North-Holland
Microprocessing and Microprogramming 18 119861 251-262

An Algorithm for the Synthesis of Processor Structures from Behavioural Specifications

Peter Marwedel

Institut fur Informatik and Praktische Mathematik
Olshausenstr. 40-60, D-2300 Kiel. W. Germany

This paper describes a method for the automatic generation of the internal structure of digital processors from a specification of the required behaviour. The latter is specified by a high-level, PASCAL-like program. The internal structure is described in terms of memories, arithmetic/logic function boxes, multiplexers and their interconnections. In order to reduce the complexity of the design process, it is,partitioned into a sequence of individual steps. These steps include a flexible expression decomposition, a statement scheduling phase, anew module selection method and optimizations of interconnections and instruction word length.

1. Design Specification

The manual design of digital systems is a timely and errorprone process. While the complexity of digital systems is still increasing, there is a demand for shorter design times. Cutting design times can only be achieved by the use of new design techniques. Therefore, there is a growing interest in methods for the synthesis of digital processors from a description of the required behaviour.

There are several levels of abstraction on which the required behaviour can be specified. In order to open a large design space, the behaviour should be specified on a level as high as possible.

A common form of design specifications for digital processors is the description of an instruction set, which is to be implemented.

However, there are cases, where such a specification would restrict the design space more than necessary. As an example, consider the design of a processor for some dedicated application, like the design of a simulation engine. The only behaviour of such a processor, which can be monitored from the outside world, is the behaviour of the programs running on it. It does not matter, which instruction set the machine is executing. Therefore, programs can be used to specify the desired behaviour of digital processors. This approach was first proposed by G. Zimmermann [1]. A design specification describing the desired behaviour by programs is called a specification on the algorithmic level. Starting on the algorithmic level allows the design procedure to tailor the instruction set to the particular application. The instruction set is generated during the design and there is no need to specify it from the beginning. The instruction set may be constructed such that using the poten

tial parallelism of VLSI technology is simplified.

Frequently, however, designers will still have to design machines with a given instruction set. Fortunately enough, this turns out to be a special case of an algorithmic specification. Instruction set semantics are usually described by an interpreter, for example in ISPS. Interpreters, however, are special programs and therefore can be used as an input to a synthesis system using algorithmic specifications.

This special case, frequently leads to confusion because two different instruction sets are involved: the instruction set interpreted by the interpreter and the instruction set of the machine to be designed. The latter is implemented in hardware and is frequently called microinstruction set. In this paper, we will refer to instructions implemented in hardware simply as "instructions" since the second level of instructions may be missing.

Because of the advantages of algorithmic specifications,we designed a synthesis system startting on that level. This system uses MIMOLA (machine independent microprogramming language) [2] as its design language. MIMOLA has been changed recently to include most of PASCALs features. Hence, the algorithmic level in our sense includes e.g. recursive procedure calls and references to arrays with an arbitrary number of dimensions. The design system itself is called MIMOLA Software System (MSS).

Usually there will be a large variety of machines being able to perform the desired behaviour. Therefore, it is necessary to restrict the design space in a number of ways. The following are the most important restrictions accepted by our design system:

1. Limitation of the number of instruction bits for immediate data and addresses. E.g., it is possible to specify that at most 24 bits per instruction are allowed.

2. Specification of a set of available arithmetic/logic units (ALUS). This restriction is obviously required if the design has to be implemented using discrete devices like TTL-circuits. It is also required, if the design system is to be extended into a standard cell silicon compiler.

3. Specification of available data memories. Our previous experience indicates that there are only few choices for data memories. Therefore it is possible to completely specify all data memories to be used in the design, repeat the design process for possible other choices and then compare the resulting designs.

   One important parameter of random access memories is their number of **ports**, that is, their maximum number of simultaneous accesses to different memory locations.Small memories are usually implemented as 2- or 3- port memories.' Larger memories frequently are single-port memories, but pseudo multiport memories can be built using memory banks and crossbar switches.

In addition to the specification of the desired behaviour and the set of design contraints, our design system needs some additional information on how to link behavioural and structural domains. This information is concerned with the implementation of high-level programming elements (like procedure calls) in hardware. Examples are given below.

