Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MSS *

Peter Marwedel, Wolfgang Schenk

University of Dortmund, Informatik XII
Dortmund, Germany

Abstract

This paper demonstrates how the different tools in the MIMOLA hardware design system MSS are used during a typical design process. Typical design processes are partly automatic and partly manual. They include high-level synthesis, manual post-optimization, retargetable code generation, testability evaluation and simulation. The paper demonstrates how consistent tools can help to solve a variety of related design tasks. There is no other system with an equivalent set of consistent tools. A key contribution of this paper is to show how current high-level synthesis systems can be extended by retargetable code generators which map algorithms to predefined structures. This extension is necessary in order to support manual design modifications.

1 Introduction

The MIMOLA hardware design system MSS is a set of tools for the design of digital hardware structures. Main emphasis is on programmable structures. The MSS contains tools for high-level synthesis [10, 9], for retargetable code generation [16, 12], for test generation [6, 7] and for simulation and schematics generation. Due to the lack of space, we cannot describe these tools in detail in this paper and the interested reader is requested to study cited references. Whereas each of these tools has been described separately, there does not exist a comprehensive description of how these tools can be used together. The first three of the tools mentioned above are especially designed for programmable hardware structures and therefore can be used for software/hardware codesign.

A typical design process using the tools just mentioned, is shown in fig. 1. The organization of this paper follows the control flow shown in this figure.

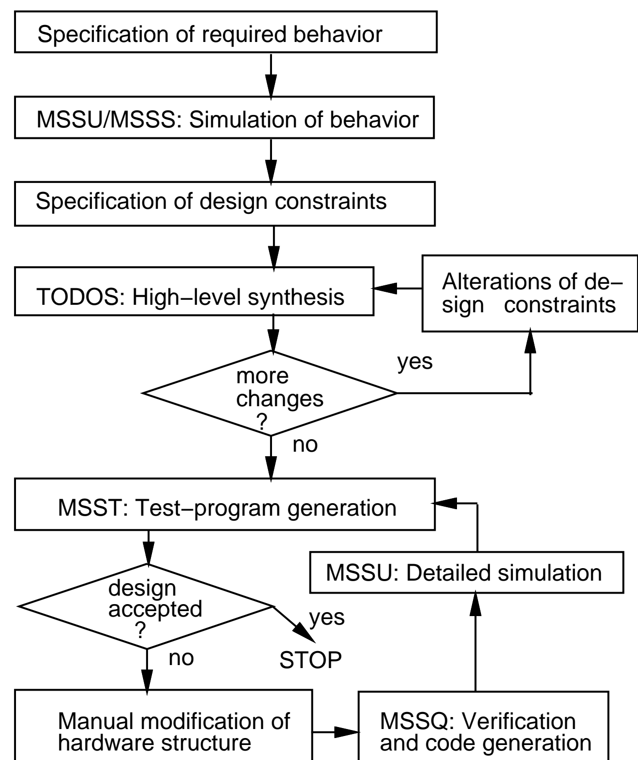

Figure 1: Control flow during a design with the MSS

2 Specification of Required Behavior

The design process starts with the specification of the current unit under design (CUD). This specification describes the required behavior of the CUD in the form of a PASCAL-like program. The program represents either an algorithm at the application level or an instruction interpreter for a given instruction set. In the first case, there will be no typical instruction set in the final design (c.f. fig. 2). The final design will directly implement user applications. For the sake of

*Reprint from latex-source. Copyright of original publication: IEEE Computer Society.

simplicity, a multiplication program with no input and output is used as an example for an algorithm at the application level.

```
PROGRAM Mult IS
VAR  A,
     B,
     C: Integer;
BEGIN
  A := 5;
  B := 7;
  C := 0;
  REPEAT
    C := C + A;
    B := B -1;
  UNTIL B = 0;
END.
```

Figure 2: Application level algorithm

In the last case, a microcoded program will interpret the instruction set (c.f. fig. 3, '#' means hex).

```
PROGRAM Interpreter IS
TYPE word = (15:0); -- bit_array(15:0)
VAR  PC,              -- program counter
     RI   : word;   -- instr. register
     M    : Array [0..#FFFF] OF word;
BEGIN
  PC:=0;
  REPEAT
    RI:=M[PC];
    CASE RI.(15:8) OF -- (15:8)=OpCode
     #00 : ...         --instruction 00
     #01 : ...
     ..
    END;
  UNTIL False;
END;
```

Figure 3: Instruction set interpreter

Throughout this paper, we use version 4.0 of the MIMOLA language, which has been very much influenced by VHDL*.

