A new synthesis algorithm for the MIMOLA software system

Peter Marwedel

Institut für Informatik und Prakt.Math., University of Kiel
Olshausenstr. 40-60, D-2300 Kiel 1, W. Germany

Abstract
The MIMOLA software system is a system for the design of digital processors. The system includes subsystems for retargetable microcode generation, automatic generation of self-test programs and a synthesis subsystem. This paper describes the synthesis part of the system, which accepts a PASCAL-like, high-level program as specification and produces a register transfer structure. Because of the complexity of this design process, a set of subproblems is identified and algorithms for their solution are indicated. These algorithms include a flexible statement decomposition, statement scheduling, register assignment, module selection and optimizations of interconnections and instruction word length.

1. Introduction

Synthesis methods for the design of digital hardware have received a significant amount of attention, since these methods are capable of producing correct designs in a short turn-around time. Although some major contributions have been made in this area (e.g. [5,6,8,19,22,23]), there is still a lack of fast methods for the synthesis of hardware structures from high-level specifications. One of the reasons is that the design process consists of solving a large number of highly interdependent design problems, each being computationally complex.

By carefully partioning the design process into a sequence of subprocesses we have tried to reduce the complexity and to keep interactions between subprocesses as small as possible. Decisions are delayed until they cannot be postponed any longer. In one of the subprocesses, a decision is required, before its consequences are known. In this case, several possible solutions (**versions**) are handed over to the succeeding subprocesses until one of them is selected.

Algorithms for the subprocesses have been designed and implemented in our MIMOLA software system, version 2 (MSS2).

This research has been supported by the German Ministry of Research and Technology (BMFT) under contract NT 2816 9.

2. Global view of the MIMOLA software system

Work on the MIMOLA software system was initiated by G. Zimmermann in 1976. A first version of the design tools, called MSS1, was completed in 1979. As a result of the experiences with MSS1, work on an enhanced version, called MSS2, was started.

MSS2 presently supports 3 main applications (c.f. Fig. 1):

1. Synthesis of register transfer (RT-) structures from high-level PASCAL-like specifications.
2. Retargetable generation of (micro-) code for PASCAL-like programs and known RT-structures[14].
3. Generation of (micro-) diagnostics for known RT-structures.

At the RT-structure level, hardware is described in terms of registers, random access memories, ALUs and their interconnections. At the RT-behaviour level, the operation of hardware is specified in the form of assignment statements and interconnections are implicit [18].

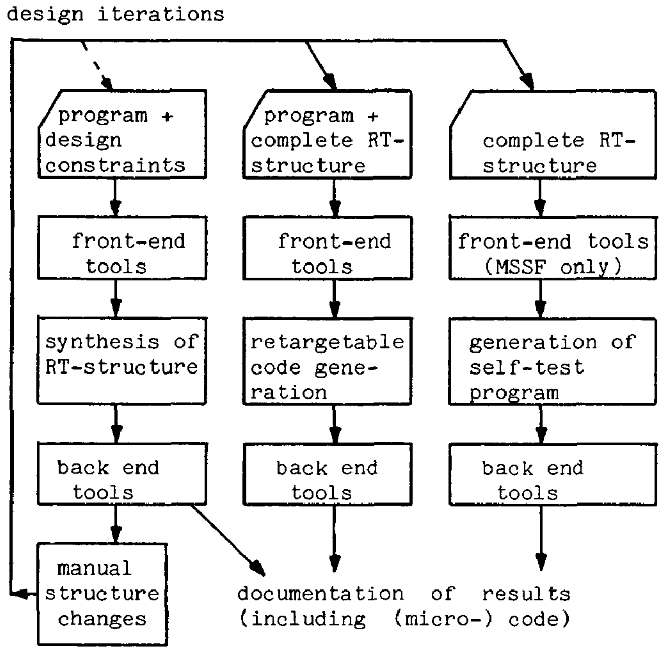
design iterations
program + design constraints
program + complete RT-structure
complete RT-structure
front-end tools
front-end tools
front-end tools (MSSF only)
synthesis of RT-structure
retargetable code generation
generation of self-test program
back end tools
back end tools
back end tools
manual structure changes
documentation of results (including (micro-) code)

Fig. 1 Global view of the MSS2

Previous papers described the motivation behind MSS2 and its general outline [13,15]. The aim of the present paper is to present details of the recently designed and implemented synthesis subsystem. A companion paper [9] demonstrates features of the test generation subsystem.