The following is a sample of a complete design specification in MIMOLA. Due to space constraints, this sample is much shorter than typical design specifications. The syntax used in this example reflects version 4.0 of the design system.

```
TARGET CPU;              (*Name of the system*)

 (*Those parts of the system structure,     *)
 (*which are known before synthesis starts:*)

 STRUCTURE
 TYPE
  word   = (15:0);              (*bits 15 to 0 *)
  ext    = fields               (*packed record*)
             res : word;
             eq  : (16);        (*bit 16       *)
           end;

  MODULE B7483 <<COST=3>>
   (IN l,r : word; OUT f : word);
    BEGIN                 (*Behaviour of       *)
     f:=l + r             (*available component*)
    END;
```

```
  MODULE B74xy <<COST=10>>
   (IN a,b:word; FCT sel:(1:0); OUT f: ext);
    PARBEGIN
     f.res:= CASE sel OF
             0 : a  -  b;
             1 : a  +  b;
             2 : a  *  b;
             3 : a  /  b;
            END;
     f.eq := CASE sel OF
             0 : a <=  b;
             1 : a  <  b;
             2 : a  =  b;
             3 : a <>  b;
            END
    END;

PARTS
SR: MODULE S8 <<SIZE=8>>
     PORT P1
      (IN e :word; ADR ad:(2:0); FCT c:(0));
     PORT P2,P3
      (OUT f:word; ADR ad:(2:0); FCT c:(0));
      PARBEGIN
       WITH P1 DO
        CASE c OF
         0 : S8[ad]:=e;
         1 : ;
        END;
       WITH P2,P3 DO
        f:= CASE c OF
             0 : S8[ad];
             1 : TRISTATE;
            END
      END;

SM: MODULE S4k   <<SIZE=4096>> (IN e:word;
     OUT a:word; ADR ad:word; FCT sel:(0));
      BEGIN
       CASE sel OF
        0 : PARBEGIN S4k[ad]:=e; a:=X END;
        1 : a:=S4k[ad];
       END
     END;

PC: MODULE RP (IN e : word; OUT a : word);
     PARBEGIN
      RP:=e; a:=RP
     END;
END;

(*Information concerning replacement of*)
(*high-level language elements :       *)

LOCATIONS_FOR_VARIABLES   SH [0..4095];
LOCATIONS_FOR_TEMPORARIES SR [3..7];

MACRO
 REPLACE
  &a: WHILE &expr DO &block
 WITH
  &a:IF &expr THEN BEGIN &block; RP:=&a END
 END;
 REPLACE
  GOTO &a
 WITH
  RP:=&a
 END;
```

```
(* Desired system behaviour:  *)

PROGRAM rem;
 LABEL Lx;
 TYPE  integer = word;
 VAR   a,b     : integer;
 BEGIN
      read_pascal(a,b);
  Lx: WHILE a > 0 DO a := a - b;
      a := a + b;
 END.
```

The keyword STRUCTURE indicates that the following lines describe the internal structure of the CPU - system. At the present state of the design, this structure is only incompletely described. For example, the instruction format and interconnections between modules are not yet included. The purpose of this partial structure description is to restrict the design space.
B7483 and B74xy are available ALU-types. Their presence in the design specification does not imply that any of these is present in the final design. The design system is responsible for selecting an appropiate number of copies of these types.

SR and SM are two data memories. SR is a small memory allowing simultaneous access to at most three different locations via ports P1, P2 and P3. SM is a larger memory with a single port. SR and SM are copies of module types SS and S4k, respectively. The design system has to use these and no other copies.

Module type RP, by convention, denotes the program counter.

Variables declared in the program need some memory space. In the example, locations 0 to 4095 of SM are available for this purpose. Memory space for intermediate results is to be taken from the set SR[3..7] of locations.

The next lines tell the system, how WHILE loops and GOTO - statements are to be implemented. Each of the two replacement-rules defines a program transformation. Identifiers starting with an &-sign denote parameters. &expr, for example, corresponds to the expression, which is tested for each iteration of the while loop.

The keyword PROGRAM introduces the section describing the desired behaviour. The current version of MIMOLA includes almost all PASCAL features. Not included are real numbers, the WITH-statement and enumeration types. Furthermore, MIMOLA contains a large number of additional features like bit level addressing and explicit parallelism. The latter is indicated by the keyword PARBEGIN.