In order to check the behavior against expected results, the behavior is normally simulated. This simulation does not make reference to hardware structures and hence does not include component timing.

*An interface to VHDL is under construction.

3 High-level synthesis

High-level synthesis converts behavioral specifications into a structural description at the RT-level. Good surveys on high-level synthesis exist [14] and their contents need not be repeated.

In general, there is a large design space to be considered by high-level synthesis tools. However, only designs satisfying additional design constraints will be accepted. The most important design constraint adds some bottom-up information to the top-down design process: our synthesis algorithms assume that only predesigned modules should be used in the final design. This restriction guarantees that only efficient modules will be used. This restriction is especially useful if a library of complex cells (ALUs, memories) is available. The behavior of these cells is also described in MIMOLA, using a non-imperative form of the behavior specification (see fig. 4 for an example).

```
MODULE Alu(IN a,b:(15:0); IN s:(0);
           OUT f:(15:0));
 BEHAVIOR AtRtLevel OF Alu IS
  BEGIN
   f <- CASE s OF -- signal assignment
          0 : a   -  b;
          1 : a   +  b;
        END_case;
  END_behavior;
MODULE TransAlu(IN a,b:(15:0); IN s:(1:0);
                OUT f:(15:0));
 BEHAVIOR AtRtLevel OF TransAlu IS
  BEGIN
   f <- CASE s OF
          0 : a   -  b;
          1 : a   +  b;
          2 : a;     -- transparent mode
          3 : b;
        END_case;
  END_behavior;
```

Figure 4: Behavior of available components

The input to our synthesis algorithms contains the required behavior of the CUD, the behavior of library components and some additional information (like cost-constraints, bindings etc.).

Three synthesis algorithms have been implemented in the MSS environment: the two experimental tools SUCCASS [3, 18] and IN^3 [2] and TODOS (= TOp DOwn Synthesis), a stable tool used for a number of

internal and external designs. TODOS is an extension of the work described in [10, 9].

Synthesis generates structures using some phases. *Early phases* include flow analysis and scheduling. After completion of these phases, the required behavior has been transformed into a set of control steps.

The multiplication program, for example, could have been transformed into the sequence of control steps `CSi` shown in fig. 5. In this sequence, `PARBEGIN` ... `PAREND` encloses assignments to be executed in one control step, `Mem[ ]` denotes a reference to the memory implementing variables and `Reg` and `Rcc` are registers required for storing intermediate values. ``=0'' is a monadic function in MIMOLA.

```
:CS1: PARBEGIN Mem[0] := 5; PC:=CS2 PAREND;
:CS2: PARBEGIN Mem[1] := 7; PC:=CS3 PAREND;
:CS3: PARBEGIN Mem[2] := 0; PC:=CS4 PAREND;
:CS4: PARBEGIN Reg:=Mem[0]; PC:=CS5 PAREND;
:CS5: PARBEGIN
          Rcc := "=0" (Mem[1]-1) ;
          Mem[1] := Mem[1] - 1;
          PC := incr(PC);
      PAREND;
:CS6: PARBEGIN
         Mem[2] := Mem[2] + Reg;
         PC := if  Rcc  then CS7 else CS5
      PAREND;
:CS7: STOP
```

Figure 5: Control steps for multiplication program

The next synthesis phases include resource allocation and binding as well as the generation of the required controller.

Fig. 6 shows a possible solution to the synthesis problem. `Mem.A` and `Mem.B` are memory outputs and inputs, respectively. `I.xxx` denotes signals coming from the controller. The `ALU` is able to add and to subtract. The multiplexer at the input of `PC` has two modes: a transparent mode for input `c` (used for incrementing `PC`) and an IF-mode: `c` is selected if `a` is true, otherwise `b` is selected (this mode is used for ELSE-jumps). Note that neither THEN-jumps nor unconditional jumps can be implemented. The former would require an IF-mode with `c` and `b` reversed, the latter would require a transparent mode for `b`.

