Built-in Chaining:
Introducing Complex Components into Architectural Synthesis

Peter Marwedel, Birger Landwehr

Dept. of Computer Science XII
University of Dortmund
D-44221 Dortmund, Germany
e-mail: {marwedel, landwehr}@
ls12.informatik.uni-dortmund.de

Rainer Dömer

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
e-mail: doemer@ics.uci.edu

Abstract– In this paper, we extend the set of library components which are usually considered in architectural synthesis by components with built-in chaining. For such components, the result of some internally computed arithmetic function is made available as an argument to some other function through a local connection. These components can be used to implement chaining in a datapath in a single component. Components with built-in chaining are combinatorial circuits. They correspond to "complex gates" in logic synthesis. If compared to implementations with several components, components with built-in chaining usually provide a denser layout, reduced power consumption, and a shorter delay time. Multiplier/accumulators are the most prominent example of such components. Such components require new approaches for library mapping in architectural synthesis. In this paper, we describe an IP-based approach taken in our OSCAR synthesis system.

I. Motivation

Architectural synthesis (also known as *high-level synthesis (HLS)*) can be defined as the task of implementing a given behavioural specification by means of an appropriate register-transfer (RT-) level architecture. Architectural synthesis is considered to provide the next productivity boost for designers of information-processing devices.

RT-level components, which have been considered so far, include registers, register-files, busses, multiplexers and multi-functional units (ALUs). All functional units were considered to compute essentially a single (possibly control-selectable) standard function, such as addition, subtraction, or multiplication. On the other hand, current component libraries contain a growing number of components with *built-in chaining (BIC)* or *internal chaining*. With built-in chaining, the result of some computed standard function is made available as an argument to another function. For example, *multiplier/adders* multiply two numbers and add the result to a third one (see fig. 1). With BIC components, chaining in HLS can be implemented by a single component. In contrast, standard chaining requires external wiring between two or more components (therefore, we call it *external chaining*).

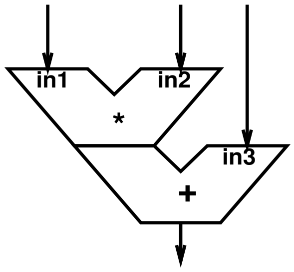

Figure 1: Multiplier/adder

In addition to multiplier/adders, multiplier/adder/accumulators (MACs), ALUs followed by (internal) shifters, and adders followed by (internal) comparators are components with internal chaining.

The main advantages of such components include:

- *The ability to generate efficient layout.* The layout of BIC components is usually more efficient than the combined layout of several independently designed functional units. For example, abutment of corresponding lines may be possible.
- *Delay and power consumption may be smaller than for separate components.* The delay of BIC components may be small due to a) the more compact layout, b) exploitation of context-dependent information during logic synthesis, and c) adjustment of the strength of drivers.
- *Design-reuse of complex components is facilitated.* Exploiting the presence of BIC components is important for re-using available valuable designs.

Example:

Table 1 shows area, delay times and power consumption of adders, multipliers, and multiplier-adders, respectively. The same information

*This work has been supported by the Commission of the European Communities under contract ESPRIT 6855 (LINK). In addition, the first author was partially supported through NATO grant # CRG 950910.

Cell	Area $[(k\lambda)^2]$	Power (@ 8 MHz) [mW]	Propagation delay [ns]
adder (64 Bit)	5.410	unknown	18.44
multiplier (32 Bit)	14.717	20.286	107.76
multiplier-adder-cell (32/64 bit) (built-in, internal chaining)	15.312	20.832	108.65
multiplier-cells + adder-cells (external chaining) (32/64 bit)	36.585	unknown	117.66