3. Synthesis with MSS2

3.1 Design specification

Design specifications for the synthesis subsystem consist of an algorithmic specification of the desired behaviour plus a set of design constraints.

The behaviour is described by a PASCAL-like program. The program may be either an interpreter for a given instruction set or an application program (e.g. a logic simulator). Programs may include high level language elements like recursive procedure calls, multi-dimensional arrays and PASCAL-like variables. There is no one-to-one correspondence between variables and registers.

Design constraints include limits for the number of immediate fields in the instruction, types of ALUs, and the type and number of available random access memories.

Details about the specification and its syntax have been included in previous papers [13,15].

3.2 Front-end tools

MSS2 consists of a number of independent PASCAL programs, called components. Communication between components is via intermediate files.

MIMOLA design specifications are translated into intermediate files by a component called MSSF. MSSF checks for conformance to the MIMOLA syntax and compile-time semantics.

MSSR is a component which maps high-level algorithmic programs to programs at the RT-behaviour level. One of the tasks of MSSR is to assign memory locations to variables. Variables can be bound to locations either manually or automatically. For both methods, static bindings (like in FORTRAN) or dynamic bindings (on a run-time stack) can be generated.

Example:

Let SM(i) denote location i of memory SM and let RP be the name of the program counter. Then, the program segment

```
IF a > 1 THEN a:= b - c; GOTO Lx FI
```

could be transformed by MSSR into

```
IF SM(1) > 1 THEN SM(1) := SM(2)-SM(3); RP:= Lx FI
```

In this case, static bindings (constant addresses) for variables a, b and c were assumed.

MSSR generates an RT-level behavioural description which still contains IF-statements. In addition to the usual implementation of IF-statements by conditional jumps and unconditional assignments, MSS2 provides for hardware-implemented conditional assignments and conditional expressions.

Examples:
The following forms are equivalent to the above example:

```
conditional jump:
     RP:= (IF SM(1) > 1 THEN L1 ELSE L2 FI);
 L1: SM(1):= SM(2) - SM(3); RP:=Lx;
 L2: ...

conditional assignment:
(/../ corresponds to CDL's label)
     /SM(1) > 1/    SM(1):= SM(2) - SM(3);
     RP:=(IF SM(1) >1 THEN Lx ELSE L2 FI);
 L2: ...
```

Sequential execution is necessary for the first form and an implementation requires at least two (micro-) instructions.

In contrast, both assignments in the second form can be done in parallel. Therefore, it can be implemented by a single (micro-) instruction if a sufficient amount of hardware is available.

It is hard to anticipate, which implementation will be the fastest, if only a limited number of hardware resources is allowed. Therefore the design decision is delayed by generating up to three different **versions** of control flow implementations in a component called MSSI. One of these versions is selected after the number of required instruction steps has been computed for each version.