Other results can be obtained by running the synthesis algorithms with different inputs. In general, new constraints will be used for additional synthesis runs.

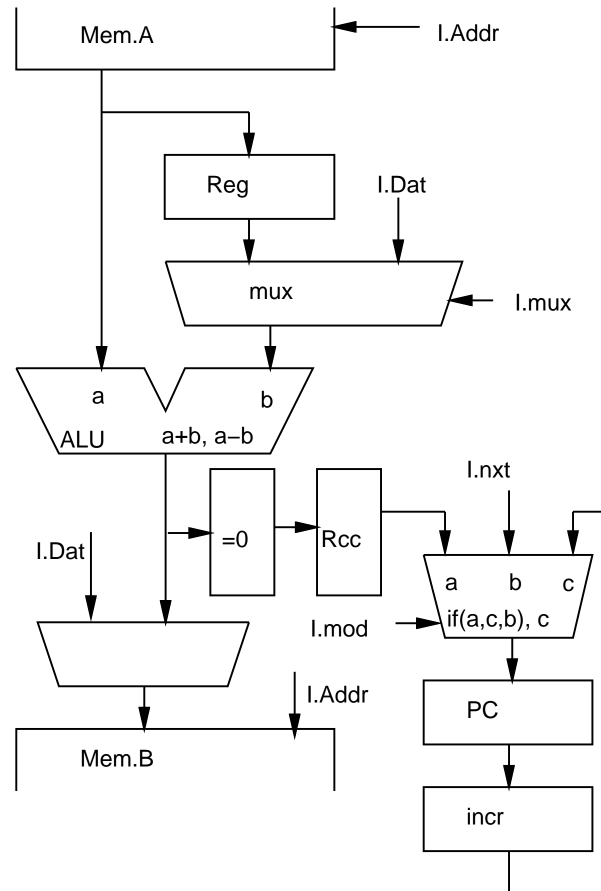

Figure 6: Solution to the synthesis problem

4 Evaluation of testability and generation of self-tests

Frequently, the designs resulting from synthesis are well-testable by self-test programs (programs which are executed on the CUD). However, this is not guaranteed. Addition of built-in self-test hardware is left to the user, because we believe that this should be fully under user-control.

Only few papers have been published on the generation of tests at the RT-level (see e.g. [4]). Our tool MSST [6, 7] is the only one that is integrated into a consistent design environment. MSST primarily is intended to compute the binary code for self-test programs. However, it also generates error messages whenever sections of the hardware cannot be tested by self-test programs.

MSST would, for example, generate a self-test sequence testing `Reg` for stuck-at errors. In the case of a stuck-at error, the generated code would cause a jump to an error report routine (see fig. 7).

```
Mem[0]:=#5555;      --binary 0101.. pattern
Reg:=Mem[0];        --load into register
Rcc:="=0" (Mem[0] - Reg);
PC:=IF Rcc          --check if ok
     THEN incr(PC)  --yes
     ELSE ErrorExit;--jump to error report
Mem[0]:=#AAAA;      --binary 1010.. pattern
Reg:=Mem[0];
Rcc:="=0" (Mem[0] - Reg);
PC:=IF Rcc
     THEN incr PC
     ELSE ErrorExit;
```

Figure 7: Self-test program for Reg

In the case of the condition code register `Rcc`, MSST would detect that `Rcc` can be tested for stuck-at-zero errors (see the sequence in fig. 8). The constant `0` would be generated by first loading it into `Mem`.

```
Rcc:= "=0" (0);  -- try to generate 1 at Rcc
PC:=IF Rcc          --check if ok
     THEN incr(PC)  --no stuck-at-0
     ELSE ErrorExit --jump to error report
```

Figure 8: Self-test program for stuck-at-zero at `Rcc`

During the development of MSST we have assumed that there is a sufficient amount of methods for the computation of test vectors for each of the RT-level components. These patterns can be stored in a test-pattern library. If such a library exists, the test patterns stored in the library will replace the default constants #5555 and #AAAA. The fault-coverage depends on the set of patterns stored in the library.

