Synthesis for Control Dominated Circuits (A-22)
G. Saucier and J. Trilhe (Editors)
Elsevier Science Publishers B.V. (North-Holland)
© 1993 IFIP. All rights reserved.

Implementations of IF-statements in the TODOS microarchitecture synthesis system

Peter Marwedel

Informatik XII, University of Dortmund, P.O. Box 500 500, W-4600 Dortmund 50, Germany

Abstract

In microarchitecure synthesis, early algorithms considered only a single implementation technique for IF-statements. Focus was on scheduling and on maximum hardware sharing. In this paper, we present available options in more detail. They are described by using explicit program transformations. Some of these techniques have the potential to consider optimizations beyond the classical basic block boundary while maintaining the simplicity of basic-block oriented approaches.

Keyword Codes: B.1.1; B.5.2; B.7.2
Keywords: Control Structures and Microprogramming, Control Design Styles; Register-Transfer-Level Implementation, Design Aids; Integrated Circuits, Design Aids

1 INTRODUCTION

Microarchitecture synthesis is concerned with the generation of microarchitectures from behavioral specifications. Microarchitectures consist of building blocks such as registers, memories, arithmetic/logic units (ALUs) and multiplexers. Behavioral specifications normally use most of the language elements of traditional programming languages. One such element is the IF-statement. When implementing IF-statements with microarchitectures, we have to find proper ways of executing its operations. These consist of the controlling *condition* and the two *branches*. There are several implementation methods for IF-statements, each with different cost/performance tradeoffs. In this paper, we will demonstrate, how implementations can be selected automatically.

2 RELATED WORK

Traditional compilers usually compile IF-statements into code blocks containing conditional jump [1] instructions. If every IF-statement is implemented by a conditional jump, the resulting design may be quite slow. This becomes especially obvious, if delayed jumps are used. If n delay slots are used ($n = 0$ for undelayed jumps), $n + 2$ control steps are at least required for a single IF-statement.

In microarchitecture synthesis, many of the early algorithms took advantage of the fact, that THEN- and ELSE-branches are mutually exclusive and hence can share the same hardware resources (see e.g. [12, 10]). Sharing of hardware resources, however, requires the condition to be evaluated before the computation of the branches starts. This has to be guaranteed by scheduling algorithms. In [11], Paulin describes how this fits into the context of force-directed scheduling. This form of handling IF-statements is called *C-select* [5].

A second way of handling IF-statements during scheduling is *D-select* [5]. With D-select, the condition is not evaluated first. THEN- and ELSE-branches are computed concurrently without sharing hardware resources. Correct values are selected after both the branches and the condition have been evaluated. This approach leaves more freedom for scheduling the condition but requires more hardware. Branches generating anything else but mere values cannot be implemented this way.

A third way of handling IF-statements during scheduling is path-based scheduling [3]. Path-based scheduling generates schedules for each path through the data flow graph representation of the required behavior separately. This results in very fast schedules but potentially requires multiple control words for single unconditional operations. It is therefore targetted towards the synthesis of fast architectures from flow graphs with a small number of paths.

Early synthesis systems did not automatically select the "best" method for implementing IF-statements. For example, Fuhrmann [4] mentions missing support for D-select as a key problem in using a well-known high-level synthesis tool. An improvement in this respect was the work by Wakabayashi [14]. Wakabayashi's method allows automatic tradeoffs between C-select and D-select during scheduling. However, as Rim and Jain pointed out recently [13], it can produce incorrect results.

In their paper, Rim and Jain propose a new method for scheduling conditions and branches. This method allows arbitrary reordering of nested IF-statements. This method can be applied only if a) the specification language follows the *single assignment principle*, and b) no (and absolutely *no*) computation causes side-effects or exceptions like overflows, illegal memory accesses etc.). We believe that requirement b) is inadequate. Moving to system-level synthesis, languages like C will become candidates for specification languages. Compiler writers for these languages are aware of the fact that "optimizations" changing the order of computation may result in incorrect code.

