On the Use of Hierarchies in the MIMOLA Hardware Design System

Peter Marwedel

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

Abstract

Until recently, the use of hierarchies in CAD for VLSI has almost exclusively been restricted to layout and simulation problems. The paper describes representation and handling of design hierarchies in the MIMOLA design system, featuring RT-level synthesis, test program generation and retargetable code generation. A method of embedding these tools in a common design environment, providing access to the hierarchy, is described. Advantages of having a design hierarchy are presented for each of the tools. In addition, relevant features of the MIMOLA language are explained.

1. Introduction

In the design automation area, hierarchy has almost exclusively been used for simulations and for layout systems [9]. In these cases, the problem of designing hierarchical tools has been addressed for a single tool. With the advent of integrated design systems, covering more design problems and more levels of abstractions, it is becoming necessary to handle hierarchies in a more general context [9].

The hardware design system MSS (**M**IMOLA **S**oftware **S**ystem) which we developed during recent years, includes tools to solve several design problems. The MSS is based upon MIMOLA (**m**achine **i**ndependent **m**icro-programming **la**nguage) [7], a language covering several levels of abstraction.

The highest level is the algorithmic level, similar to the level at which PASCAL programs describe algorithms. The lowest level is the register transfer (RT-) structure level. At this level, hardware is described in terms of memories, registers, arithmetic/logic units (ALUs), multiplexers, busses and the interconnections between these elements [14].

The tools within the MSS perform the following tasks:

1. Simulation.
2. Synthesis of RT-structures from an algorithmic specification [12,13].
3. Generation of the binary form of an algorithm described at a PASCAL-like level. The machine description is part of the input ("retargetable compiler") [10].
4. Automatic generation of self-test programs for a given machine [8].
5. Generation of schematics [18].

This research has been supported by the German Ministry of Research and Technology (BMFT) under contract NT 2816 A9.

The MSS, as described in [11], did not make use of design hierarchies. This paper describes our approach for extending the MSS towards a hierarchical, integrated design system.

Two major issues have to be solved: that of suitable design language and that of a suitable design system.

Let us first consider the language issue by briefly reviewing some relevant features of the present version (version 3.4) of MIMOLA.

2. Hierarchical features of the MIMOLA language

MIMOLA has been specifically designed to support RT-level synthesis. It is an important characteristic that the specification at the algorithmic level as well as synthesized RT-structures can be described with this language. Hence, both the specification and the implementation can be represented in MIMOLA.

This could also be done with a sufficiently general simulation language like CAP (DACAPO) [5,16] or VHDL. In a typical design process using such a language, one would first simulate the specification. Next, one would manually design an implementation. This implementation would again be simulated. In such an approach, simulator does not need to know whether it is simulating a specification or an implementation. This knowledge exists only in the user's mind.

The situation is different for tools handling several levels of abstractions simultaneously, like synthesis, abstraction, or verification tools. The MSS, for example, allows the user to specify a behaviour and a partial structure (e.g. some memories) and to synthesize interconnections and control automatically. The RT-synthesizer must be able to distinguish between specified behaviour and a partially specified structure. There are two ways of providing the required information:

1. The distinction between structure and behaviour is made in the design language.
2. The design environment provides this information, e.g. behaviour and structure could be kept in different files.

For MIMOLA (as well as for CIRCAL [2] and SBL [3]), the first alternative has been chosen. This choice allows the inclusion of behaviour and structure in a single listing.

Behavioural specifications in MIMOLA use a syntax and semantics that is almost equivalent to that of PASCAL. Available language features include recursive procedures, dynamic variables, and, in addition to PASCAL, constructs of systems implementation languages like C. Differences between MIMOLA and PASCAL include the following:

functions cannot have side effects,
static (own) variables are possible,
there are no REAL variables,
for hierarchical designs, the body of the "main program" is of a restricted form.

Since MIMOLA is a hardware design language, **implementation** descriptions are restricted to denoting **hardware structures**. Hardware structures are described in terms of modules and a netlist. The MSS provides the user with a check for the completeness of the netlist. Thus, the MSS detects errors which cannot be detected in systems not requiring a netlist. Users have reported that this check turned out to be valuable for many applications. The netlist is a basic source of information, both for the code generator and for the test program generator MSST. As a result, MSST is able to generate a small set of machine instructions testing for stuck-at errors on physical wires.