Table 1: Area, power, and propagation delay for commercial library components

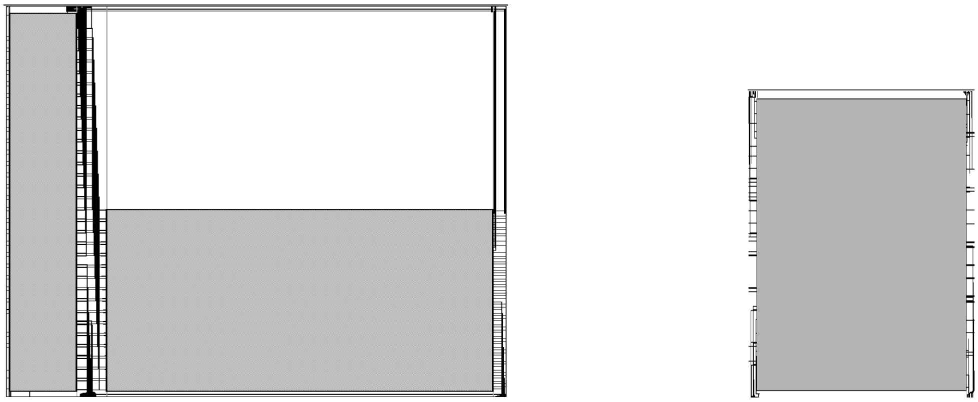

Figure 2: Layout for multiplier/adder with external vs. internal chaining

is also included for a multiplier-adder chain, built from individual multiplier and adder components (external chaining). In order to allow a fair comparison of the values, 64-bit adders have been used in all cases. The information has been generated with the help of the COMPASS DataPath Synthesizer for a 1μ CMOS technology.

Fig. 2 shows the layout of the design with external chaining.

Most importantly, the *area* for external chaining (computed as the bounding box of the combined layout), is more than two times larger than the BIC solution. This is caused by a very poor abutment of the two components. Some of the wiring area of the bounding box can possibly be saved, but external chaining will never be as efficient as internal chaining.

The *propagation delay* has been computed with the COMPASS QTV timing analyser. Indicated values represent the maximum of the values for rising and falling edges. It is obvious, that the delay of the implementation with external chaining is less than the sum of the delays of the individual components (this is the reason for using non-additive delays models, see e.g. Rabaey et al.). Nevertheless, the delay is even smaller if internal chaining is used.

BIC is important for many designs. Many digital signal processing architectures meet their constraints only because BIC is available. For example, many filter algorithms require MACs in order to meet their throughput constraints. Hence, the support of BIC in HLS is a must, despite the fact that the number of BIC components in libraries may not be very large.

The behavioural specification of digital signal processing architectures can be represented by *data-flow graphs (DFGs)*. Fig. 3 shows the DFG of the well-known elliptical wave filter and indicates where multiplier/adders can be used. It is obvious that quite a number of operations can be implemented by multiplier/adders.

II. Previous Work

In HLS synthesis, there has been only a very limited amount of work which took BIC components into account.

Work on the Cathedral silicon compilers (see e.g. [7]) is one of the few contributions to the area. The execution unit (EXU) model of Cathedral includes BIC. However, only an essentially fixed number of EXU types has been considered and no results are known which contribute to the topic of the paper.

Research at the University of Eindhoven is directed at creating regular layouts from netlists in which regularity is not immediately obvious [8]. As a special case, these algorithms would be able to create a regular layout if, for example, an adder follows a multiplier in the netlist. However, the approach does not include exploitation of BIC components in HLS.

In logic synthesis, the situation is different. So-called complex gates have been used in library mapping for many years (see [5] as an example).