[1] We avoid the term "conditional branch" because "branch" has a different meaning in this and other papers on the subject.

Another problem of existing approaches is the fact that they are putting emphasis just on scheduling. Certain schedules may have undesirable effects upon the required controller. For example, straightforward implementations of the D-select method require two separate control fields for all resources that are used differently in the two branches.

3 MOTIVATION OF OUR APPROACH

Our TODOS[2] synthesis algorithm uses a different approach. The order of computing conditions is never changed. Emphasis is not just on scheduling. *Resulting implementations are studied in more detail.* Moreover, we prefer to represent implementation techniques by *program transformations* for the following reasons:

- According to Rim and Jain [13], "there is still no consensus on the representation of conditional branches" in data flow graphs,
- Some conditional actions for D-select cannot be represented [13],
- Users of microarchitecture synthesis systems frequently ask for more control over the synthesis process. They want to analyse partial results rather that using a "turnkey system". In this respect, the use of text for specifications and of graphics for intermediate results is not ideal. The designer has to learn at least two design representation languages.

  Therefore, all our tools try to display partial results using the language in which the specification was written, namely MIMOLA. As long as possible, synthesis is modelled as the application of *program transformations*[3]. Required dependency relations, when needed, are computed on the fly.

4 IMPLEMENTATION METHODS FOR IF-STATEMENTS

In order to demonstrate our technique for program transformations we will use the following running example:

```
Lx: IF MAIN[5] > 0 THEN
     BEGIN
      MAIN[5] := MAIN[5] - MAIN[6];
Lx_a: PC := Lx
     END;
Ly
```

[2]TODOS stands for "TOp DOwn Synthesis". TODOS is an extension of the work described in [7] and in [6]. The name TODOS was chosen partially because the Spanish sentence 'Todos quieren TODOS' means 'Everybody loves (or desires) TODOS'.

[3]In practice, program transformations are applied to our internal representation of MIMOLA, called TREEMOLA. The mapping from TREEMOLA to MIMOLA is much easier than the one from flow graphs to textual programming languages.

In this example, MAIN is assumed to be the name of a large storage unit. 5 and 6 are addresses.

Generally speaking, implementation methods for IF-statements can be classified according to the affected component inputs. There are six categories, two for condition-controlled inputs of the controller's state register, three for condition-controlled inputs at other storage units, and one for condition-controlled combinational units:

1. *Conditional jump*

   In this case, the *data input of the state register* PC is affected. For example, the condition may control a multiplexer at this input (see fig. 1). As a result, the value loaded into the state register depends upon the condition, effectively implementing *conditional jumps.*

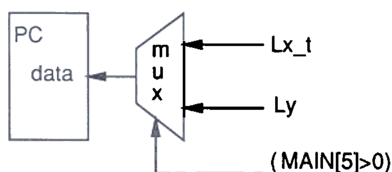

Figure 1: Hardware for conditional jumps

   This technique leads to a fork in the corresponding state chart. Using the technique, the above statement can be transformed into the following sequence:

```
Lx:    PC := (IF MAIN[5]>0 THEN Lx_t ELSE Ly);
Lx_t:  PARBEGIN
        MAIN[5] := MAIN[5] - MAIN[6];
        PC := Lx;
       PAREND;
Ly:
```

   Statements included in PARBEGIN .. PAREND are assumed to be executed in the same control step[4].

   This technique corresponds to C-select with the restriction that the computation of the condition and the computation of the branches are separated by at least one state transition. This restriction avoids long combinational delays and having to generate multiple control fields per hardware resource.

2. *Conditional continuation*

   In this case, the *control input of the state register* is disabled, as long as the condition evaluates to *false*. This technique requires a circuit like that of fig. 2.

   Depending upon the value of the condition, either the control code for loading the register or the control code for disabling the register is applied to its control input. In

[4]Generation of PARBEGIN .. PAREND is usually done during scheduling but is shown here to demonstrate the resulting number of control steps.