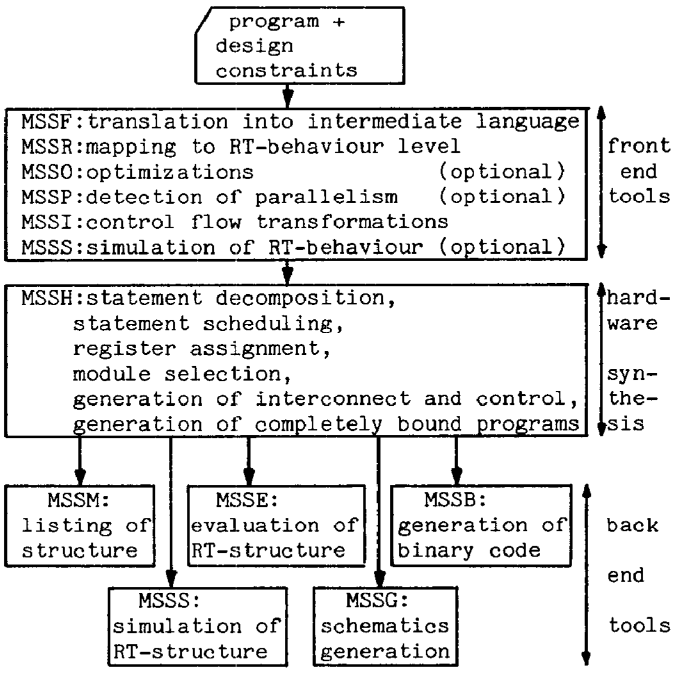
program + design constraints
MSSF:translation into intermediate language
MSSR:mapping to RT-behaviour level
MSSO:optimizations (optional)
MSSP:detection of parallelism (optional)
MSSI:control flow transformations
MSSS:simulation of RT-behaviour (optional)
front end tools
MSSH:statement decomposition, statement scheduling, register assignment, module selection, generation of interconnect and control, generation of completely bound programs
hardware synthesis
MSSM: listing of structure
MSSE: evaluation of RT-structure
MSSB: generation of binary code
MSSS: simulation of RT-structure
MSSG: schematics generation
back end tools

Fig. 2 Steps in the synthesis of RT-structures

MSSF, MSSR and MSSI are three of the so-called front-end tools. The execution of these tools precedes the execution of the synthesis algorithm (see Fig. 2). Other front-end tools are MSSS (a simulator capable of simulating RT-behaviour), MSSO (an optimizer for RT programs) and MSSP (a component detecting possible parallelism).

3.3 The synthesis subsystem

3.3.1 Statement decomposition

The synthesis system uses instruction bits in order to generate (address- and data-) constants. Design constraints may include a maximum for the number of immediate bits per instruction. Hence, complex statements, containing many constants, must be decomposed into a sequence of simpler statements not violating these design constraints. Necessary temporary variables must be introduced.

For the present version of the MIMOLA system it is also assumed that there is no reassignment of hardware resources during the execution of a generated instruction. As a consequence, e.g. the number of memory references per instruction cannot exceed the number of memory ports (available memories are described as part of the design constraints). Therefore, statements containing many memory references must also be decomposed into simpler statements.

Other design constraints include a maximum for the number of ALUs to be generated. Hence, the maximum number of arithmetic operations per instruction also is restricted.

Finding an optimal decomposition is known to be NP-complete. Traditional compiler techniques like [21] are optimal only for special cases. One of the frequent simplifications is ignoring the existence of common subexpressions. Our previous experience however indicates that taking advantage of common subexpressions is required for acceptable designs. Optimal algorithms, which do consider common subexpressions (e.g.[20]), do not handle general expressions.

We therefore developed a heuristic method. The virtue of this method is that it is very flexible with respect to different design constraints and that it takes advantage of common subexpressions.

Let t be an arbitrary expression or assignment. Define **treetoobig** (t) such that **treetoobig**(t) is **true** if t cannot be evaluated in a single cycle and **false** otherwise. The precise definition of **treetoobig** includes the number of available memory ports, the upper limit for the total instruction length and predictions of the cost to implement arithmetic operations present in t. For example, **treetoobig** is true, if the number of memory references in t exceeds the number of available memory ports.

Let t again denote an arbitrary expression or assignment. Define **mostcomplex**(t) to mean a subexpression e of t, where e is by some heuristic criteria the most complex subexpression of t, which can be assigned to a temporary variable without violating design constraints. In MSS2, the number of memory references is the most important criterion for the selection of e.