RT-modules, in turn, are specified by their behaviour and/or their structure. Thus, hierarchical nesting of modules is possible:

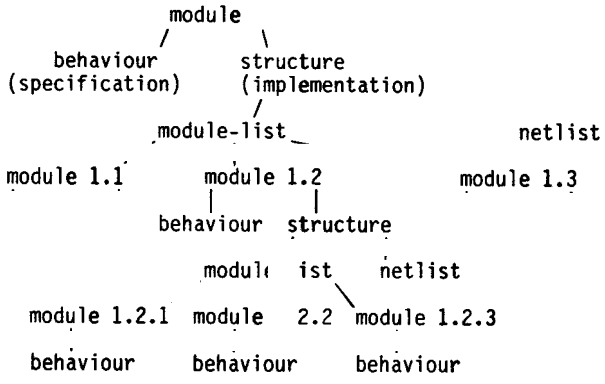

Fig. 1 Implementation hierarchy

Note that for the bottom level modules, the "leaf modules", only the behaviour is defined in terms of some primitive operations of the language.

The following is an example of a hierarchically structured MIMOLA description:

```
MODULE Stack (INOUT data:(15:0);   (*16 i/o lines*)
              FCT ctl : (1:0)  ; CLK clock : (0) );

 STRUCTURE   (*incomplete structure decription of*)
             (*Stack: no control logic specified *)
  TYPE
   word = (15:0);          (*bitstring of length 16*)

  PARTS                    (*incomplete module list*)

   Memory   MODULE SRam  <<SIZE=#10000>>
             (INOUT d:word;      (*data i/o lines *)
              ADR   a:word;      (*address input  *)
              FCT   s:(0) ;      (*function select*)
              CLK   c:(0) );     (*clock input    *)
            (*no structure defined for leaf module*)
             BEHAVIOUR
              BEGIN
               AT c DO
                CASE s OF
                 0 : SRam[a]:=d;          (*write*)
                 1 : d <- SRam[a];        (*read *)
                END;
              END;

   Pointer: MODULE Reg <<SIZE=1>>
                    (IN i : word; OUT o: word;
                     FCT   s:(0); CLK c:(0)  );
             BEHAVIOUR
              BEGIN
               AT c DO
                CASE s OF
                 0 : Reg := i;
                 1 : o <- Reg;
                END;
              END;

  CONNECTIONS                (*incomplete  netlist*)
   clock          -> Memory .c;
   clock          -> Pointer.c;
   Pointer        -> Memory .a;
   data           -> Memory .d;
END_structure;

BEHAVIOUR                    (*behaviour of Stack*)

 PROCEDURE push    (IN i : word);
  BEGIN
   Memory[Pointer]:=i;
   Pointer := "INCR" Pointer;
  END;

 PROCEDURE pop     (OUT o : word);
  BEGIN
   Pointer:= "DECR" Pointer; (*assign to store *)
   o <- Memory[Pointer];     (*assign to signal*)
  END;

 PROCEDURE read    (OUT o : word);
  BEGIN
   o <- Memory[Pointer];
  END;

 PROCEDURE clear   ;
  BEGIN
   Pointer:=0;
  END;

 BEGIN_behaviour
  AT clock DO
   CASE ctl OF
    0 : push(data);
    1 : pop (data);
    2 : read(data);
    3 : clear;
   END_case;
END_behaviour;
```

Procedures and functions, which are called in the block enclosed by **BEGIN_behaviour** and **END_behaviour** are exported by the module "Stack". Hence, modules are similar to abstract data types and to monitors (c.f. [3,16]). They contain some private storage and some public operations (procedures). Note that MIMOLA in contrast to most other languages requires that public operations can be selected by control codes (the case labels). In the module header, control code inputs are denoted by **FCT**. Control code inputs can only be used as **CASE**-selectors. Clock inputs are denoted by **CLK**. Clocks can only be used in an **AT**-statement.

3. Common Tool Environment

.et us now start studying the design system issue.

The integration of several design tools requires a suitable tool environment. Propably the most important parts of such an environment are a common design data base and a common system monitor [6]. The purpose of the latter is to call appropriate tools,

check consistency of the design data base, and to provide an interface to help, display and edit functions. In a tightly coupled system, it handles **all** communication with the user. In a loosely coupled system, tools are allowed to communicate with the user directly.