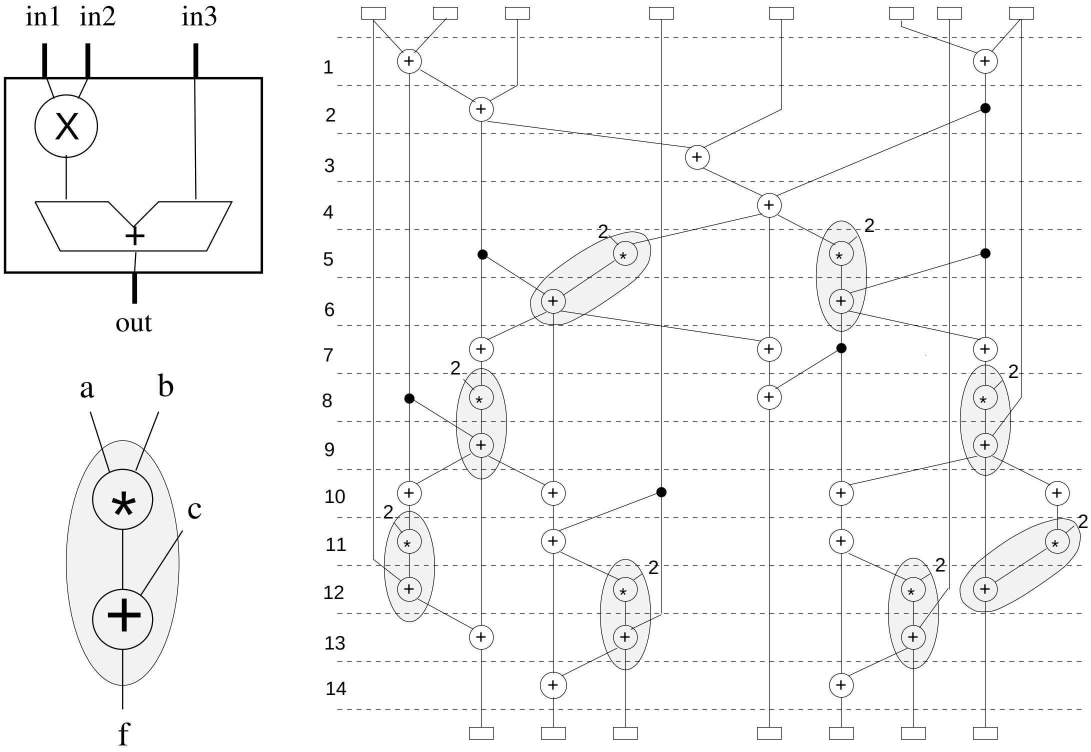

Figure 3: Use for multiplier/adders for elliptical wave filter

III. The problem

In order to support BIC components in HLS, several issues have to be considered. Let's start with an example to make these issues clear. Fig. 4 shows a section of a dataflow-graph (DFG).

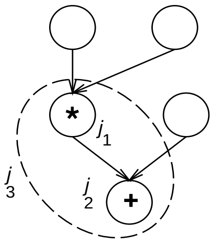

Figure 4: DFG with associated operation labels

Assume that a library containing adders, multipliers and multiplier/adders is given. There are several options for implementing this section of the DFG:

1. Implement $+$ and $*$ with two separate components in different control steps.

2. Implement $+$ and $*$ with two separate components in a single control step using standard chaining.

3. Implement $+$ and $*$ with a multiplier/adder in a single control step.

4. Implement $+$ and $*$ with a multiplier/adder performing a multi-cycle operation.

Clearly, decisions have to take many factors into account: delay, cost (power and area), intended clocking frequency, predicted wiring delays, and surrounding operations. Also, one might want to consider testability and error-recovery aspects of the different design options [3].

In this context, it has to be mentioned, that well-established standard techniques in HLS have to be revisited:

1. Scheduling and resource assignment can be modelled by functions taking DFG nodes (operations) as arguments and returning the corresponding control step and resource. In the context of BIC components, it is sometimes more adequate to use *sets of nodes or operations* as arguments.

2. The function performed by a component can no longer be described by a single operation identifier or an expression involving such a single identifier. Rather, expressions including several operation identifiers are required.

IV. Approach taken in OSCAR

A. Naming conventions

For our HLS system OSCAR (Optimum simultaneous scheduling, allocation and resource assignment), the ability to consider BIC components has been a major design goal. A very essential part of this is to select components for the final architecture. We use binary decision variables for modelling the selection of certain components:

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

Each $k \in K$ denotes a potential instance of a library element type $m \in M$. Before synthesis is started, a sufficient set of instance identifiers is made available for each $m \in M$. Fig. 5 is a graphical representation of naming in our model.

We distinguish between *behavioural* and *structural domains*. For each of these, we distinguish between *types* and *instances*. Specific types and instances are denoted by elements taken from sets of discrete elements. In actual implementations, sets of integers are used. In the paper, we are sometimes using sets of characters in order to improve readability. As can be seen from fig. 5, the sets for operation instances, operation types, component instances and component types are denoted by J, G, K, and M, respectively.

Component functionality is modelled by relation *executable on*.

Def.: $\forall j \in J, k \in K : j$ executable on $k \iff k \in K$ is able to perform operation $j \in J$.

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 after $\ell(j,k)$ control steps if j is the operation most recently started. $\ell(j,k)$ is called the component *data initiation intervall (dii)*.

Let G' denote the set of standard operations which are supported for an HLS system. In the case of OSCAR, G' corresponds to operations described in standard synthesis packages such as [9]. These operations can be represented by expressions containing single operation identifiers, for example "+" and "$*$". These operations are called *simple operations*.

In addition to these operations, OSCAR considers *complex operations* or *macro operations*, operations which can be performed by BIC components. If, for example, the library contains a MAC, then `((in1 * in2) + in3)` will be considered as a complex operation. Let G'' denote complex operation types.