Using **treetoobig** and **mostcomplex**, statements are decomposed by the following procedure:

```
PROCEDURE decompose(s);
 BEGIN
  FOR ALL subexpressions t of statement s,
          starting with the leaves DO
    WHILE treetoobig(t) DO
      e := mostcomplex(t);
      generate assignment of e to temporary variable;
      push assignment onto the top of a stack;
      replace e by a read operation of the temporary
      variable; replace all subexpressions of the
      current (parallel) block being equal to e by a
      read operation of the temporary variable;
    OD; OD;
  push statement s onto the top of a stack;
 END;
```

Example:
Consider one of the assignments shown in section 3.2:

SM(1) := SM(2) - SM(3)

Let SM be a memory with a single port and let SR be a (small) multiport memory. Then **decompose** will deposit the following sequence of statements in the stack:

contents of stack	pushed when t is equal to
SR(v1):=SM(2);	SM(2) - SM(3)
SR(v2):=SR(v1)-SM(3);	SM(1):= SR(v1) - SM(3)
SM(1) :=SR(v2)	(by final push (s))

SR(v1) and SR(v2) are temporary variables. In order to simplify the following steps, there is only a single assignment to each of the temporary variables. Although **decompose** assigns a memory (SR) to temporary variables, it leaves their addresses (v1 and v2) unspecified.

3.3.2 Statement scheduling

In order to allow the design of fast parallel machines of the horizontally microprogrammed type, the MSS2 tries to schedule several assignments for parallel execution. That is, MSS2 tries to pack several statements into a single instruction, thereby creating **parallel (micro-) instructions**. Parallel execution of statements is allowed as long as **treetoobig** remains **false** for the parallel instruction.

Example:
The two assignments

SM(1) := SR(v2)   and   RP := Lx

can be compacted into a single instruction, if the number of memory ports is the only design restriction.

Scheduling statements for parallel execution is also known as **microcode compaction**. Several algorithms for microcode compaction have been published (see e.g.[12]).

For the MSS2, we modified the pairwise comparison algorithm, which was first proposed by Dasgupta and Tartar [3]. Necessary modifications include the following:

1. In the Dasgupta/Tartar algorithm, assignments to temporary variables (e.g. SR(v2):=..) are placed, before their references (e.g. SH(1):=SR(v2)) are considered for compaction. As shown in [16], backtracking may become necessary, if a cyclic data dependence exists. Such a cyclic data dependence may occur in a language like MIMOLA, which allows parallel blocks like **parbegin** a:=b, b:=a **parend** (this parallel block denotes a swapping of variables). Backtracking can be avoided, if statements are considered for compaction in the reverse order, that is: assignments to temporaries are considered for compaction only after all references to them have already been placed. It is for this reason, that **decompose** deposits statements on a stack. Compaction starts with statements at the top of the stack (with SM(1):= SR(v2) in the last example).

2. Usually compaction algorithms assume that temporary variables have been bound to memory locations before compaction starts. This may result in an unnecessary data dependence between two variables, which have been assigned to the same location (to be more precise, this may result in an **anti**-data dependence [17]). Therefore the decomposition procedure assigns only a certain memory to each of the temporary variables and delays the assignment of locations within that memory.

As the name indicates, the pairwise comparison algorithm compares statements pairwise for data-dependence and resource constraint violations. This comparison is limited to statements contained in the same block. Hence, the complexity of the pairwise comparison algorithm grows quadratically with respect to the size of blocks and linearly with respect to the number of blocks. This complexity is equal to that of the statement decomposition phase, because **decompose** requires that common subexpressions within a block are detected. Detecting common subexpressions also requires a pairwise comparison of expressions.

At the end of the scheduling phase, the behaviour of the RT-program has been decomposed into the behaviour of each of the instructions. The number of instructions for every version generated by MSSI therefore is known and the shortest instruction sequence can be selected.