After careful analysis of our design tools, we concluded that none of these needs to process hierarchically structured **inputs**. Hierarchies can be expanded before these tools start operating. This does not mean, however, that the hierarchy is lost. The tools just work on an expanded copy of the design description and the result of the tool execution is stored in an appropiate level of the design hierarchy. Hierarchy expansion is another part of the common tool environment.

This environment and the tools available in the present MIMOLA design system are shown in Fig. 2:

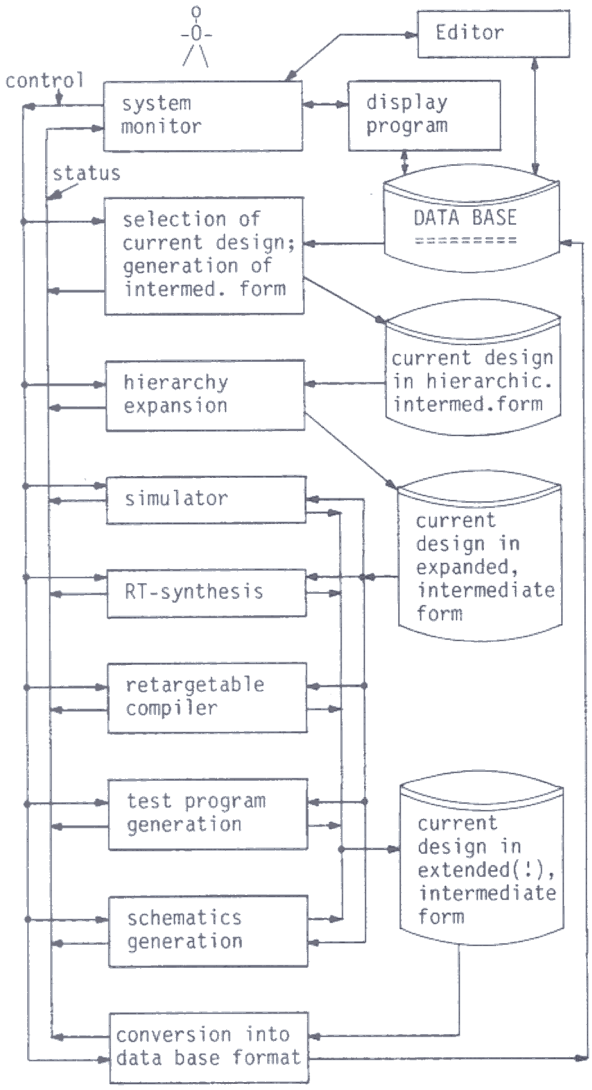

Fig. 2 Common tool environment

This scheme is applicable to several degrees of integration. Currently we are implementing a loosely coupled system, using (with very few exceptions) only standard PASCAL. In this approach, all the boxes shown in Fig. 2 are separate programs, communicating via files. The resulting system is highly portable.

In cooperation with CADLAB of Paderborn, we will generate tightly coupled implementation, taking full advantage of todays workstation technology.

4. Use of the hierarchy by existing tools

4.1 The simulator

The separation of the hierarchy expansion and the simulation eludes the need for handling the hierarchy explicitly in the simulator. The hierarchy is just expanded to give the required amount of simulation details. The only requirement for the simulator is that it covers all relevant levels of abstraction. Our current simulator for RT-structures will be extended to fulfill this requirement.

4.2 The retargetable compiler

Currently, the most frequent application of our retargetable compiler [10] is the generation of microcode for designs based upon the AMD-2900 series of chips [1]. These chips include arithmetic/logic units as well as registers and RAMs. The compiler requires that modules are either functional or storing. Hence, the description of AMD-chips must be expanded until this condition is met.

With an automatic expansion of the hierarchy, it is possible to store descriptions of AMD29xx- or similar chips in a library and to describe a processor by selecting and interconnecting elements from the library. This description then defines the target architecture for which the retargetable compiler generates binary code. The behavioural specification is considered to be the high-level source program.

Example

This example is used to explain how a connection to the address input of memory port B is expanded. Only those parts of the description which are relevant to the expansion are presented.