The design specifications like the one above are processed by the synthesis procedures of our MIMOLA Software System in order to generate the description of an RT-structure.

2. Synthesis

2.1 Decomposition of the design problem

Design specifications share a property with conventional programs: they may well extend over hundreds and thousands of lines. Hence, synthesis algorithms with an execution time proportional to the length of the program are acceptable but any higher complexity cannot be tolerated.

Hafer[3] demonstrated that modelling the design problem as a single optimization problem leads to an unacceptable complexity. Therefore, we partioned the design problem into a number of subproblems.

This partioning of the design problem made it possible to implement the MSS as a number of independent PASCAL programs, called components. Communication between components is via intermediate files.

2.2 Front-end tools

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. At the RT-behaviour level, there is no high-level language element (except the IF-statement) and all operations (including computations of effective addresses) are explicit.

For example, an application of the rule for transforming WHILE - loops will convert the WHILE-loop of our sample program into:

```
Lx: IF a>0 THEN BEGIN a := a - b; RP:= Lx END;
```

Another task 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:

In the following declaration, locations are manually assigned relative to a stack pointer:

```
VAR base  : word    AT SR[0]:= 0 ;
    a     : integer AT SM[base+5];
    b     : integer AT SM[base+6];
```

This form of the declaration will be assumed in the examples below. ":=0" means initialize to 0.

With this declaration, our WHILE-loop would be converted to:

```
Lx: IF SM[SR[0]+5] > 0 THEN
      BEGIN
       SM[SR[0]+5]:=SM[SR[0]+5] - SM[SR[0]+6];
       RP := Lx
      END;
```

MSSR generates an RT-level behavioural description which still contains IF-statements. In addition to the usual implementation of IFstatements by conditional jumps and unconditional assignments, MSS provides for hardwareimplemented conditional assignments and conditional expressions.
Examples:
The following forms are equivalent to the above example:

conditional jump:

```
Lx: RP:= (IF SM[SR[0]+5]>0 THEN L1 ELSE L2);
L1: SM[SR[0]+5]:= SM[SR[0]+5] - SM[SR[0]+6];
    RP:=Lx;
L2: ...
```

conditional assignment (The expression enclosed in slashes is a "write enable" condition for the first assignment) :

```
Lx: /SM[SR[0]+5] > 0/
   SM[SR[0]+5] := SM[SR[0]+5] - SM[SR[0]+6];
   RP:=(IF SM[SR[0]+5] > 0 THEN Lx ELSE L2);
L2: ...
```

Sequential execution is necessary for the first form and an implementation requires at least two instructions.
In contrast, both assignments in the second form can be done in parallel. Therefore, it can be implemented by a single instruction if hardware resources are sufficient.
It is hard to anticipate, which implementation will be the fastest under the constraints of limited hardware resources. 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.
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. 1). 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).

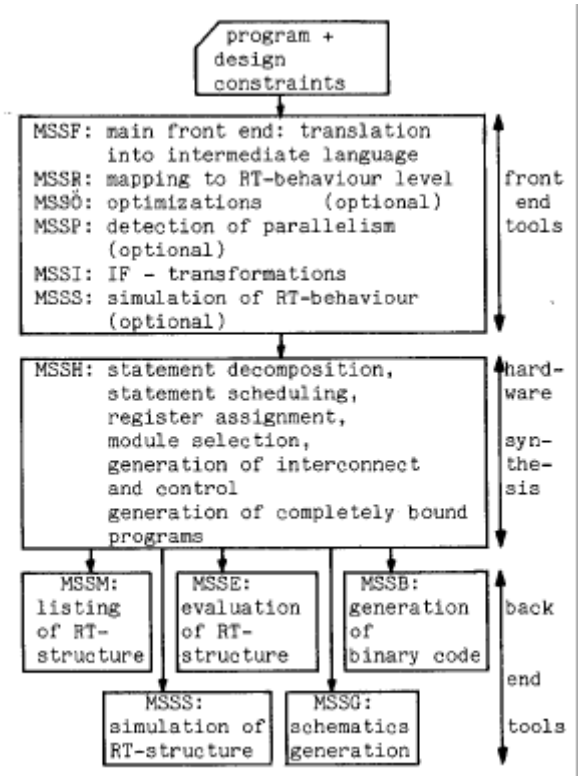

Fig. 1 Steps in the synthesis of RT-structures

2.3 The synthesis subsystem

2.3.1 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. Required 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. Therefore, statements containing many memory references must also be decomposed into simpler statements.

