Improving the Performance of High-Level Synthesis

Peter Marwedel
Wolfgang Schenk
Universität Kiel
Institut für Informatik und Prakt. Mathematik
Olshausenstr. 40-60
D-2300 Kiel, W. Germany

Abstract: *In this paper we study possible improvements of high-level (architectural) synthesis processes. We allow the designer to indicate a set of bindings between behaviour and structure in order to add some of the designer's knowledge to the design process. These bindings can be used to exclude inefficient designs. The remaining design space may then be studied in more detail, using unified backtracking. Backtracking, together with preliminary floor-planning, is required for area-efficient designs.*

1 Introduction

High-level synthesis is concerned with the mapping of behavioural descriptions into hardware structures. Behaviour frequently is described by imperative programs written in ISPS, subsets of PASCAL, VHDL, Occam or MIMOLA. The basic advantage of this approach is that it is potentially able to generate correct designs ("correctness by construction") in fast turnaround time. Due to this, high-level synthesis is becoming popular.

High-level synthesis is called high-level silicon compilation if it is supporting the implementation as an integrated circuit. Only few integrated circuits have been fabricated using the original results of a (general) high-level synthesis systems. The reason is, that the results in general are still poor. In the following we will analyse problems with current synthesis techniques. Futhermore, we propose ways of solving these problems. Instead of proposing a specific new synthesis method, we describe guidelines for new synthesis techniques.

1.1 Problems with Current Synthesis Systems

Below, we study the case of high-level synthesis algorithms generating a structure at the RT-level. The constituents of this structure are RT-components (ALUs, registers, RAMs, etc.) and their interconnections.

A number of problems exists with current synthesis algorithms, but two related important problems are generally present: the phase-coupling problem and the complexity problem.

1.1.1 Phase Coupling

Most synthesis systems partition the synthesis algorithm into a number of phases solving subproblems. Many of the subproblems can be considered as binding problems:

- Variable to storage (RAM or register) binding. This step includes:
  - Storage allocation for user variables.
  - Storage allocation for temporary variables.
- Control step binding. This step includes:
  - Decomposition of complex expressions.
  - Binding of assignments to control steps (instructions). This step is frequently called **scheduling**.
- Binding of operations to hardware components. This step includes:
  - Component selection (selection of library components or creation of new components),
  - Assignment of components and control code to every operation of the procedural specification. This includes:
    - The binding of arithmetic/logic operations to ALUs and similar networks,
    - The binding of READ- and WRITE-operations to RAM ports and
    - The binding of constants to hardwired constants, control word fields or decoders.

In addition, silicon compilers have to handle problems like module generation, floor planning, routing and layout generation.

There have been attempts to combine several of these subproblems and to solve them in a single step. However, no one has succeeded in and no one probably will ever succeed in solving all subproblems in a single step. Solving subproblem after subproblem does not guarantee an optimum or even an acceptable solution for the complete problem. Most people hope that this approach will work "on the average", but McFarland demonstrated in a key paper [McF87], that this does not hold for high-level silicon compilation.

1.1.2 Complexity

Most of the subproblems just mentioned can be proved to be NP-complete. Scheduling is an important example for this. Actually, many design systems currently focus on scheduling. Even with special heuristics, pruning etc. an enormous amount of scheduling decisions are taken by a typical synthesis system. The number of decisions is at least proportional to the length of the behavioural specification. On the other hand, the number of possible hardware designs for a given set of constraints is fixed. Therefore, for large behavioural specifications, many of the scheduling decisions are superfluous. We call scheduling decisions decisions of secondary importance. The scheduling problem is not really a subproblem of the original problem, it is only generated by a specific decomposition of the original problem. Accordingly, no one has actually proven that high-level synthesis is NP-complete. It has only been proven that the subproblems of a certain solution method are NP-complete.

It is worth mentioning that a considerable amount of computing time currently is spent in each phase predicting the effect of the current phase on the next phases. This time could be avoided by first enumerating all possible designs and then performing e.g. scheduling for a fixed hardware structure.

2 Attempts to Solve Current Problems

Below we describe techniques we are currently studying to overcome the above limitations.

2.1 Area-oriented Design Decisions

2.1.1 Floor-Planning

Almost every design decision has a potential to result in an unacceptable layout. Therefore, all relevant design decisions should be based on an preliminary floor-planning. Peng [Pen87] was the first to propose such an approach. His algorithm, however, is not general enough to handle real design problems. For example, backtracking is restricted in order to avoid infinite loops.