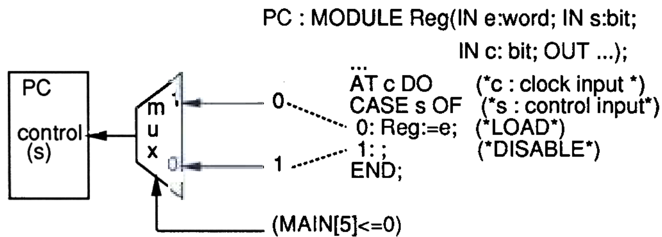

Figure 2: Hardware for conditional continuation

TODOS, the codes are assumed to be included in the description of the component behavior. For the control codes of fig. 2, the multiplexer may be replaced by a condition inverter.

This technique is very efficient for loops consisting of a single control step, i.e. for implementing multiplication in a microprogram. Using the technique, the assignment to `PC` can be transformed into a *guarded assignment* like in the following example::

```
Lx:  PARBEGIN
      /MAIN[5] ≤ 0/ PC:=Ly;
      /MAIN[5] >0 / MAIN[5] := MAIN[5] - MAIN[6];
     PAREND;
Ly:  ...
```

In this example, /< *condition* >/ denotes a guard. The left-hand variable in the following assignment will only be modified if *condition* evaluates to *true*.

In this and in the previous implementation techniques, the *state* stored in the controller is affected by conditions. We assume, that the controller is a Moore-type controller. This is not a restriction because the data-path is assumed to be a Mealy-type machine and hence repartitioning can always be applied to turn the controller into a Moore-type controller. As a result, the *output* of the controller cannot depend upon the value of conditions computed during the same control step.

Hence, all other implementation techniques assume that inputs of data-path components are affected by conditions.

3. *Conditional expression*

   In the first case, *data inputs of memories within the data path* are affected by conditions. For example, there may be multiplexers at the data input of the storage unit(s) (see fig. 3).

   This technique does not allow the sharing of hardware among the branches (except for common subexpressions) and corresponds to D-select.

   Using this method, we could transform the assignment to `MAIN` into the form shown in the following example:

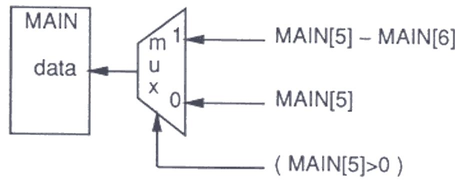

Figure 3: Hardware for conditional expression:

```
Lx:  PARBEGIN
      PC:=(IF MAIN[5] >0 THEN Lx ELSE Ly);
      MAIN[5]:= (IF MAIN[5]>0 THEN MAIN[5]-MAIN[6] ELSE MAIN[5]);
     PAREND;
```

A problem with this approach is that the conditional expression is always evaluated completely. Undesirable side-effects are therefore possible.

4. *Conditional assignment*

For the next method, condition(s) affect the *control input of memories within the data path.* In general, a multiplexer is required in order to disable or enable writing into the memory. The circuit at this input is similar to the one in fig. 2 (see fig. 4).

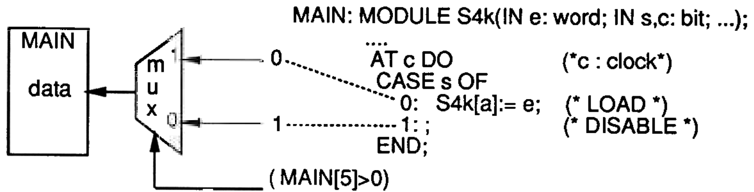

Figure 4: Hardware for conditional assignment

Using this method, we can transform the assignment to `MAIN` into the form that can be seen in the following example:

```
Lx:  PARBEGIN
      PC := (IF MAIN[5] >0 THEN Lx ELSE Ly);
      /MAIN[5]>0/ MAIN[5]:= MAIN[5]-MAIN[6];
     PAREND;
Ly:  ...
```

As in the last case, a problem with this approach is that the statement is always evaluated completely.