Def.: $G = G' \cup G''$. G denotes the set of all operation types.

Operation instances corresponding to G, G' and G'' are denoted by sets J, J' and J'', respectively.

In OSCAR, complex operations in the DFG are labelled just like simple operations (see fig. 4). Note that a certain node in the DFG may belong to several labels, to one for the node as a simple operation and possibly also to others if it is an element of complex operations.

The essential task of HLS is to establish bindings between operation and component instances. Moreover, most HLS algorithms also generate bindings to *control steps* at which these operations are started. In OSCAR, control steps are represented by integers i from an index set I. Bindings are represented by decision variables x. In the case of predefined instance bindings or cost functions containing interconnect costs, OSCAR uses triple-indexed decision variables with the following definition:

$$x_{i,j,k} = \begin{cases} 1, & \text{if operation } j \text{ is started} \\ & \quad \text{on component instance } k \\ & \quad \text{at control step } i \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

For each operation $j \in J$, $R(j)$ denotes the set of control steps during which j could be executed. $R(j)$ can be computed by a simple ASAP/ALAP analysis or by techniques taking resource constraints into account.

B. The model

For a given component library, the cost function is a linear function of these variables.

$$\sum_{m \in M} (\text{COSTS}(m) * \sum_{\substack{k \in K \\ \underline{type}(k) = m}} b_k) \quad (3)$$

This cost function is minimized under constraints describing correct solutions. The following constraints are not affected by the need of modelling BIC (see [6] for detailed equations):

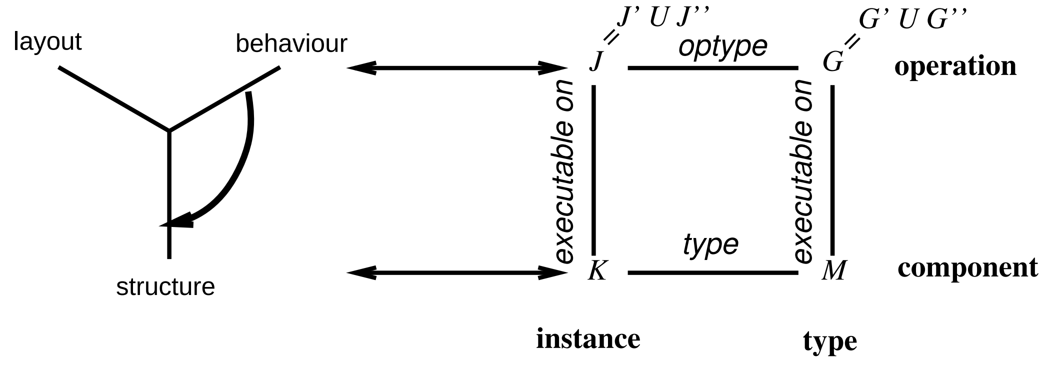

Figure 5: Naming conventions in the OSCAR model

1. **Timing constraints:**

   these constraints can be used to specify minimum and maximum delay between operations. Applied on read/write-operations it allows to meet given timing specifications.

2. **Precedence constraints:**

   These constraints reflect that operations cannot be started before their arguments have been computed.

3. **Chaining constraints**

   These constraints reflect the fact that the total combinatorial delay of the design should not exceed a given threshhold. In OSCAR, we assume that such a threshhold is known during the design process. An outer loop can be used to find a good threshhold [4]. Using this threshhold, OSCAR generates constraints in case the combined delay of data-dependent operations exceeds the threshhold.

   Example:
   Assume that the combined delay of j_1 and j_2 in fig. 4 exceeds the threshhold. Then, the following constraints can be generated:

   $$\forall i \in R(j_1) \cap R(j_2) : \sum_k (x_{i,j_1,k} + x_{i,j_2,k}) \leq 1 \qquad (4)$$

   □

   Chaining constraints are required in order to avoid solution 2 of section 3 in case high clocking frequencies have been specified. These constraints are not part of any other IP-model we are aware of.

Other constraints are affected by the need to model BIC:

4. **Resource assignment constraints**

   Resource assigment constraints guarantee that generated solutions respect the minimum data initiation intervall `dii`. `dii`'s are modelled by the following constraints:

   $$\forall k \in K : \sum_{\substack{j \in J \\ j \text{ executable on } k}} \sum_{\substack{i'=i \\ i \in R(j)}}^{i+\ell(j,k)-1} x_{i',j,k} \leq b_k \qquad (5)$$

   Example:
   Consider fig. 6. For the sake of better readability, we use a, m, M to denote an adder, a multiplier and a MAC, respectively. Furthermore, we use $+, *, \otimes$ to denote addition, multiplication and MAC operations.