Finding an optimal decomposition is known to be NP-complete. Traditional compiler techniques like [4] 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.[5]), 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 a of t, where a is by a heuristic criteria the most complex subexpression of t, which can be assigned to a temporary variable without violating design constraints. In the MSS, **mostcomplex** selects a subexpression of t according to the following priority list:

1. maximum number of memory references,
2. maximum number of references to memories not used to store intermediate results,
3. common subexpressions,
4. boolean subexpressions,
5. left to right.

E.g. if two subexpressions of t contain exactly the same number of memory references and one of them represents a common subexpression, it will be selected by **mostcomplex**.

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 a temporary
    variable;
    push assignment onto the top of a stack;
    replace e by a read operation of the tempo-
    rary 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 2.2:

SM[SR[0]+5] := SM[SR[0]+5] - SM[SR[0]+6]

Fig. 2 is a flow tree representing this statement. Numbers in parentheses indicate the sequence, in which **decompose** traverses the tree. Reading from and writing to memories is simply denoted the name of the memory.

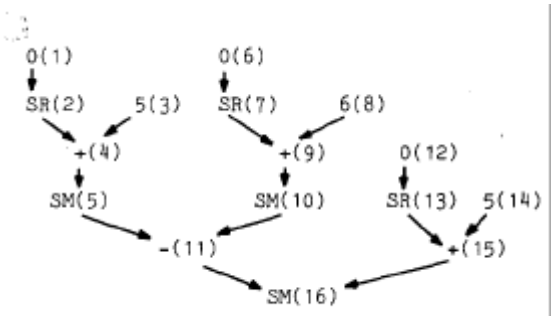

Fig. 2 Flow tree

decompose will deposit the following sequence of statements in the stack:

contents of stack	pushed at node
SR[v1]:=SM[SR[0]+6];	11
SR[v2]:=SR[v1]-SM[SR[0]+6];	16
SM[SR[0]+5] :=SR[v2]	(by final push)

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.

2.3.2 Statement scheduling

The final RT-structure usually contains several memories, e.g. a main memory, a register memory, a condition code register and a program counter. Each of the assignments generated in the previous step represents an assignment to one of the memories. It would be a waste of time if instructions could only cause assignments to a single memory. Therefore it is necessary, that the design system is able to allocate several assignments to a single instruction.

Example:

The two assignments

SM[SR[0]+5] := SR[v2] and RP := Lx

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

The sequence, in which assignments are executed, depends upon this allocation and therefore this allocation is called "scheduling of statements". A problem closely related to that of scheduling statements in the MSS is that of

compacting microprograms. Many algorithms for microcode compaction have been published (see e.g.[6]).

Scheduling algorithms have to consider dependence relations in order to preserve a program's meaning.

Scheduling in the MSS is based upon the following two relations:

Def.: Statement si is data dependent on statement sj (si **dd** sj) iff sj generates the contents of a memory location read by si.

Example:

```
BEGIN
 s1: a:=p;
 s2: c:=q * a;
 s3: a:=a * c;
END;
```

The following relations hold:
s2 **dd** s1; s3 **dd** s1; s3 **dd** s2

Def.: Statement si is anti dependent on statement sj (si **ad** sj) iff si writes to a location read by sj, but the contents generated by si is not the one needed by sj.

Example: In the sequence above, s3 **ad** s2 holds.

Frequently, no distinction is made between dd and ad (e.g. in [6]). As Vegdahl [7] points out, this prevents some blocks of code being moved as a whole. Therefore the distinction between the two relations is important. The definitions of dd and ad in [81 are applicable only to sequential blocks, because they make use of the order, in which statements are written. MIMOLA allows the user to specify parallel blocks like

```
PARBEGIN a:=b; b:=a END.
```

The two assignments are expected to interchange the contents of a and b and the order in which they appear in the program is redundant. The above definition can be applied to sequential as well as to parallel blocks. A more precise form is contained in [9].