The floor-planner required for synthesis has to be tightly coupled to synthesis. It should be an incremental floor-planner, which is able to predict the effect of a small design change rapidly.

2.1.2 Busses

Many synthesis systems use multiplexers for selection of the appropriate data in a certain control step. It is well known that this is not area-efficient. There have been attempts to use busses [deM86]. However, current systems just minimize the number of busses. This number serves only as a very rough estimate of the required area. Floor-planning based design decisions are required for implementing area-efficient data selection.

2.2 Partial Structures and Binding Information

2.2.1 Motivation

With a design space as big as the design space for high-level synthesis and with the problems of predicting the precise effects of early design decisions on the final design, it is extremely important to reduce the design space as much as possible. Unreasonable designs should be excluded by the reduction of the design space. Only then will we be able to generate several reasonable designs and to fully explore the remaining design space.

Two means for the reduction of the design space can be identified: partial structures and binding information.

Partial structures are normally available for most design problems. Users of our own MIMOLA hardware design system frequently have a pretty good knowledge of some parts of the hardware structure. They want to express this knowledge in a partial description of the hardware structure and to use the synthesis system to design some additional circuitry like decoders, clocks and control. The existence of a partial designs should speed up the design process and should not slow it down. Partial designs were accepted in our first synthesis system, the MSS1 [Zim80]. This system, and specifically this feature, is used at Honeywell to design integrated circuits. Unfortunally, this feature has not been implemented in our second version [Mar86] of the system. Recently, we have redirected our attention to it [Bal89] [Bro89].

High-level synthesis has to generate a lot of bindings. The task for the synthesis system is simplified, if some of the bindings are generated by the user. This is only possible, if the input language allows the description of such bindings.

A third method for the reduction of the design space is the restriction to a special application area. This does not solve all the problems, though. For example, the Cathedral-II silicon compiler, which has been designed for signal processing applications [deM86], has to handle the same type of problems as any other architectural synthesis tool.

Below we discuss the use of partial structures and binding information during architectural synthesis. We will explain, how the bindings mentioned in the introduction can be conveniently expressed in a design language. Our own language MIMOLA will be used as an example.

2.2.2 Prebound Variables

Procedural descriptions of the required behaviour contain a set of abstract variables. It is not obvious, how these should be bound to hardware registers and RAMs. Optimizing storage allocation techniques have been used in optimizing compilers. There, the problem is known to be far from trivial. The problem is much harder in the synthesis area: the final structure is not yet known at storage allocation time.

In order to simplify the problem, several techniques have been used. Many synthesis systems identify each scalar variable with a register and each (one- or two-dimensional) array with a RAM. The result is a behavioural description at a low level, i.e. recursive procedures cannot be used unless the stack is explicitly modified. Furthermore, a large number of registers causes testing problems, requires bus minimization techniques and makes context switching hard to implement.

In order to simplify the storage allocation problem, MIMOLA allows the user to define variable to location bindings. We distinguish between user and temporary variables.

User variables can be bound in the variable declaration.

Example:

```
VAR
  counter : integer AT CounterReg;
```

Here, CounterReg is the name of a visible hardware component. In VHDL the same information could be provided by configurations. However, the information about types and bindings of variables would be scattered in the input text.

Variables, which have not been bound by the user, will be bound to one of the locations set aside for user variables. In MIMOLA (version 4.0), these locations are defined by a construct similar to VHDL configurations.

Example:

```
RESERVED locations IS
  FOR Variables USE Main[0..4095];
END;
```

Temporary variables are bound to one of the locations defined to be exclusively available for such variables:

Example:

```
RESERVED locations IS
  FOR Temporaries USE Reg[0..3];
END;
```

This binding information cannot change the semantics and hence it is compatible with the "correctness by construction" principle. If an insufficient number of locations is available, an error message can be easily generated. The additional information has a potential to speed up synthesis algorithms.

2.2.3 Manual Decomposition of Expressions

Manual decomposition of complex expressions and allocation of storage for intermediate values does not require special design language features. However, it obscures the procedural design specification and it is a possible source of errors. Hence, it should be avoided.

Note that all variables, by definition, are user variables if the user is responsible for expression decomposition.

