack{i_2 \leq i+(C(j_1,k)-1)+T \\ i_2 \in R(j_2)}} x_{i_2,j_2,k} \leq 1 \end{array} \qquad (10)$$

6. Register constraints
Register constraints can also be taken over from [5].

3.2.3 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:

$$\sum_{m \in M} (c_m * \sum_{\substack{k \in K \\ \underline{type(k)=m}}} b_k) + \sum_{k_1,k_2} c_{k_1,k_2} * w_{k_1,k_2} \qquad (11)$$

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

$$\text{with } w_{k_1,k_2} = \begin{cases} 1, & \text{if } k_2 \text{ is connected to } k_1 \\ 0, & \text{otherwise} \end{cases} \quad (12)$$

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.
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 [12]. The trick consists in defining w_{k_1,k_2} as: $\forall j_1 \prec j_2$:

$$w_{k_1,k_2} \geq \left(\sum_{i_1 \in R(j_1)} x_{i_1,j_1,k_1} + \sum_{i_2 \in R(j_2)} x_{i,j_2,k_2} \right) - 1 \quad (13)$$

$$w_{k_1,k_2} \geq 0 \quad (14)$$

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 (14) 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.

4 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 [2].
In the following, we present experimental results for the 5th-order Elliptical Wave Filter [9] and the Differential Equation Solver benchmark.

5^{th}-Order Elliptical Wave Filter
Table 2 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.
Table 3 presents results with chained (left part) and unchained (right part) operations. Chaining was applied to + and * -operations which allows to perform both operations together within one control step. Note, that all component delays are specified in ns. For this example, the application of chaining yields a reduction by 3 control steps of the minimum schedule presuming a clock cycle of $1000ns$. Table 4 presents synthesis results based on a complex component library. We employed MACs 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	30	40	t[s]
cs 14	2	1	1	34
15	2	0	1	13
16	2	1	0	170
17	2	1	0	764

FU	+	*	{+,*}		+	*	{+,*}	
delay	1	2	2		1	2:1	2:1	
costs	20	30	40	t[s]	20	30	40	t[s]
cs 17	4	2	0	13	3	2	0	5
18	2	2	0	33	3	1	0	81
19	2	2	0	1185	2	1	0	349

Table 2: Effect of different speeds of components

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

Table 3: Enabling vs. disabling chaining of add/mult-operations

Differential Equation Solver
The results calculated for the Diff-Eq benchmark are given in the tables 5-7.

Table 7 presents synthesis results of interconnect minimization. The number of interconnects could be reduced without increasing the number of required components.

5 Conclusion

We presented a new IP-model and its implementation in OSCAR which extends existing approaches to IP-based high-level synthesis into six different directions:
1.) OSCAR extends previous models for scheduling and allocation to wiring optimization.
2.) In addition, OSCAR is able to handle complex libraries with multi-functional components.
3.) Furthermore, OSCAR is able to handle components with mixed speeds.
4.) OSCAR has the potential to exploit the existence of complex components such as multiplier-accumulators.
5.) OSCAR has the capability of assigning several operations operating on short bit vectors to components with a large bit width.
6.) Finally, OSCAR allows chaining in its general form.

FU	+	*	MAC	
delay	1	1	1	
costs	20	10	25	time
cs 11	2	0	2	70s
12	2	0	1	975s
13	2	0	1	9519s
14	2	0	1	9464s
15	1	0	1	666s
16	1	0	1	3195s

Table 4: Employing complex component libraries

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

Table 5: Enabling vs. disabling chaining of sub/sub-operations

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

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 6: Effect of different speeds of components

FU	+	-	*			+	-	*		
delay	1	1	1			1	1	1		
costs	20	20	30	#w	t[s]	20	20	30	#w	t[s]
cs 4	1	1	2	5	1	1	1	2	5	1
5	1	1	2	4	8	1	1	2	5	3
6	1	1	2	4	149	1	1	2	5	72
7	1	1	1	4	39	1	1	1	4	14

Table 7: Enabling vs. disabling interconnect minimization

References

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

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] D. IEEE. IEEE standard VHDL language reference manual (IEEE Std. 1076-87). *IEEE Inc., New York*, 1988.

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

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

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

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

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

[14] L. R. 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.