5. *Conditional address*

A multiplexer at the *address input of memories within the data path* may be used to write values into condition-dependent storage locations (see fig. 5).

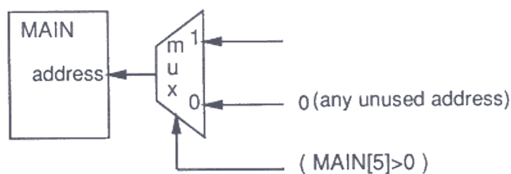

Figure 5: Hardware for conditional address

In this example, MAIN[0] is assumed to be some unused memory location.

This technique can be used to effectively suppress writing into MEM[5] if the condition is false.

Example:

```
Lx:  PARBEGIN
      PC := (IF MAIN[5] >0 THEN Lx ELSE Ly);
      MAIN[(IF MAIN[5] >0 THEN 5 ELSE 0)]:= MAIN[5] - MAIN[6];
     PAREND;
Ly:  ...
```

As in the last two cases, a problem with this approach is that the statement is always evaluated completely.

6. *Conditional operation of some combinational unit*

   Finally, *inputs of combinational units within the data path* may be affected by conditions. For example, an ALU may either add or subtract two values (this is required e.g. by Booth's multiplication algorithm) or there may be condition-controlled multiplexers at data inputs of combinational components.

   This technique allows sharing of resources among branches, but potentially requires multiple control fields per combinational unit.

   Example:

   For our running example, we could use the hardware structure of fig. 6.

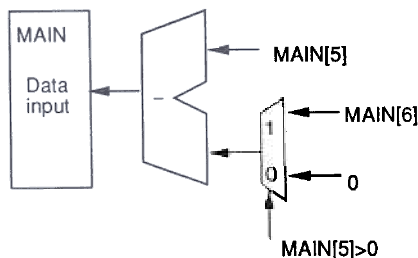

Figure 6: Combinational components affected by condition(s)

This hardware structure could be used by assignments of the following forms:

```
Lx:  PARBEGIN
      PC := (IF MAIN[5] >0 THEN Lx ELSE Ly);
      MAIN[5]:= MAIN[5] - (IF MAIN[5] >0 THEN MAIN[6] ELSE 0);
     PAREND;
Ly:  ...
```

5 IMPLEMENTATION IN TODOS

5.1 Supported implementation methods

TODOS has been designed to support implementation methods 1, 3 and 4 automatically. Methods 2, 5 and 6 can be applied only in very special cases. Using these methods in TODOS is possible, but requires *manual* program transformations. The MIMOLA language [1] is powerful enough to describe these implementation methods.

The preceding discussion does not mention nested IF-statements. Nested IF-statements could be handled by generating logical expressions containing all relevant conditions (see e.g. [9]). This approach was used in early versions of TODOS. It turned out that this resulted in very complex logical circuits, containing AND-gates, OR-gates, multiplexers and control signals (especially for long behavioral descriptions with a large set of different conditions). This approach was therefore rejected and replaced by the following: the methods mentioned above are applied to the innermost IF-statement level. All other levels use conditional jumps. By manually reducing the nesting level, complex logical expressions can still be generated.

Definitions:

1. A *section* is a portion of the behavioral description which, after the transformation of the innermost IF-statements into conditional assignments and conditional expressions satifies the conditions of basic blocks (control flow join at the beginning, control flow fork at the end).