2.2.4 Fixed Control Sequence

The scheduling phase of high-level synthesis can be omitted, if the mapping to control steps is already done by the user. This is actually required by some CAD-systems (e.g. [Anc87]). Some mechanism is required to express this mapping. The mechanism has to represent a program as as set of control steps, each of which potentially contains several parallel assignments. An elegant solution is a special block structure of the form

```
SEQBEGIN
  PARBEGIN
    set of statements
  PAREND;
  PARBEGIN
    set of statements
  PAREND;
SEQEND;
```

Each parallel block represents a control step and optionally contains ORIGIN-commands. SEQBEGIN ... SEQEND means that the sequence cannot be changed by any tool. In contrast BEGIN ... END means that any semantically equivalent partitioning into control steps is accepted. Alternatively, the distinction between the two types of blocks could be expressed by block attributes.

This step is error-prone and it should only be left to the user, when other solutions are too complex, too time-consuming or inefficient.

2.2.5 Preselection of Key Components

Frequently, the user has a pretty good knowledge about the type and number of required key hardware components. Key components can be defined to be those modules which perform some operation that is explicitly listed in the required system behaviour. Hence, key components include RAMs, registers, ALUs etc. Decoders, multiplexers and tristate-drivers do not belong to this category.

The following example presents a partial description of key components using MIMOLA:

Example:

```
PARTS                - - components
  alu : MODULE add_sub(IN a, b: word;
        IN c:twobits) RETURN f : word;
        BEHAVIOUR AtRtLevel IS
          BEGIN
            f <- CASE c OF
              0 : a + b;
              1 : a - b;
              2 : a OR b;
              3 : a AND b
            END;
          END;
  sadd: MODULE shft_add(IN a, b: word;
        IN c:twobits; OUT f : word)
        BEHAVIOUR AtRtLevel IS
          BEGIN
            f <- CASE c OF
              0 : SHIFTRL(a + b);
              1 : SHIFTLL(a);   - -shift left logical
              2 : a + b
              3 : a - b
            END;
          END;
```

It is relatively easy to use this information during scheduling, because now all the essential resources are known. Scheduling now reduces to the type of scheduling implemented in microcode generators.

Without this knowledge, the scheduling algorithm has to predict the result of each scheduling decision on module selection. This makes already complex algorithms, heuristically solving an NP-complete problem, even more time-consuming. Paulin [Pau87] has described a scheduling algorithm which tries to influence module selection by a proper look-ahead to that phase. The complexity of this algorithm demonstrates the advantages of knowing the key components in advance.

The use of this information does not violate the "correctness by construction" principle. An error message can easily be generated by the synthesis tool, if required hardware modules are missing. This information reduces the design space and can be easily used to speed up synthesis algorithms.

2.2.6 Prebound Operations

As a result of previous design iterations, users sometimes realize, that better designs would be generated, if the operation to hardware component binding would be changed.

This is only possible, if the input language allows the user to define such a binding. Most design languages are unable to express such a binding. Sometimes, limited ad hoc solutions like pragmas are used [deM86]. Normally, pragmas cannot be attached to the operations which are to be bound and therefore violate the principle of locality (that means, closely related information should not be scattered all over the input text).

All versions of MIMOLA provided constructs for this binding. For the sake of standardization, we are currently moving towards a straight-forward extension of attributes in VHDL.

Examples:

```
a := a +'alu b
a := a *'sadd 2
```

Attributes attached to operations indicate a binding of that operation to a hardware component. Inclusion of this feature in VHDL is subject to discussion [Har89].

This binding information cannot change the semantics and hence it is compatible with the "correctness by construction" principle. If a component does not provide the required function, an error message can be easily generated. The additional information has a potential to speed up synthesis algorithms.

2.2.7 Prebound Control Codes

ALUs normally provide a number of functions like addition, subtraction and all logical operations. Frequently it is desirable to make all these functions accessable from the controlling instruction word. The coding of the operations in the instruction could be completely different from that in the ALUs. However, a decoder (or separate control steps) can be saved, if the codes are the same.

Explicit references to hardware components are possible with almost all hardware design languages, including MIMOLA. This notation, however, is dangerous: the input to the synthesis tool is hard to understand and possibly incorrect. The step from the left hand side to the right hand side of the example requires verification and analysis of side effects. Our users, however, kept asking for this feature and an automatic transformation between the two descriptions is not trivial. Hence, explicit component activations are included in the MIMOLA language.