Our example hardware structure unfortunately cannot be tested for a stuck-at-one at `Rcc`. Such a test would need a THEN-jump, which cannot be implemented by the multiplexer at the input of `PC`.

5 Design Modifications

The user could decide to solve the above problem by extending the jump hardware such that unconditional jumps are possible. Unconditional jumps together with ELSE-jumps can be used to emulate THEN-jumps.

The user could also decide to make additional modifications to the generated hardware. These manual post-synthesis design changes are frequently required in order to overcome some limitations of the synthesis algorithm or e.g. in order to conform to some company standards. Verification of this step will be described in the next section.

In the case of our sample hardware, the user could decide to reroute immediate data to be stored in memory `Mem`. He could omit the multiplexer at the memory input and select an `ALU` that has an transparent mode at least for input `b` (c.f. fig. 9).

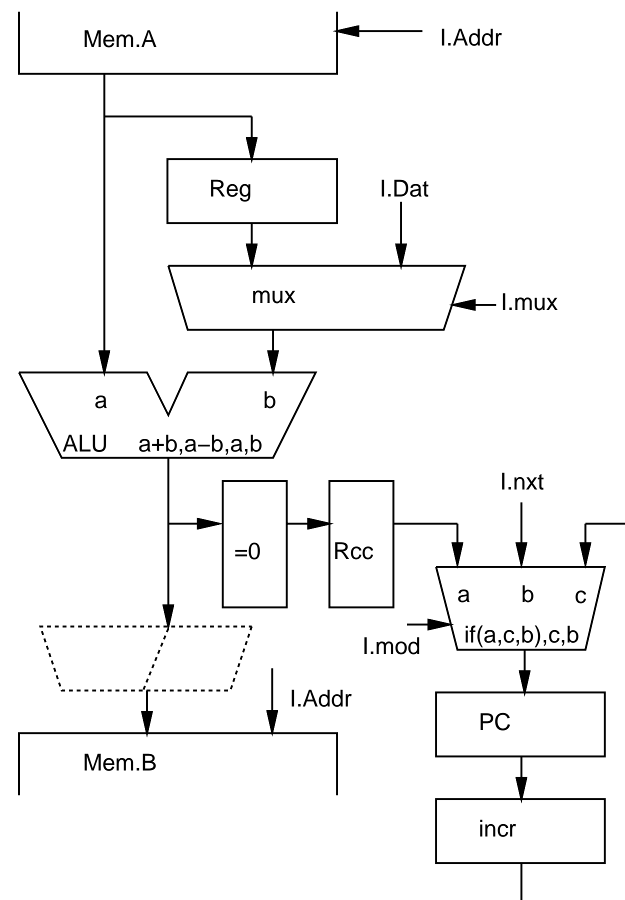

Figure 9: Modified hardware structure

This structure allows the test shown in fig. 10 for a stuck-at-1 at `Rcc`.

Rerouting immediate data possibly influences the resulting performance: parallel assignments to `Mem` and to `Rcc` would now both store the result computed by the `ALU`. The precise effect of this change on the performance depends upon the original program. The MSS allows the user to compute the resulting performance conveniently by using the algorithm mapper MSSQ (see below).

```
Rcc:= "=0" (#5555);-- try to generate 0
PC := IF Rcc THEN incr(PC) --stuck-at-one
            ELSE Cont;    --ok
PC := ErrorExit;   --jump to error report
Cont: <next test>;
```

Figure 10: Self-test program for stuck-at-one at Rcc

6 Retargetable Code Generation

Hardware structures generated by correct synthesis systems are automatically correct ("correctness by construction"). In this context, "correct" means: the structure together with the binary code implement the behavior.

Manually modified structures are potentially incorrect. The MSS allows verifying the design by mapping algorithms to predefined structures. This is done by our retargetable code generator. It tries to translate programs into the binary control code of a given hardware structure. If the binary code can be generated by the compiler, the structure together with the binary code implement the behavior. If no code can be generated by the compiler, then either the structure is incorrect or the compiler does not have enough semantic knowledge. Several methods are used to convey semantic knowledge to the compiler. The most important feature is the fact that the compiler is retargetable. That means: the code for a different machine can be generated by changing the structure being used as input.