   If the macro operation is assigned to a MAC and to control step $i = 2$, then no other operation can be performed on this component for control steps in the range $[i, i + \ell(j,k) - 1] = [2, 1 + \ell(j,k)]$. This is the situation described in fig. 6.

   □

   Since (5) does not distinguish between simple and complex operations, our model is able to handle components which can execute a mix of simple and complex operations.

   The form of equation (5) is very similar to resource constraints for other IP-based models. However, in our model, the set J includes complex operations. Also, two operations $j_1, j_2 \in J$ may, in fact, represent overlapping segments of the DFG.

   In previous approaches [2], constant 1 is used on the right hand side of resource constraints. The current approach is required for HLS with integrated scheduling and assignment in order to avoid solutions, in which operations are assigned to non-selected component instances, i.e. instances for which b_k is 0.

5. **Operation assignment constraints**

   These constraints reflect the fact that each operation has to be performed by a suitable hardware resource. This is guaranteed by the following constraints:

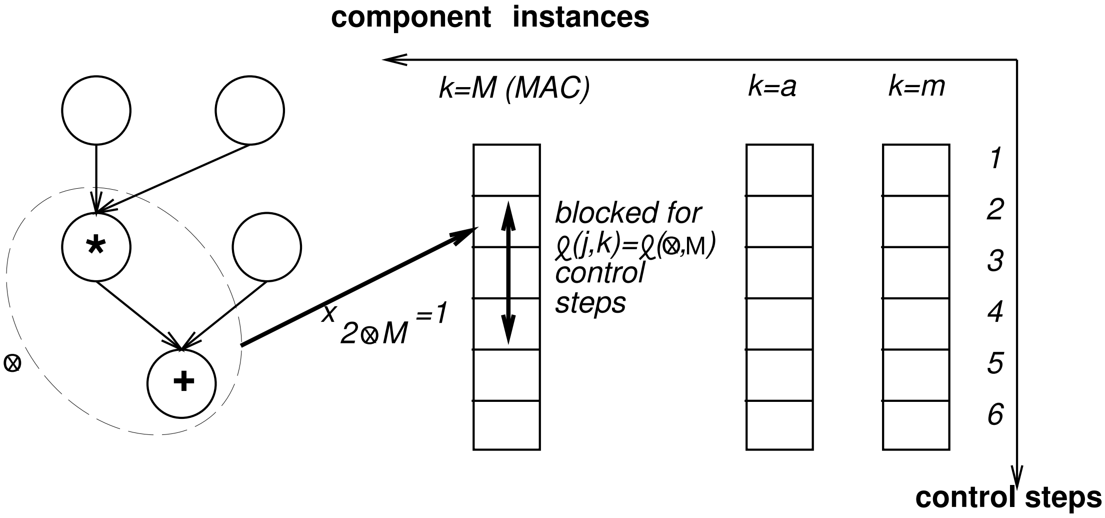

Figure 6: Graphical interpretation of assignment constraints

$$\forall j' \in J' : \sum_{i \in R(j')} \sum_{\substack{k \in K \\ j \text{ exec on } k}} x_{i,j',k} +$$

$$\sum_{\substack{j'' \in J'': \\ j' \in j''}} \sum_{i \in R(j'')} \sum_{\substack{k \in K \\ j'' \text{ exec on } k}} x_{i,j'',k} = 1 \qquad (6)$$

The meaning of these constraints is as follows:

Each simple operation j' has to be implemented. There are two ways of doing this:

a) j' is implemented individualy. In this case, the sum of $x_{i,j',k}$ over all possible control steps $R(j')$ and all components will be 1 (1st term in (6) = 1).

b) j' is implemented as part of a macro operation j''. j' may actually be part of several macro operations, but only one of these can be implemented as a macro operation. Hence, the sum of $x_{i,j'',k}$ over all macro operations j'' enclosing j', over all control steps $R(j'')$, and over all resources k capable of executing j'' must be one (second term in (6)).