3.3.3 Register assignment

After all statements have been assigned to one of the instructions, locations are assigned to temporary variables. Since optimizations at this step are limited to straight-line sequences of instructions, this step is almost trivial:

```
Mark all locations being available for temporaries
as deallocated.
FOR ALL instructions i in the present sequence DO
  if i contains the last reference of some
   temporary variable then deallocate the location
   used by this variable;
  if i contains an assignment to a temporary
   variable then find an unallocated location and
   allocate it.
 OD;
```

Example:

Consider the sequence listed in section 3.3.1:

```
SR(v1) := SM(2);
SR(v2) := SR(v1) - SM(3);
SM(1)  := SR(v2);
```

Scanning this sequence from the top to the bottom, we will allocate the same physical location to both SR(v1) and SR(v2).

For a given sequence of parallel instructions, this algorithm uses only the minimum number of required locations. If one would change the sequence after allocating temporary locations, this feature would be lost. During the scheduling phase the sequence of statements is frequently changed. Hence, too many locations would be required, if the allocation would already be done in the statement decomposition phase.

3.3.4 Module selection

The previous design steps did not synthesize an RT-structure. They just transformed the program such that the selection of hardware resources is simplified. The next design step now is the first of those steps which actually build up an RT-structure.

As a result of the scheduling phase, arithmetic and logic operations in each of the instructions are known. We now use this knowledge in order to generate arithmetic/logic units (ALUs).

There are basically three methods for the generation of ALUs by a synthesis system:

1. Available functional modules are **completely** specified in the design specification [6].
2. Based upon information about concurrently executed operations, **new** ALUs are designed by the synthesis system [22, 23].
3. The design specification includes **types** of predesigned modules. The synthesis system then selects an appropriate number of incarnations of these modules.

All three methods may be used in a single synthesis system. Our present system, however, concentrates on the last method. This last method is required for a standard-cell silicon compiler.

It is assumed that for each module type m, there is an associated cost c_m. The task then is to select an appropriate number x_m of incarnations of each type such that there is a sufficient amount of hardware for every instruction and such that total cost $c = \text{sum}_m(x_m * c_m)$ is minimal.

Let $f_{i,j}$ be the number of operators of type j being used in instruction i. Let $F_i = \{j \mid f_{i,j} > 0\}$ be the set of operators used in instruction i. Let F_i^* be the powerset of F_i, that is, the set of all subsets of F_i.

Then, a sufficient and necessary condition for a sufficient number of incarnations is that:

$$\forall i, \forall g \in F_i^*: \text{sum}_m (x_m) \geq \text{sum}_{j \in g} (f_{i,j}), \qquad (1)$$

where the sum over m is taken over those ALU types, which are able to perform some operation $j \in g$.

Let $b_g = \max_i (\text{sum}_{j \in g} f_{i,j})$.

Let $F^* = \bigcup_i F_i^*$ be the union of the F_i^*'s and let $a_{g,m}$ be 1 if module type m is able to perform some operation $j \in g$ and 0 otherwise.

Then, from (1) it follows that

$$\forall g \in F^* : \text{sum}_{m \in M} (a_{g,m} * x_m) \geq b_g \qquad (2)$$

The selection task therefore reduces to minimizing $c = \text{sum} (x_m * c_m)$ subject to the set (2) of constraints. This is a classical integer programming (IP) problem.

The virtue of our module selection method lays in the fact that it combines global optimization with a low algorithmic complexity. The number of integer variables is equal to the number of module types. The number of relations typically grows sublinearly with respect to the length of source program. This behaviour can be demonstrated by the folowing example:

Example:
Assume, there are two instructions. In one of them there are two occurences of operation type "+" and one occurence of operation type "-". In the other, there are two occurences of operation type "-" and one occurence of operation type "+". The powersets for both instructions are **identical** and equal to the union of the powersets:

$$F^* = \{ \{"+"\}, \{"-"\}, \{"+", "-"\} \}.$$

Therefore there are three algebraic relations:

1. The number of ALUs being able to add is $\geq$ 2.
2. The number of ALUs being able to subtract is $\geq$2.
3. The number of ALUs being able perform either operation is $\geq$ 3.