Currently, there is a rapidly growing interest in mapping algorithms to predefined structures. *We believe that such mappers will be a necessary extension of high-level synthesis tools.* Only few algorithms have been published in the literature. Most of them are based on retargetable microcode generation. These include the ones designed by Baba [1], Mueller [15] and Mavaddat [13]. The specific advantage of the compilers that have been designed for the MSS is that no part of the code generator needs to be rewritten for new targets. Two compilers have been used:

- MSSV [8, 11], which is based on tree matching. The trees involved are a) expression trees and b) trees representing the hardware structure. In order to get by with tree matching, common subexpressions and fanouts in the hardware result in duplicated nodes. Code generation and microcode compaction (scheduling) are separate phases.
- MSSQ (currently being extended to MAPS = Mapping Algorithms to Predefined Structures) [16, 12], which is based on graph matching. Furthermore, the different phases of code generation are more integrated and code alternatives are better supported in MSSQ. As a result, there are examples for which MSSQ is two orders of magnitudes faster than MSSV.

MSSQ has been used to generate code for different VLIW-machines, including AMD-2900-based designs. MSSQ was the only compiler that was able to handle the benchmark at MICRO-20.

In the particular example of fig. 9, MSSQ will generate the binary code for the modified architecture.

7 Detailed Simulation

This binary code is used for detailed simulation. The simulator loads (see fig. 11) the binary code into the appropiate memories.

This time, simulation includes the simulation of all hardware components, including their timing.

The purpose of detailed simulation is to predict the resulting performance and achieve some additional confidence about the correctness of the design as long as the MSS itself is not formally verified.

For the particular example, another evaluation of the testability will finish the design process.

8 Other Possible Design Sequences

Fig. 1 displays the sequence of design steps in a typical design process. Actually, the control flow of fig. 1 is a special case of allowed sequences. The user is free to select any sequence of actions as long as each tool is supplied with the required input. This means, that the user is only restricted by the data dependence between tools. Fig. 11 shows the data dependence among the main tools of the MSS. Tools for floor-planning have been designed at the University of Kaiserslautern [19].

9 Designs Using the MSS

Designs using the MSS include

- *the design of the asynchronous, microprogrammed SAMP by Nowak [17]*

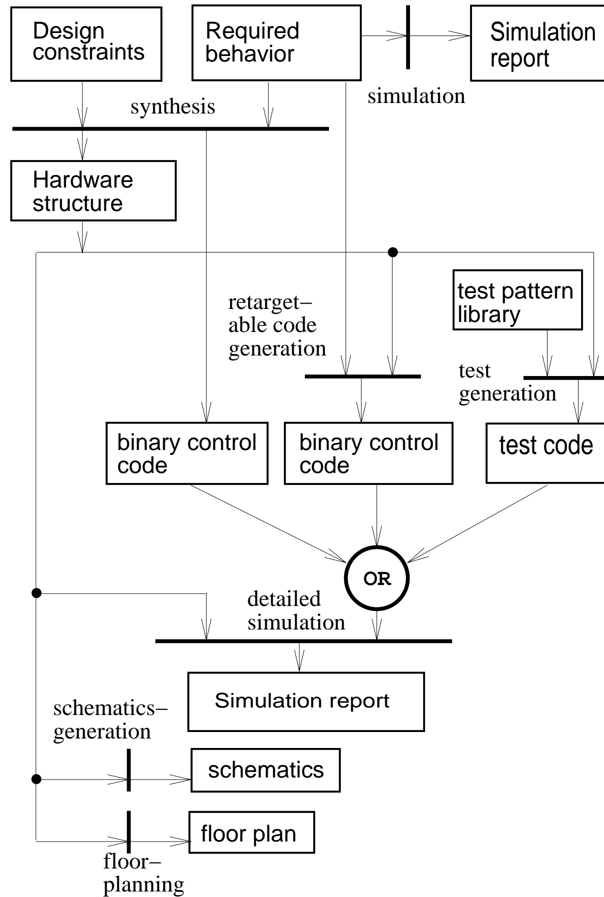

Figure 11: Dependence among MSS tools

- *the design of a CISC-processor at the University of Hamburg*

  In this case, the design was first done manually by directly entering designs into the SOLO-1400 system. Significant effort went into optimizing the design.