2. The corresponding transformed portion of the behavioral description is called a *generalized basic block.*

In the following example, a *section* is printed in boldface:

```
i:=0; j:=0; k:=0;
REPEAT            (*join*)
 IF a[i]<b[j]
 THEN BEGIN c[k]:=a[i]; i:=i+1 END
 ELSE BEGIN c[k]:=b[j]; j:=j+1 END;
 k : = k+1;
UNTIL (i > imax) OR (j > jmax);
IF (i > imax) THEN ...
```

The innermost IF-statement can be replaced by a conditional expression for `c[k]` and conditional assignments to `i` and `j`. `UNTIL`-tests denote control flow and cannot be replaced by the above techniques. Hence, they represent the final fork of the control flow in the generalized basic block.

Like in the example above, generalized basic blocks are usually larger than standard basic blocks. Most of the flow analysis algorithms for basic blocks also apply to generalized basic blocks. Therefore, we are able to generate faster designs, we have a larger scope for optimizations and still maintain the simplicity of 'basic block'-oriented algorithms.

5.2 Generation of alternatives

Using the above definitions, TODOS partitions the behavioral description into sections. In general, a behavioral description consists of several such sections, because IF-statements may be nested and there may be jumps which cannot be transformed into conditional assignments (e.g. `UNTIL`-tests).

For each section, TODOS[5] generates up to three *alternatives* of implementations:

- One alternative is created with both the transformations to conditional assignments and to conditional expressions enabled. This alternative contains just a single generalized basic block.
- The second alternative is created with only the transformation to conditional expressions enabled. This transformation will remove the innermost IF-statements only if THEN- and ELSE-branch always contain assignments to the same variable. Otherwise, conditional jumps have to be added. Therefore, this alternative may contain several generalized basic blocks.

  This alternative exists only if there is at least one variable to which an assignment is made in both the THEN-branch and the ELSE-branch of an IF-statement.
- The third alternative uses only conditional jumps and therefore may contain more than one basic block.

In total, this means that the specification is partitioned into sections. For each section, there are up to three alternatives. Each alternative contains a set of generalized basic blocks, which, in turn, contain statements. The entire information is represented using our TREEMOLA intermediate format [2].

Sections and alternatives are generated before the kernel of the synthesis algorithm is started (see fig. 7).

Each of the sections is then processed by our synthesis system until a decision to use one of their alternatives is possible. Currently, the decision is taken after scheduling.

[5]The same alternative-generating mechanism is also used for other MIMOLA tools, such as the retargetable code generator.

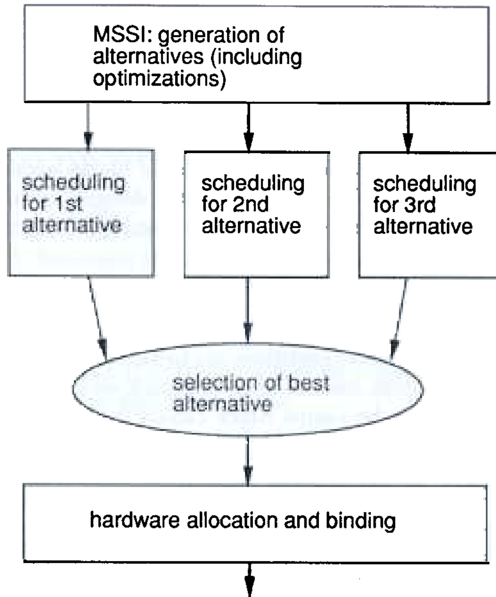

Figure 7: Handling of alternatives in TODOS

5.3 Scheduling

The scheduling algorithm works on one generalized basic block at a time. Note that such a generalized basic block contains at most one conditional jump-statement. The same scheduling algorithm can be used for all alternatives as long as some care is taken about special cases:

- *Conditional jumps*

  For conditional jumps, we just have to make sure that jumping is implemented in the last control step of an generalized basic block. We do this by simply defining a dependency relation such that:

$$\forall s \in S : CS(s) \leq CS(jump - statement)$$

  where:

S	:	statements in current generalized basic block
$CS(s)$	:	control step of statement s
$\leq$	:	order among control steps

- *Conditional expresssions*

No special care is required for this alternative, apart from the fact that jump-statements have to be handled like in the previous case.