Example:

With decoder or separate control steps	Without decoder
CASE Instr.Opcode OF 0 : a := a - b; 1 : a := a OR b; 2 : a := a AND b; 3 : a := a + b; END;	a:= alu(a,b,Instr.Opcode)

2.2.8 Partial Description of the Interconnection Structure

Until now we have discussed information concerning components and the binding of operations to components. The next step is to include at least a partial description of the interconnection structure in the synthesis input. This information shall be used to evaluate the effects of design decisions.

A simple way of using this information is easy to implement: Many of the synthesis tools are not globally optimizing. They consider hardware requirements control step by control step. For each of the control steps the hardware is augmented, taking into consideration the hardware requirements by previous control steps. With this approach, it is relatively easy to take advantage of information about the interconnection structure. This information is presented to the synthesis tool as a partial structure generated by a virtual 0th control step. This is the concept employed in a new synthesis algorithm for the MIMOLA design system [Bal89].

There are problems, however, if this concept is combined with the usual simple way of matching system and component behaviour. In order to get a flavour of the problems, consider module sadd, which was described above. Assume that sadd consists of two local modules (c.f. fig. 1).

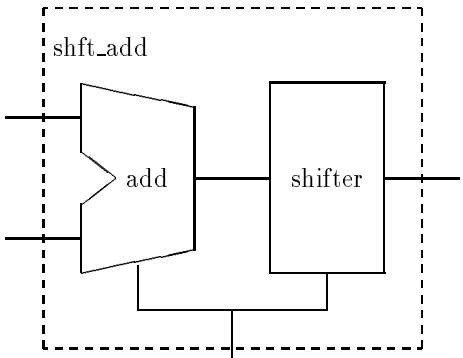

Figure 1: Prewired shift/add block

This component contains its own structure, even if modelled at the RT-level. These components will be called **composite** components. In contrast, components without a local RT-structure will be called **primitive** components. Problems with this type of components result from a common restriction in the algorithm which matches system and component behaviour.

In order to explain the problem, we shall describe the technique used in MSSH [Mar86]. The components described above are internally represented as a set of implications:

```
f=alu (a,b,0)   ⇒   f=(a + b)
f=alu (a,b,1)   ⇒   f=(a - b)
f=alu (a,b,2)   ⇒   f=(a OR b)
f=alu (a,b,3)   ⇒   f=(a AND b)
sadd(a,b,0,f)   ⇒   f=SHIFTRL(a + b)
sadd(a,b,1,f)   ⇒   f=SHIFTLL(a)
sadd(a,b,2,f)   ⇒   f=(a + b)
sadd(a,b,3,f)   ⇒   f=(a - b)
```

Components, which are called in the form of a procedure or a function call, are said to be **activated** by this call. Each argument position denotes a signal. In the final structure, these signals have to be implemented as interconnections. In the example above, the alu has been described like a function. Hence, the output signal is implicit. Component sadd has been described in the form of a procedure. In this case, the output signal f is included in the parameter list itself.

The behaviour at the right hand sides can be implemented by the component activations at the left hand sides. A separate analysis in MSSH guarantees that selected module activations do not have undesirable side effects (e.g. do not destroy register contents).

Left and right hand sides of these implications can be represented as trees. The leaves at the right hand side of these trees can be considered as free variables. In MSSH, trees of both sides have been restricted to a depth of two.

The specification of the system behaviour again is represented as a set of trees. The pattern matching process starts at the leaves of this specification. Each node of the system specification has to be bound to one hardware component (to be replaced by a module activation).

Example: Assume that the procedural behaviour contains an expression, specifying that the result of an addition is to be shifted to the right.

system specification:

in source format	in tree format
SHIFTRL(ac + 2)	SHIFTRL | + / \\ 2 READ,ac

Figure 2: Expression in source and tree formats

The matching process starts at the leaves (either "2" or "READ,ac") and binds hardware resources to the leaf operations. The process continues at the "+"-operation and tries to allocate a hardware component. Finally, the algorithm tries to allocated hardware to the shift operation. An error message is generated in case no component can be assigned to an operation.

This match is performed for each operation separately. In the particular example, the behaviour of the composite component shft_add is too complex to be matched with the system behaviour that way.

The expansion of composite components to their leaf cells would be one method of using these components with MSSH. However, MSSH would not be able to use the shifter as a via for the adder and vice versa. It would consider direct connections to the output of the adder and to the input of the shifter to be very costly but it would probably directly connect to and from both submodules. Hence, expansion it no solution to this problem.

2.2.9 Composite Components

The example demonstrates that it is necessary to drop the restriction that the component trees have a depth of two.