The following table contains actual numbers of relations, variables and CPU-times for the GOMORY I IP-algorithm [10].

program	kernel of a parser	kernel of an expert system	logic simulator
# lines	562	1330	430
# relations	20	33	5
# variables	11	11	7
CPU-time [ms] (1 Mips)	35	30	33

The worst case number of relations is an exponential function of the number of operation types in the source language (and independent of the size of the program). The only way to create a large set of relations is to generate instructions with a large number of different operation types. But even if 7 or 8 different operation types were present in a single instruction, the IP-problem would be managable because the structure of the relations is such that only few iterations are required.

Integer programming has already been proposed as a solution to the module selection problem. In [7] it is described as a method to select logic gates. At the gate level, a large number of binary decision variables has to be used to model the fact that there are various ways to implement simple logic operations. This large number seemingly has prohibited using this method. At the RT-level, there is essentially but one way to implement "+" or "-" (Leive [11] focusses on the aspect of having multiple choices to implement an operation). Hafer [5] used mixed integer linear programming to select ALUs. In Hafers approach, module selection is included in a large set of relations. Therefore it became impossible to solve large design problems in reasonable time.

At the end of this design step, all major hardware components have been selected. However, behavioural level operations have not yet been bound to specific hardware modules.

3.3.5 Generating interconnect

Allocating hardware modules to behavioural level operations implies the existence of physical paths from source modules to sink modules. The problem is to find assignments of modules to operations such that the cost for interconnect is minimal. Unfortunately we are unable to predict the effect of such an assignment in terms of wiring area. We therefore use a simplified design objective: minimize the total number of paths!

The optimization problem is formulated as follows: For each operation to be performed by one of the instructions, there is a set of matching hardware resources. E.g. for each arithmetic operation, there is a set of functional modules, which are able to perform this operation and for each constant (0-ary operation), there is a set of instruction fields of the required length. Now, for each operation find a resource from this set such that no resource is assigned to more than one operation per instruction and such that the minimal number of paths between resources is required.

In the present implementation of the MSS2, a branch-and-bound algorithm is used to solve this assignment problem. Unfortunately the complexity of this algorithm makes it impossible to generate globally optimal assignments. Therefore, it is necessary to solve the assignment problem for one instruction at a time, starting with the most complex instruction.

In the case the above algorithm computes a solution requiring more than a single path to an input, multiplexers are generated by the MSS2 in a straight-forward manner.

3.3.6 Generating control

In the MSS2 we assume that the hardware is controlled by instructions with a format similar to horizontal microinstructions. More specifically, we assume that the **direct encoding** method [1] is used to control RT-modules: for each module with a control input, there is a corresponding instruction field f_i, which may be used to select one of the modules $_i$operations. Since not all the modules are used simultaneously, some of them may **share** instruction bits.

Let l_i denote the length of field f_i in bits and let $c_{i,j}$=**true** denote that fields f_i and f_j are used concurrently in at least one of the generated instructions. The task then is to find a mapping from f_i to the set of instruction bits, such that:

1. No two fields f_i and f_j, for which $c_{i,j}$ is **true**, share an instruction bit and
2. the total number of instruction bits is minimal.

This problem is equivalent to scheduling a number of tasks $\{f_i\}$, each with execution time l_i and with resource conflicts $c_{i,j}$ such that the completion time is minimized.

This problem is one of the hard scheduling problems. The solution used in the MSS2 is one of the common scheduling policies: shedule long fields and fields with many conflicts first.

Most of the previously published methods for word length minimizations [4] are not applicable to the direct encoding model, because they use Wilkes original microinstruction model (which Dasgupta [4] calls **direct control**). In Wilkes model, there are no multi-functional modules like ALUs.

The only optimization method for direct control the author of this paper is aware of, is the use of cliques [23], which cannot be easily applied to fields of different length.

The instructions, which are generated by our system could be called "microinstructions", because they are parallel instructions and because they are directly interpreted by the hardware. However, these "microinstructions" are not necessarily interpreting "machine instructions". This will only be the case if the specification contains an instruction set interpreter.