Using dd and ad, the set of allowable schedules can be defined. Let MI(si) be the instruction allocated to si. Let MI(si)>MI(sj) denote that the execution of sj precedes that of si and let MI(si)_>MI(sj) denote that either MI(si) > MI(sj) or MI(si) = MI(sj). Then, the following *conditions must* hold:

1. For all generated instructions MI(si), **treetoobig**(MI(si)) must be false.

2. ¥ si,sj: si **dd** sj : MI(si) > MI(sj)
   ¥ si,sj: si **dd** sj there does not exist an sk such that sk **ad** si and MI(si)>MI(sk)≥MI(sj)

As .long as these conditions are met, many different scheduling algorithms may be used. In the MSS we modified the pairwise comparison algorithm [10] such that is does no longer rely on a strict order of statements.

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.

2.3•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, it is almost trivial:

```
Mark all locations being available for temporaries as deallocated.
FOR ALL instructions i in the present block 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 2.3.1:

```
SR[v1] := SM[SR[0]+6];                  SR[v1]
SR[v2] := SR[v1] - SM[SR[0]+6];         SR[v2]
SM[SR[0]+5]  := 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 sche-

duling 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.

2.3.4 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 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).

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 copies of each type such that hardware resources are sufficient for every instruction and such that
$c = \sum_m (x_m * c_m)$ 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 copies is that:

$$\forall\ i,\ \forall g \in F_i^*:\ \sum_m (x_m) \geq \sum_{j \in g} (f_{i,j}), \qquad (1)$$

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

Example:

Assume that there are 4 module types. They are able to perform the following operations:

type 1: +; type 2: +,-; type 3: +,OR; type 4: -

An instruction containing one addition and one subtraction would generate the following three relations:

$$\begin{aligned} \{+\ \}&: x_1 + x_2 + x_3 \geq 1 \\ \{-\ \}&: x_2 + x_4 \geq 1 \\ \{+,-\}&: x_1 + x_2 + x_3 + x_4 \geq 2 \end{aligned}$$

Let $b_g = \max_i \left(\sum_{j \in g} f_{i,j} \right)$.

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

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

The selection task therefore reduces to minimizing $c = \sum (x_m * c_m)$ subject to the set (2) of constraints. This is a classical integer programming (IP) problem. In the MSS, the GOMORY-I algorithm [11] is used to solve the 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. The reason for that is that the size of F^* is typically much less than the sum of the sizes of F_i^*. Usually F^* does not contain more than 40 elements an

the IP-problem can be solved in less than 100 ms on a 1 Mips machine.

If a program contains an operation which cannot be performed by any of the available module types, the MSS creates a new type being able to perform just the required operation. A warning is generated whenever this occurs.

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.

2.3.5 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.

Example:

MIMOLA V.4 uses the dot notation in order to specify a particular port of a module. E.g. SR.P2 means "port P2 of module SR". Consider the sequence:

```
SR[3]       := SR[3] - SM[SR[0]+6];
SM[SR[0]+5] := SR[3]
```

The assignment

```
SR.P1[3]       := SR.P2[3] - SM[SR.P3[0]+6];
SM[SR.P2[0]+5] := SR.P3[3]
```

as well as the assignment

```
SR.P1[3]       := SR.P2[3] - SM[SR.P3[0]+6];
SM[SR.P3[0]+5] := SR.P2[3]
```

are valid. (SM is a single port memory and there is no need to specify its port). For the first assignment, both P2 and P3 must be connected to the address input of SM. Hence, the second assignment is superior.

In the present implementation of the MSS, 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 a few instructions at a time, starting with the most complex instructions.

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

2.3.6 Generating control

In the MSS 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 [12] is used to control RT-modules: for each module with a control input, there is a corresponding instruction field f., which may be used to select one of the module's operations. Since not all the modules are used simultaneously, some of them may **share** instruction bits.

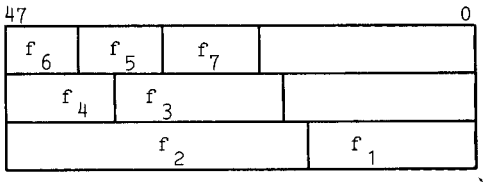

Fig. 3 Sharing of instruction bits

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 completion time is minimized.

This problem is one of the hard scheduling problems. The solution used in the MSS is one of the common scheduling policies: schedule long fields and fields with many conflicts first. No backtracking is done. If there are several possible assignments for a field, the "best fit" algorithm is used.