Example:

Let us assume that $R(+) = 3$, $R(*) = 2$, $R(\otimes) = [2..3]$. Then, the following equations will be generated:

$$j' = * \quad : \quad x_{2*m} + x_{2\otimes M} + x_{3\otimes M} = 1 \qquad (7)$$

$$j' = + \quad : \quad x_{3+a} + x_{2\otimes M} + x_{3\otimes M} = 1 \qquad (8)$$

These equations guarantee that either the macro operation $\otimes$ or the simple operations will be bound to a hardware component.

□

By considering simple operations as a special case of macro operations, it would be possible to use just the right term of equation 6, but this would lead to some redundant computations.

V. Results

In the following, we will describe our results for two standard examples: the elliptical wave filter (EWF) and an edge-detection algorithm. All results use the components described in table 1. The cycle time has been set to 50 ns in order to allow a controller and multiplexer delay of about 34 ns for single-cycle adds. For this cycle time, multiplications and multiply/adds will always be multi-cycle operations.

None of the results uses external chaining. Hence, the following tables mainly demonstrate the effect of built-in chaining. The execution times have been measured on a SPARC 20 using the mixed IP-solver `lp_solve` [1].

Tables 2 and 3 show the results for the elliptical wave filter.

cs	add	mult	active area $[(k\lambda)^2]$	runtime [s]
20	3	4	66.083	≤ 1
21	2	3	48.961	≤ 1
22	2	2	34.244	3

Table 2: Results for EWF without BIC components

cs	add	mult	mac	active area $[(k\lambda)^2]$	runtime [s]
17	3	1	3	67.872	≤ 1
18	2	1	2	50.154	37
19	2	0	2	35.434	216

Table 3: Results for EWF with BIC components

Note that the use of BIC can be exploited in two ways: a) to get a smaller design if the cycle budget is fixed and b) to get a faster design if the area budget is fixed.

Tables 4 and 5 show the results for the edge detector algorithm.

cs	add	sub	mult	active area $[(k\lambda)^2]$	runtime [s]
12	2	2	8	127.418	≤ 1
13	2	2	7	112.700	≤ 1
14	2	2	4	68.548	57

Table 4: Results for edge detector without BIC components

cs	add	sub	mult	mac	active area $[(k\lambda)^2]$	runtime [s]
12	2	2	4	1	83.860	2
13	1	2	3	1	66.737	10
14	1	1	3	1	64.304	192

Table 5: Results for edge detector with BIC components

VI. Conclusion

In this paper, we have proposed to pay attention to the need of modelling components with built-in chaining (BIC). We have stressed the importance of the support of these components in high-level synthesis. As an example, we have shown how BIC can be modelled in integer-programming (IP) based synthesis algorithms. IP-models provide a basis for adding support for BIC in a rather straightforward way. Due to recent advances in IP-based modelling, we have been able to generate designs in acceptable computation time [6].

These results demonstrate the efficiency of designs using BIC.

The authors appreciate the comments of Fadi Kurdahi and Nikil Dutt (UC Irvine) on an earlier version of the manuscript.

References

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

[2] C. H. Gebotys and M. I. Elmasry. *Optimal VLSI Architectural Synthesis*. Kluwer Academic Publishers, 1992.

[3] I. G. Harris and A. Orailoglu. Microarchitectural synthesis of VLSI designs with high test concurrency. *31st ACM/IEEE Design Automation Conference*, pages 206–211, 1994.

[4] P. K. Jha, S. Parameswaran, and N. D. Dutt. Reclocking controllers for minimum execution time. Technical Report 94-40, Information and Computer Science, University of California at Irvine, 1994.

[5] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. *24th Design Automation Conference*, pages 341–347, 1987.

[6] B. Landwehr, P. Marwedel, and R. Dömer. OSCAR: Optimum simultaneous scheduling, allocation and resource binding based on integer programming. *Euro-DAC*, 1994.

[7] H. De Man, J. Rabaey, and P. Six. CATHEDRAL II: A synthesis and module generation system for multiprocessor systems on a chip. *in: G.DeMicheli, A.Sangiovanni-Vincentelli, P.Antognetti: Design Systems for VLSI Circuits–Logic Synthesis and Silicon Compilation–, Martinus Nijhoff Publishers*, 1987.

[8] R. Nijssen and J.A.G. Jess. Data path regularity extraction. *IFIP Workshop on Logic and Architecture Synthesis*, 1994.

[9] Special Interest Group on Synthesis from VHDL. VHDL arithmetic package for synthesis. *repository at INTERNET host "vhdl.org", login "anonymous", file "vi/vhdlsynth/numeric_bit.vhd"*, 1993.