- *Conditional assignment*

  A straight-forward, unoptimized scheduling for conditional assignments would also assign jumping to the last control step and would implement assignments within the THEN- and ELSE-branches as conditional assignments. It would therefore frequently create schedules like the following:

  Example:

  In the following example, assume that `assignment-1` is not part of the IF-statement and `assignment-2` to `assignment-m` are assignments included in the THEN-branch of the IF-statement. Straight-forward scheduling would create the following schedule:

```
CS1   assignment-1;
      ...
CSp   PARBEGIN
       /condition/ assignment-2;
       /condition/ assignment-3;
      PAREND;

CSn   PARBEGIN
       /condition/ assignment-m;
       PC := (IF condition THEN destin-1
             ELSE destin-2);
      PAREND;
```

The disadvantage of this schedule is that the condition is evaluated several times and that the execution time for the case when the condition is *false* is rather long. Furthermore, condition evaluations may result in different values unless the single assignment rule applies. We prefer to generate the following schedule:

```
CS1:  assignment-1;
       ...
CSp:  PARBEGIN
       PC := (IF condition THEN CSp+1
             ELSE destin-2);
       /condition/ assignment-2;
      PAREND;
CSp+1:PARBEGIN
       /condition/ assignment-3;
      PAREND;
```

```
CSn   PARBEGIN
       /condition/ assignment-m;
       PC := destin-1;
      PAREND;
```

Note that the guards in `CSp+1..CSn` are redundant. They are therefore eliminated by TODOS, resulting in reduced hardware requirements and/or a faster schedule. Note that the guard is required in control step `CSp`.

In oder to generate this schedule, we define the dependency relation such that:

$$CS(s) \leq CS(t) \Leftrightarrow \begin{array}{lcll} s: \text{normal assignment} & \wedge & t & \text{tagged assignment} \\ s: \text{normal assignment} & \wedge & t & \text{conditional jump} \\ s: \text{conditional jump} & \wedge & t & \text{tagged assignment} \\ s: \text{tagged assignment} & \wedge & t & \text{final unconditional jump} \end{array}$$

In this context, the term 'tagged assignment' denotes all conditional assignments with the same condition as in the conditional jump (there is at most one such jump per generalized basic block). All other statements (including other conditional assignments) are called 'normal assignments'.

5.4 Selection of an alternative

Selection of an alternative currently is based upon analysing the number of generated control steps and the number of dynamically executed control steps. This latter information is available since profile information can be added to MIMOLA behavioral descriptions. If one alternative is best with respect to both numbers, it is selected. Otherwise, the user is prompted to select between minimization of generated control steps and executed control steps. In future releases, other factors may be taken into account.

The essentials of the remaining synthesis steps have been described earlier. There is a strong focus on handling predefined library elements (see [8]).

6 RESULTS

We have analysed the effect of enabling conditional assignment and conditional expressions as implementation techniques. This was done using a MIMOLA version of the *mergesort* program from Wirth [15] as an example. As an indicator of the parallelism in the data path we used the number of storage unit ports. All other resources were unconstrained. Fig. 8 contains the results[6].

The effect of these additional implementation techniques increases with the parallelism in the data path (as expected). This effect would be even larger, if delayed jumps were used. These additional implementation techniques obviously have a potential to lead to

[6]These results were obtained with the last mainframe version (version 3.40) of TODOS.