3.3.7 Generating completely bound programs

Using the results of sections 3.3.5 and 3.3.6, so-called **completely bound** programs can be generated. Completely bound programs explicitly specify all used hardware resources and all used instruction bits [13]. Completely bound programs may be processed by the back-end tools MSSM, MSSS, MSSB, MSSE and MSSG (c.f. Fig. 2).

MSSM generates a MIMOLA description of the resulting RT-structure and of the bound program. This is possible, because MIMOLA can be used to describe **structure and behaviour** (the companion paper [9] contains an example of a structural hardware description in MIMOLA). This unique feature of MIMOLA simplifies supporting design iterations.

MSSS is a simulator capable of simulating RT-structures. MSSB generates listings of binary instruction code. MSSE evaluates RT-structures. MSSG is currently being implemented. Its purpose is to generate a graphic description of the RT- structure (schematics).It is modelled after an algorithm developed at the IIT in Delhi [2].

4. First results

The algorithm just described performs significantly better than the one which has been used in MSS1.

A processor, which had been designed with MSS1, has been partly redesigned with MSS2. The number of interconnections, which had been produced by MSS1, has been manually reduced by about 50%. The design produced by MSS2 contains the same number of interconnections as the manually optimized design.

The reduction of the number of generated connections has been made possible by a more global (although not yet completely global) optimization during the allocation of hardware resources to operations. This in one of the achievements in MSS2.

MSS1 and MSS2 also differ in the way they handle control lines. In the MSS1, some of the control lines were not explicitly represented. This flaw made it difficult to handle all instruction fields in a uniform manner. One of the design principles of the MSS2 therefore was to avoid any implied hardware structures and make all design decisions explicit. This approach made it possible to modularize the design system (the MSS1 basically consisted of a single, large program).

Because of the full inclusion of the control section, it was possible to quantify the effect resulting from conditional assignments and conditional expressions if the number of memory ports is large (Fig. 3):

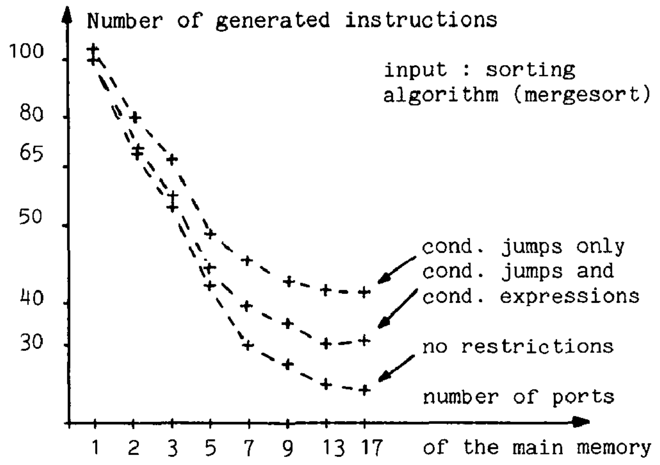

Fig.3 Effect of different implementations of IF-statements

Using a sorting algorithm as input, we designed a processor being about as complex but about twice as fast as our SIEMENS 7.760 instruction set processor (about 1 MIPS). Although the MSS2 design contains only a minimum amount of instruction decoding, the size of the code for the MSS2 design is about the same as the size of the code generated by the SIEMENS PASCAL compiler.

5. Conclusion

The task of generating structural descriptions for high-level behavioural specifications is a complex task, which has to be decomposed into a number of subtasks in order to achieve acceptable design times. This paper introduces such a decomposition and identifies associated subtasks. For each of the subtasks, at least basic ideas for possible solutions are included in the paper. The complexity of the algorithms presented typically grows quadratically with respect to the size of the blocks and linearly with respect to the number of blocks in the design specification. Therefore this method can be applied to large design specifications. Although we are not yet able to globally optimize the interconnection structure, we were able to significantly improve the results obtained with an earlier design system.