Optimization techniques similar to our's have been used by Takagi [13] and in the CMU-DA system [14]. The basic idea in both cases is modelling the problem as a clique partitioning problem. The heuristics used for solving the clique partitioning problem and the scheduling problem are very similar.

2.3.7 Generating completely bound programs

Using the results of sections 2.3.5 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 [15]. Completely bound programs may be processed by the back-end tools MSSM, MSSS, MSSB, MSSE and MSSG (c.f. Fig. 1).

2.3.8 Sample output

The following is a partial description of the RT-structure generated for our sample input. Note that the instruction format and the interconnections between modules are now described. One copy of each of the ALU types B7483 and B74xy has been selected.

```
TARGET CPU;
STRUCTURE

 ...          (*module types as in section 1*)

 PARTS
  U0: B7483;
  U1: B74xy;
  SR: S8;
  SM: S4k;
  PC: RP;
  MaU1: MODULE Nmux02x16 (OUT f:(15:0);
          IN inp00,inp01:(15:0); FCT sel:(0));
         BEGIN
          f:= CASE sel OF
               0   : inp00;
               1   : inp01
              END
         END;
  ...              (*some parts omitted here*)

 INSTRUCTION            (*instruction format*)
  I : FIELDS
        MaU1    :  (60);
        ....
        D2      :  (31:16);
        ....
      END;
 CONNECTIONS
  SM                -> U1              .b;
  MaU1              -> U1              .a;
  I .MaU1           -> MaU1            .sel;
  I .D2             -> MaU1            .inp01;
  SR.P2             -> MaU1            .inp00;
  .........
END;
```

The interconnections with multiplexer MaU1 can be seen in Fig.4, which is a graphical representation of the resulting structure. Address inputs to SR and control inputs except to MaU1 have been omitted.

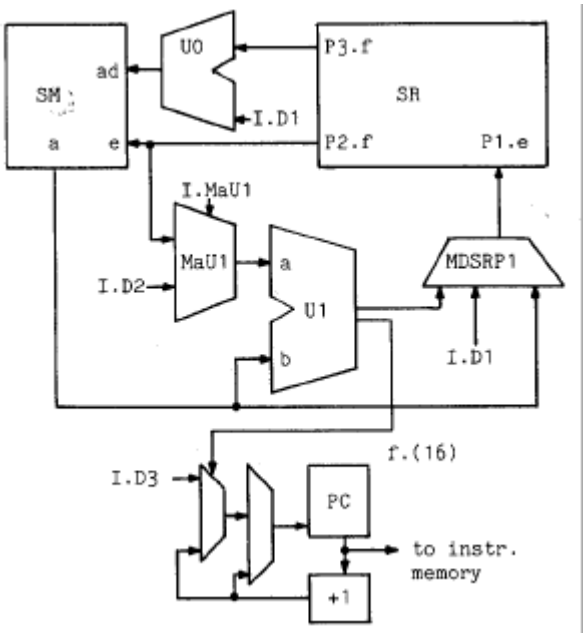

Fig. 4 Synthesized RT-structure

2.3.9 First results

One of the earlier examples, which was used for testing the MSS, was the mergesort algorithm as described by Wirth [16]. The performance of the RT-structures created by the MSS was compared with the performance of an IBM/370 type machine. The results are shown in table 1.:

	MSS designs		SIEMENS 7.760 (1 Mips)
# ports SM	1	4	PASCAL V 3.1
Immediate fields	1	4	(no runtime checks)
ALUs	1	4	
execution time [ms]	3,3	1,0	6,2
program code [bytes]	1143	755	936

Table 1: Performance and code of MSS designs

The speed of the MSS designs is not a result of tailoring a machine to exactly one program. The synthesis algorithm does not, for example, hardwire constants (except zero). The resulting RT-structure is mainly influenced by some of the programs properties like addressing modes, used arithmetic/logic operators and the amount of bit level addressing. In a case study we analysed the effect of adding the mergesort algorithm to a behavioural specification con

sisting of the gomory-I algorithm. Except for a single bit wire, the resulting structure was the same.

The MSS designs in table 1 require at most 22% more program code than the SIEMENS. This is a remarkable result, because the direct encoding scheme is believed not to lead to compact code.