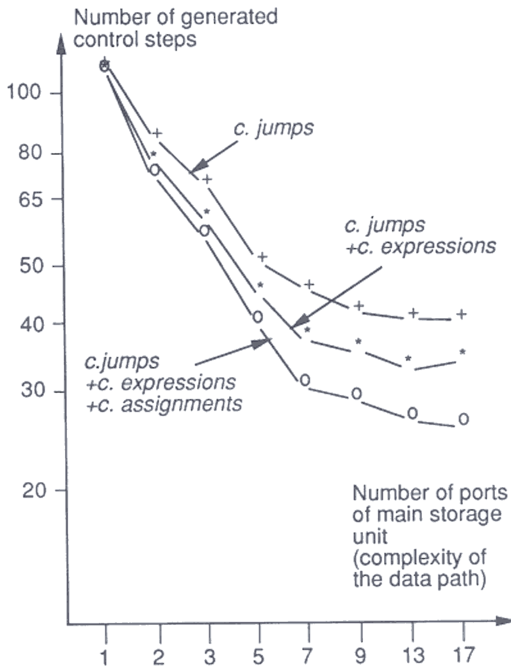

Figure 8: Influence of implementation techniques on # of control steps

faster designs. It should be mentioned, however, that long behavioral descriptions with many conditions and storage units frequently result in the generation of several complex condition multiplexers.

The current attempt to consider conditional assignments and conditional expressions is a compromise between ignoring these implementation techniques completely and having to handle alternatives in the very kernel of the synthesis system. The concept of alternatives has proved to be valuable. Future releases of our synthesis tools should explore this concept further and use them in a more general manner (i.e. could use alternatives in a more general way).

References

[1] R. Beckmann, P. Marwedel, D. Pusch, W. Schenk, and R. Jöhnk. The MIMOLA language reference manual - version 4.0 -, 2nd edition. *Report No.401, Computer Science Dpt., University of Dortmund*, 1992.

[2] R. Beckmann, D. Pusch, W. Schenk, and R. Jöhnk. The TREEMOLA language reference manual – version 4.0 –. Technical Report 391, Computer Science Dpt.,

University of Dortmund, 1991.

[3] R. Camposano, R.A. Bergamashi, C.E. Haynes, M. Payer, and S.M. Wu. The IBM high-level synthesis system. in: *R. Camposano and W. Wolf (ed.): High-Level VLSI Synthesis, Kluwer Academic Publishers*, pages 79–104, 1992.

[4] T.E. Fuhrmann. Industrial extensions to university high-level synthesis tools: Making it work in the real world. *Proceedings of the 28th Design Automation Conference*, pages 520–525, 1991.

[5] L. Hafer and A. C. Parker. A formal method for the specification, analysis and design of register-transfer level digital logic. *IEEE Trans. on Computer-Aided Design, Vol.2*, pages 4–18, 1983.

[6] P. Marwedel. An algorithm for the synthesis of processor structures from behavioural specifications. *Microprogramming and Microprocessing*, pages 251–261, 1986.

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

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

[9] S.-M. Moon, S. C. Carson, and A. K. Agrawala. Hardware-implementation of a general multi-way jump mechanism. *Proc. of the 23rd Ann. Workshop on Microprogramming and Microarchitecture (MICRO-23)*, pages 38–45, 1990.

[10] N. Park and A.C. Parker. Sehwa: A software package for the synthesis of pipelines from behavioral specifications. *IEEE Trans. on Computer-Aided Design*, 1988.

[11] P. Paulin. Global scheduling and allocation algorithms in the HAL system. in: *R. Camposano and W. Wolf (ed.): High-Level VLSI Synthesis, Kluwer Academic Publishers*, pages 255–282, 1991.

[12] M. Potkonjak and J. Rabaey. A scheduling and resource allocation algorithm for hierarchical signal flow graphs. *Proceedings of the 26th Design Automation Conference*, pages 7–11, 1989.

[13] M. Rim and R. Jain. Representing conditional branches for high-level synthesis applications. *Proceedings of the 29th Design Automation Conference*, pages 106–111, 1992.

[14] K. Wakabayashi and T. Yoshimura. A resource sharing and control synthesis method for conditional branches. *Int. Conf. on Computer-Aided Design (ICCAD)*, pages 62–65, 1989.

[15] N. Wirth. *Algorithmen und Datenstrukturen*. Teubner, 1975.