6. References

[1] A.K. Agrawala and T.G. Rauscher: Foundations of Microprogramming,Acadamic Press, New York, 1976

[2] A. Arya, A. Kumar, V.V. Swaminathan and A. Misra: Automatic Generation of Digital System Schematics, 22nd Design Automation Conf., 1985, p. 388-395

[3] S. Dasgupta and J. Tartar : The Identification of Maximal Parallelism in Straight - Line Microprograms, IEEE Trans. Comp., Vol. C-25, 10(1976), p. 986-992

[4] S. Dasgupta : Some Aspects of High - Level Microprogramming, Computing Surveys, Vol. 11, 3(1980), p. 295-323

[5] L. Hafer and A. Parker : A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic, IEEE Trans. on Computer-Aided Design, Vol. CAD-2, 1(1983), p.4-18

[6] C.-L. Huang: Computer-Aided Logic Synthesis Based on a New Multi-Level Hardware Design Language -- LALSD II, PhD thesis, State University of New York at Binghamton, 1981

[7] U.R. Kodres : Partioning and Card Selection, in: M.A. Breuer (ed.) : Design Automation of Digital Systems,Vol.1, Prentice Hall, Englewood Cliffs, 1972

[8] T.J. Kowalski and D.E. Thomas: The VLSI Design Automation Assistant : Prototype System, 20th Design Automation Conf., 1983, p. 479-483

[9] G. Krüger : Automatic Generation of Self-Test Programs - A New Feature of the MIMOLA Design System, these proceedings

[10] H. Langmaack: Gomory I, Collected Algorithms of the ACM, Algorithm 263A, 1978

[11] G.W. Leive : The Design, Implementation and Analysis of an Automated Logic Synthesis and Module Selection System, PhD thesis, Carnegie-Mellon University, Pittsburgh, 1981

[12] P.W. Mallett : Methods of Compacting Microprograms, PhD thesis, University of Southwestern Louisiana, Lafayette, 1978

[13] P. Marwedel: The MIMOLA Design System: Tools for the Design of Digital Processors, 21st Design Automation Conference 1984, p. 587-593

[14] P. Marwedel : A Retargetable Compiler for a High-Level Microprogramming Language, 17th Annual Microprogramming Workshop (MICRO-17), 1984, p. 267-276

[15] P. Marwedel : The MIMOLA Design System : A Design System Which Spans Several Levels, in: W.K. Giloi and B.D. Shriver (ed.): Methodologies for Computer System Design, North Holland, 1985, p. 223-237

[16] P. Marwedel: Ein Software-System zur Synthese von Rechnerstrukturen und zur Erzeugung von Mikrocode, habilitation thesis, University of Kiel, Germany, submitted sept. 1985

[17] D.A. Padua, D.J. Kuck and D.H. Lawrie : High-Speed Multiprocessors and Compilation Techniques, IEEE Trans. Comp.,Vol. C-29, 9(1980), p. 763-776

[18] A.C. Parker : Automated Synthesis of Digital Systems, IEEE Design and Test of Computers, Vol.1, 4(1984), p. 75-81

[19] A.C. Parker, F. Kurdahl and M. Mlinar : A General Methodology for Synthesis and Verification of Register-Transfer Designs, 21th Design Automation Conf., 1984, p. 329-335

[20] B. Prabhala and R. Sethi : Efficient Computation of Expressions with Common Subexpressions, Journal of the ACM, Vol. 27, 1(1980), p. 146-163

[21] R. Sethi and J.D. Ullman : The Generation of Optimal Code for Arithmetic Expressions, Journal of the ACM, Vol. 17, 4(1970), p. 715-728

[22] S. Takagi: Rule Based Synthesis, Verification and Compensation of Data Paths, Int. Conf. on Computer Aided Design (ICCAD) 1984, p.133-138

[23] C.-J. Tseng and D.P. Siewiorek : Facet : A Procedure for the Automated Synthesis of Digital Systems, 20th Design Automation Conf., 1983, p. 490-496