Flexibility has always been a design goal for the MSS. This allows us studying the effect of different design alternatives. Fig. 5 for example, is the result of a study comparing different addressing modes for variables. It turns out that the addressing mode significantly affects the number of executed instructions, even if 4 ALUs are present in the design.

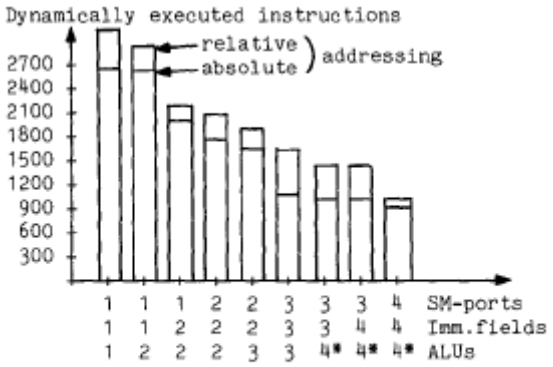

Fig.5 Effect of addressing modes,
*: 4th ALU generated only for relative addressing; SR : 3 ports.

The results of the present MSS have been compared with those of an earlier version of the MSS, called MSS1 [171.

MSS1 has been used to design a processor for a design specification consisting of the kernel of an operating system and some typical sections of PASCAL-programs. The MSS1 was not particularly successful in minimizing the number of data paths and therefore these have been reduced manually by about 50%. The resulting structure and the structure generated by the current MSS had the same number of paths.

3. Other tools in the MSS

3.1 Retargetable code generation

To a certain extent generated hardware structures may be modified by changing the resource constraints. Design iterations initially should be done by using this method, that is, by executing the synthesis procedures for different design constraints. However, after a certain number of iterations, ideas about the hardware structure become more and more precise. As a result, the designer usually knows the structure he would like Hopefully, it is similar to a structure generated by the synthesis system. But, in order to take advantage of the ideas of human designers, it is necessary to allow manual modifications to automatically generated structures (c.f. Fig. 6).

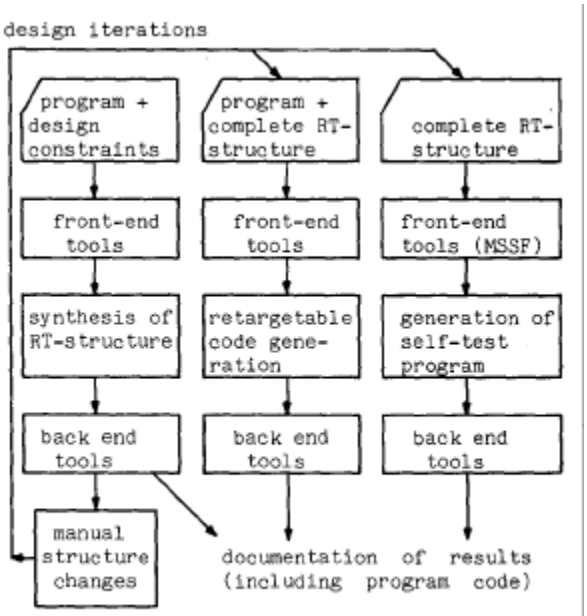

Fig. 6 Global view of the MSS

It is easy to document these changes because MIMOLA (in contrast to other languages) can be used for the description of the synthesis **result** as well as for the behavioural specification. The problem with manual changes is that they may result in an incorrect design. "Incorrect" means that the modified structure is not capable of executing the programs specifying the desired behaviour.

Therefore we have to provide the user with tools enabling him to check correctness. We do so by including a so-called retargetable code generator in the MSS [181. This code generator tries to generate code for the machine described in the hardware description section of its input. This machine is called the **target.** If the code generator is able to generate code for a manually modified structure, the design is still correct. If it fails to do so, it is either incorrect or the code generator does not know enough "tricks". "Tricks", which are frequently used by human microprogrammers, represent a certain amount of knowledge. With MIMOLA it is possible to convey this knowledge to the MSS, because MIMOLA allows the user to define valid program transformations. **These** transformations are used by the code **generator like** in a tiny expert system.

Example:

```
REPLACE_CONDITIONALLY 0 WITH (0 AND Q) END
```

This transformation is required for programming the AMD2900 series of bitslice chips. Hardwired zeroes exist at the input of the AMD2900 ALU. In order to pass these to the output of the ALU, the "AND"-function must be used. The *suffix* "CONDITIONALLY" indicates that this rule has to be applied only if its application results in good code.