Assume the following module types are present in the data base:

```
MODULE AMD29203(IN baddr:(3:0);....
 STRUCTURE
  PARTS
   RAM : MODULE S1  <<SIZE=16>>
         PORT A (ADR addr : (3:0);   );
         PORT B (ADR addr : (3:0);   );
         ...
   ALU : MODULE A1  (...)
         ...
  CONNECTIONS
   baddr -> RAM.B.addr;

 END;

MODULE Mem4k(OUT io:(31:0);
```

The user could describe his design as follows

```
MODULE MyProcessor;
 STRUCTURE
  PARTS
   Slices : MODULE AMD29203; EXTERNAL;
   CtlMem : MODULE Mem4k   ; EXTERNAL;
```

```
  CONNECTIONS
   CtlMem.io.(13:10) -> Slices.baddr;
   ...
 END;
```

The code generator would operate on the following expanded form of the target description:

```
MODULE MyProcessor;
 STRUCTURE
  PARTS
   Slices/RAM : MODULE ...;
   Slices/ALU : MODULE ...;
   CtlMem     : MODULE ...;
   ...
  CONNECTIONS
   CtlMem.io.(13:10) -> Slices/RAM.B.addr;
   ...
 END;
```

4.3 Test program generation

The test program generator MSST [8] is a tool which automatically generates self-test programs for a given processor. MSST generates binary machine code for the following tasks of a self-test program:

1. application of test patterns to the inputs of RT-level modules.
2. observation of the module's response.
3. propagation of the response to the input of a comparator.
4. comparision with the known good response.
5. observation of the result of the comparison.

Example:

Assume the following conventions are used:

- "%" denotes binary numbers,
- "PC" is the name of the program counter,
- "CC" is the name of the condition code register,
- "ErrorExit" is the entry point of a routine reporting errors.

Then, the following three instructions could be generated for a test of location 0 of memory M:

```
(*apply test pattern:*)
   M[0]:= %0101..01;

(*observe response of M[0]; propagate through the*)
(*ALU by adding 0; compare response: *)
   CC  := (M[0] + 0) <> %0101..01;

(*observe result of comparison:*)
   PC  := IF CC THEN ErrorExit ELSE "INCR" PC;
```

By default, MSST applies patterns of alternating ones and zeroes to all the inputs of RT-level modules. Fault coverage is improved, if the user explicitly defines good test patterns. For combinatorial circuits with a known gate- or switch-level description, these patterns can be computed by using conventional TPG-tools.

Therefore, generation of self-test programs using MSST and a TPG tool consists of the following steps:

```
1:
 FOR each RT-module of the current processor
  DO
    expand the module down to the gate level;
    generate test patterns using some conventional
    ATG tool;
    store the result in the data base;
  OD;
```

```
2:
 Expand the current processor down to RT-modules;
3:
 Let MSST compile binary code resulting in an appli-
 cation of the test patterns to internal nodes and
 and in branches to ErrorExit if the response is
 wrong.
```

4.4 The RT-Synthesizer

The input to our RT-level synthesizer MSSH [11,12,13] contains a complete behavioural specification and a partial description of the structure. Typically, the latter consists of a description of the data memories. MSSH selects appropriate functional modules, generates the control unit and generates the netlist.

MSSH can be used hierarchically by changing the "current module under design (CUD)".

For example, the description of module "Stack", as it has been given above, could be used as input to MSSH. MSSH would select modules for operations "INCR" and "DECR" and it would generate control and the netlist.

The resulting design of "Stack" could be used as **PART** of a processor requiring operations "push" and "pop". This design sequence is called "bottom-up design". Top-down design would also be possible.

MSSH is an adequate design tool for complex modules, because it generates a control unit containing a program counter and an instruction store. For simple modules, a separate tool for logic synthesis is required.

RT-level synthesis is a special case in that hierarchical **outputs** for a **fixed CUD** could be useful. Frequently, the library of predesigned modules contains devices which have to be augmented to fit into a particular application. For example, tristate drivers have to be added to some of the outputs, or an ALU has to be augmented with some additional circuitry. In other cases, simple decoders have to be added to the control inputs of some modules. Synthesis is simplified if the resulting augmented, "ideal" modules can be treated like available library modules.

Generation of hierarchical outputs by MSSH would not be required, if a separate logic synthesis step were used to create the "ideal" modules.