Larger trees at the left hand side are required whenever a set (i.e. a sequence) of components activations is required for implementing a certain operation.

Trees of arbitrary depth at the right hand side will allow us using components like functional registers (registers with clear, preset, shift or counting capabilities) and complex components like adders with built-in latches, AMD-type bitslices or even complete microprocessors. The behaviour of shft_add can be represented with a depth of three at the right hand side:

Example:

```
sadd(a,b,0,f)⇒ SHIFTRL
                  |
                  +
                 / \
                a   b
```

The matching process for complex components needs to be different: The matching process now starts at the root of the behaviour expressions and continues at the system nodes matching free variables of the component specification.

The mechanism for complex components relies on their representation by single trees on both sides of the implications. Very complex trees can be generated by transformation rules, e.g. by MIMOLA "REPLACEMENT"-rules.

A matching algorithm of this type has been implemented in MSSN. MSSN has been developed as part of a forthcoming master's thesis [Bro89]. MSSN uses floor-planning based design decisions as described above.

2.2.10 Full Structure Completely Fixed

As an extreme, the specification may contain a single block with a fixed structure. This is a limiting case of incremental synthesis: only the control code can be changed during synthesis. This special case occurs due to the following reasons:

The results of automatic synthesis systems normally are analysed by human designers. Frequently they discover possible optimizations, changes required to improve testability and changes required to meet some company standards. After such changes, guaranteed correctness normally is lost.

Tools, checking whether or not a manually modified hardware still meets the requirements, are needed in order to avoid this situation. In the case of an procedural specification this means: it has to be checked, that the specification can be compiled onto the changed structure. Our retargetable code generator MSSC handles this special case of incremental synthesis [Now89]. A typical design process including synthesis and generation of control code for a fixed structure is shown in fig. 3.

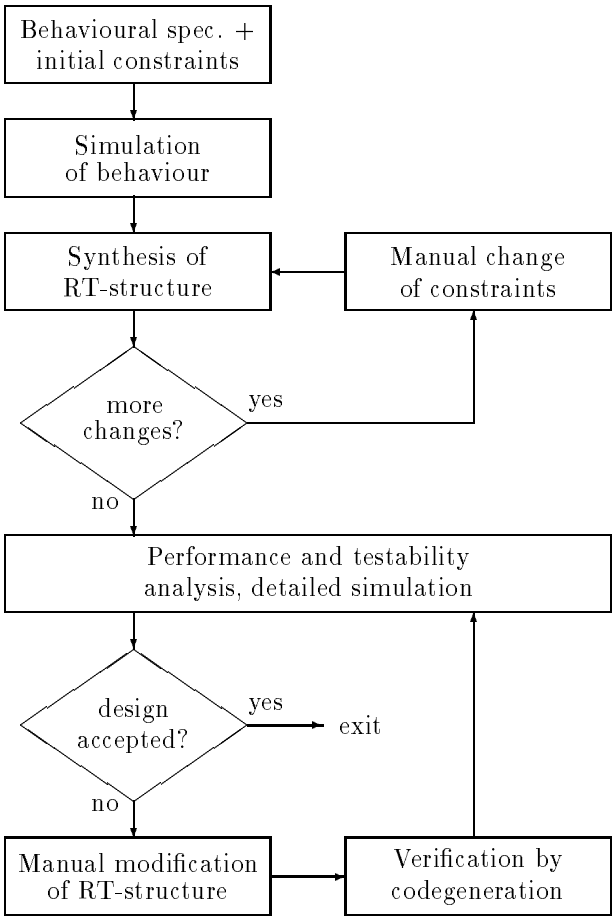

Figure 3: Control flow during a design project with MSS

2.3 Language Issues

The preceding sections clearly indicate a set of requirements for a design language supporting synthesis in general and the description of binding information in particular. Clearly, these requirements are not meet by typical languages designed for simulations. The lack of appropriate constructs in ISPS has been a handicap for synthesis applications for quite some time. To overcome this limitation, different languages were used to describe behaviour and (partial) structures. VHDL is an improvement in this respect. However, the inclusion of constructs equivalent to the ones mentioned above would help in synthesis applications.

2.4 Knowledge-Based Problem Representation

After having reduced the design space as much as possible, we should be able to explore the remaining design space in detail. Alternative solutions should be easy to generate. This is simplified, if backtracking is easy to implement.

Synthesis, code generation and test generation algorithms contain a large amount of unification. For example, the matching process between operations in the system specification and the operations in the component descriptions can be modeled as unification.