The code generator has been used for more than 20 different targets.

3.2 Test generation

Because of the increasing problem of testing VLSI chips, we also designed a component MSST, which automatically generates self - test programs (diagnostics) from a given structural hardware description [19]. These programs are intended to be executed by the real hardware.

Recently, an additional tool has been completed at the university *of* Aachen, Germany. This tool computes testability *measures for a* given hardware. The output *of* this tool as well as the output *of* MSST is intended to be used in design iterations in order to improve the testablity *of* an RT-structure.

3.3 Back-end tools

The synthesis subsystem, the code generator and the test generator are the three main components *of* the MSS. All three components generate completely bound *programs. These programs as well as* the description *of* the RT-structure can be processed by a set *of* components called back end tools.

Probably the most important back end tool is MSSM. MSSM generates listings *of* the resulting RT-structures and completely bound programs. The language used for these listings again is MIMOLA. This is possible, because MIMOLA is able to describe RT-structures and completely bound programs. Therefore, the output of MSSM can be used as input to MSSF. This feature is necessary in order to support design iterations without having to learn two or more different languages.

Another important tool is the simulator. The simulator is able to simulate the **structure** of the processor. It is based upon an analysis of the interconnections in the processor and therefore is able to detect unwanted sideeffects.

The simulator allows the user to monitor the execution of the program on the target hardware. Note, that the user does not have to load the instruction memory manually because binary code is generated by the MSS.

The simulator can be used to validate a design independently of the synthesis and code generation tools. This is especially valuable in order to detect errors made by these tools.

A third back end tool is MSSB. MSSB generates an instruction memory map in human readable form.

The purpose of MSSE is to evaluate the target structure. With the help of the simulator it computes the expected execution time of 'the program and generates utilization statistics for the RT-modules.

In order to help visualizing the result of a design process we are currently implementing MSSG. MSSG will automatically convert textual hardware descriptions into schematics. MSSG is based upon an extension of an algorithm published in 1985 [20].

The back end tools as well as all other tools are implemented in standard PASCAL. The MSS has been installed on SIEMENS, VAX, Eclipse, Sun and Apollo computers.

4. Conclusion

Synthesis methods for the design of digital hardware are capable of producing correct designs in a short turn-around time. This paper deals with the synthesis of RT-structures from an algorithmic design specification. By carefully partitioning the design process into a sequence of subprocesses we have tried to reduce the complexity and to keep interactions between the subprocesses as small as possible. The partitioning was done such that design decisions are delayed as long as possible. The complexity of the resulting subprocesses allows synthesizing hardware for large design specifications.Probably the most remarkable achievement is the algorithm for selecting modules from a set of predesigned module types. This algorithm is both globally optimizing and very fast.

5. References

[1] G. Zimmermann : Eine Methode zum Entwurf von Digitalrechnern mit der Programmiersprache MIMOLA, Springer Informatik Fachberichte, Vo1.6 1976, S. 465-478

[21 R. J6hnk and P. Marwedel: MIMOLA Language Reference Manual, Revision 3, (in preparation)

[3] 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

[4] B. Prabhala and R. Sethi : Efficient Computation of Expressions with Common Subexpressions, Journal of the ACM, Vol. 27, 10980), p. 146-163

[5] R. Sethi and J.D. Ullman: The Generation of Optimal Code for Arithmetic Expressions, Journal of the ACM, Vol. 17, 40970), p. 715-728

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

[71 S.R. Vegdahl : Local Code Generation and Compaction in Optimizing Microcode Compilers PhD thesis, Report CMU-CS-82-153, Carnegie-Mellon University, Pittsburgh, 1982

[8] 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

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

[10] 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

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

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

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

[14] 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

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

[16] N. Wirth, Algorithmen and Datenstrukturen, Teubner, Stuttgart, 1975

[17] G. Zimmermann : MDS - The MIMOLA Design Method, Journal of Digital Systems,Vo1.4, 30980), S. 337-369

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

[19] G. KrUger: Automatic Generation of SelfTest Programs - A New Feature of the MIMOLA Design System, 23th Design Automation Conf., 1986

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