It is still an open question whether or not the synthesizer could accept a hierarchically structured input for a fixed CUD. Many of the available components like microprocessors and bit slice devices are hierarchically structured. But only few ideas exist, on how these devices could be used in a synthesis system. There are many open questions in the only approach [Pee84] to this problem we are aware of.

4.5 Schematics generation

Textual descriptions of hardware structures are difficult to comprehend. Therefore, we designed a program generating schematics from a given textual hardware description [17]. Again, the separation of hierarchy expansion and schematics generation simplified the tool.

4.6 Future expansions of the tool set

In cooperation with the universities of Kaiserslautern and Paderborn, we are currently working on an extension of the current MIMOLA design system down to the layout level. The environment shown in Fig. 2 is general enough to accomodate additional tools like hierarchical floor planners [18], TPG tools, fault simulators, logic synthesizers and layout synthesizers. It would also provide a suitable setting for verifiers like VERENA [4].

Conclusion

It is commonly accepted that the complexity of VLSI systems makes hierarchical design tools necessary. While this is true in general, methods for handling the hierarchy in a particular tool are not obvious. This paper demonstrates how the hierarchy can be taken into account by a set of non-standard design tools, using an integrated tool environment. Tools operate on an expanded copy of the current design and therefore (with the exception of RT-level synthesis) tools do not need to worry about the hierarchy. Nevertheless, hierarchy information is kept throughout all phases of the design.

References

[1] Advanced Micro Devices : Bipolar Microprocessor and Interface Data Book, Sunnyvale, 1983

[2] B.S. Davie and G.J. Milne : The Role of Behaviour in VLSI Design Languages, Proc. of the IFIP WG 10.2 Conf. "From H.D.L. Descriptions to Guaranteed Correct Circuit Designs", Grenoble, 1986

[3] G.C. Gopalakrishnan : From Algebraic Specifications to Correct VLSI Circuits, Proc. of the IFIP WG 10.2 Conf. "From H.D.L. Descriptions to Guaranteed Correct Circuit Designs", Grenoble, 1986

[4] W. Grass and R. Rauscher : CAMILOD - A Program System for Designing Digital Hardware with Proven Correctness, Proc. of the IFIP WG 10.2 Conf. "From H.D.L. Descriptions to Guaranteed Correct Circuit Designs", Grenoble, 1986

[5] K. Groning, K.-D. Lewke and F. J. Rammig : A Unified Multilevel Simulation Technique, Proc. ICCAD, 1984

[6] A. Di Janni: A Monitor for complex CAD systems, 23rd Design Automation Conference, 1986, p. 145-151

[7] R. Jöhnk and P. Marwedel : MIMOLA Language Reference Manual, Language Versions 3.4 and 4.0, Report of the Institut für Informatik, University of Kiel, 1987

[8] G. Krüger : Automatic Generation of Self- Test Programs - A New Feature of the MIMOLA Design System, 23rd Design Automation Conference, 1986, p. 378-384

[9] T. Lengauer : Exploiting Hierarchy in VLSI Design, in: F. Makedon et al. (ed.): VLSI Algorithms and Architectures, Proc. Aegean Workshop on Computing, Lecture Notes in Computer Science, Vol. 227, Springer, 1986

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

[11] P. Marwedel : Ein Software - System zur Synthese von Rechnerstrukturen und zur Erzeugung von Mikrocode, habilitation thesis and report of the Institut für Informatik und Praktische Mathematik, University of Kiel, Kiel, submitted 1985

[12] P. Marwedel : A new synthesis algorithm for the MIMOLA software system, 23rd Design Automation Conference, 1986, p. 271-277

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

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

[15] A.J.H.M. Peels : A Design Method for Microprocessor-Based Systems, Proefschrift, Department of Computer Science, Twente University of Technology, Enschede, 1984

[16] F. Rammig, Hierarchical Modular Description of VLSI Systems, Proc. IEEE Workshop on VLSI and Software Engineering, 1982

[17] L. Terasa: Graphische Darstellung von Hardware-Beschreibungen, master's thesis, Institut für Informatik und P.M., University of Kiel, 1987

[18] G. Zimmermann : Top-Down Design of Digital Systems, in: E. Hörbst (ed.): Advances in CAD: Logic Design and Simulation, North-Holland, 1986