Heavy use of unification and backtracking are two reasons for knowledge-based approaches. A third reason is the need to avoid a fixed control flow during synthesis. If there is a large behavioural specification and only a small design space, possible hardware designs should be enumerated and scheduling should be done **after** it has been shown that the design fits onto the available silicon area. If the behavioural specification is small and the design space is large, scheduling should be done first.

An arbitrary sequence of synthesis steps is possible with a knowledge-based representation of the design problem. In such a context, design desions can be taken in an arbitrary sequence and an enumeration of possible designs is simple with PROLOG-like languages. Also, additional constraints, can be represented as facts. They reduce the search space and hence potentially reduce computing time.

Kowalski [Kow85] used knowledge-based techniques. He modeled the computer-aided synthesis process after the human design process. We believe that this is an unnecessary restriction.

3 Results

We do not yet have results from a synthesis system including all the proposed features. But we do have results for the features of sections 2.2.1 to 2.2.10, using MSSH as an example.

The following table contains results for the synthesis of a PDP-8. We use the number of wires and the size of the microprogram as rough estimates of the design complexity. Every new set of bindings was added to the previous set of bindings.

Manual binding	wires	microcode	micro-instructions
none	952	5886 bits	109
	=100%	=100%	=100%
expr. decomp.	100%	100%	100%
variables	93%	96%	102%
control steps	97%	82%	82%
key components	86%	81%	83%
operations	78%	72%	83%

This table indicates that the first five sets of bindings reduce the number of wires by 22% and the number of microcode-bits by 28%. The largest effect was caused by the manual binding of operations to components. The retargetable compiler was used to reduce the number of wires for two large design projects:

- The savings were 50% for our SAMP machine.
- Users, who have designed a reduction machine, reported savings of 24%.

We expect, that the results from the first algorithm [Bro89] combining all features, will be even better. The results may be different for other examples or synthesis tools. But we believe, that the effect of manual bindings would never vanish completely.

4 Conclusion

We have sketched the high-level synthesis process as the generation of a set of bindings. The ability to use the designers knowledge about efficient designs requires mechanism allowing the user to express such bindings. We propose that such mechanisms should be present in design languages supporting synthesis. Synthesis systems using such an enhanced language should have an improved performance in the sense of reduced computation time and better results.

References

[Anc87] F. Anceau: FORCE: A Formal Chcker for Executability, in: D. Borrione (ed.): From HDL Descriptions to Guaranteed Correct Circuit Designs, Proc. of IFIP WG 10.2 Working Conf., North Holland, 1987

[Bal89] M. Balakrishnan, P. Marwedel: Integrated Scheduling and Binding: A Synthesis Aproach for Design Space Exploration, 26th Design Automation Conference, 1989

[Bro89] O. Bross: Hardwaresynthese mittels Strukturfaltung, master's thesis, Institut für Informatik, University of Kiel, to be published

[deM86] H. De Man, J. Rabaey, P. Six : CATHEDRAL II: A Synthesis and Module Generation System for Multiprocessor Systems on a Chip, in: G. De Micheli, A. Sangiovanni-Vincentelli, P. Antognetti (ed.): Design Systems for VLSI Circuits, Logic Synthesis and Silicon Compilation, Martinus Nijhoff Publishers, 1987

[Har89] P. Harper, S. Krolikoski, O. Levia: Using VHDL as a Synthesis Language in the Honeywell VSYNTH System, 9th Int. Symposium on Computer Hardware Description Languages, 1989

[Kow85] T.J. Kowalski: An Artificial Intelligence Approach to VLSI Design, Kluwer Academic Publishers, 1985

[Mar86] P. Marwedel: An Algorithm for the Synthesis of Processor Structures from Behavioural Specifications, Microprocessing and Microprogramming, Vol. 18, 1986, pp. 251-262

[McF87] M.C. McFarland: Reevaluating the Design Space for Register-Transfer Synthesis, Proc. ICCAD, 1987

[Now89] L. Nowak, P. Marwedel: Verification of Hardware Descriptions by Retargetable Code Generation, 26th Design Automation Conference, 1989

[Pen87] Z. Peng: A Formal Methodology for Automated Synthesis of VLSI Systems, Ph.D. thesis, Department of Computer and Information Science, Linköping University, 1987

[Zim80] G. Zimmermann: MDS-The MIMOLA Design Method, Journal of Digital Systems, Vol.4, 1980, pp. 337-369