  Next, the instruction set was described in MIMOLA. RT-structures were then synthesized with TODOS. Due to the lack of time, only few design iterations were used. The resulting netlist was converted to MODEL, an input language for SOLO-1400 software.

  The synthesized chip was compared to the manual design. According to Rauscher et al. [5], the synthesized chip is only 7 % larger than the manually generated design. Rauscher et al. argue that the size of the synthesized design could be reduced by about 5%. One reason for this success is that the TODOS design uses register files instead of separate registers. SOLO-1400 contains generators producing RAMs with full-custom layout efficiency.

- *The design of the PRIPS PROLOG-machine at the University of Dortmund*

  PRIPS is the result of the student's project "design of PRolog chIPS" the University of Dortmund. PRIPS is a RISC-like processor with special support for the execution of WAM-code (PROLOG intermediate code).

  Early in the project it was realized that it was impossible to implement a virtual WAM machine on one standard cell chip. It was therefore decided to design a RISC-machine with some special features supporting WAM code. The design of this instruction set was based on performance studies, using different versions of a PROLOG to WAM compiler. A MIMOLA specification of the instruction set was then used for synthesis. After a few iterations, modified design constraints did not improve the design. The students therefore started to manually modify the design and to use MSSQ. Manual design modifications included: changes to the tag-ALU, folding of the control fields, removal of cascaded multiplexers, manual addition of self-test logic and changes of the clocking scheme. Due to the fact that all students had no previous experience in hardware design, we followed the principle of *least possible commitment*. We therefore decided to have an on-chip writable control store.

  Actual layout for PRIPS was generated using the Cadence EDGE system (SOLO-2030). Unfortunately, no HDL reader for this system was available to us. The PRIPS design was therefore manually entered using the schematics editor. The resulting layout has been transmitted to EUROCHIP for fabrication at ES2. PRIPS is about 12 × 8 mm large. It is one of the largest chips ever fabricated through EUROCHIP. According to the students who designed the circuit, this size of a design could not have been handled by them without the help of the MIMOLA tools. The MSS will also be used for programming fabricated PRIPS: MSST will generate tests and MSSQ will map algorithms into microcode.

10 Conclusion

The MIMOLA hardware design system supports the entire design process from behavioral specifications to RT-structures. Special emphasis is on the testability

and the correctness of the final design. The correctness is checked both by a formal, automatic procedure and by simulation. Automatic as well as verified manual design steps are used in the system. A key contribution of this paper is to show how current high-level synthesis systems can be extended by tools for mapping algorithms to predefined structures in order to support necessary manual design modifications.

References

[1] T. Baba and H. Hagiwara. The MPG system: A machine-independent microprogram generator. *IEEE Trans. on Computers, Vol. 30*, pages 373–395, 1981.

[2] M. Balakrishnan and P. Marwedel. Integrated scheduling and binding: A synthesis approach for design-space exploration. *Proceedings of the 26th Design Automation Conference*, pages 68–74, 1989.

[3] O. Broß, P. Marwedel, and W. Schenk. Incremental synthesis and support for manual binding in the MIMOLA Hardware Design System. *4th International Workshop on High-Level Synthesis, Kennebunkport*, 1989.

[4] A. Gosh, S. Devadas, and A. R. Newton. Sequential test generation at the register-transfer and logic levels. *27th Design Automation Conference*, pages 580–586, 1990.

[5] N. Hendrich, J. Lohse, and R. Rauscher. Prototyping of microprogrammed VLSI-circuits with MIMOLA and SOLO-1400. *EUROMICRO*, 1992.

[6] G. Krüger. Automatic generation of self-test programs: A new feature of the MIMOLA design system. *23rd Design Automation Conference*, pages 378–384, 1986.

[7] G. Krüger. A tool for hierarchical test generation. *IEEE Trans. on CAD, Vol. 10*, pages 519–524, 1991.

[8] P. Marwedel. A retargetable compiler for a high-level microprogramming language. *ACM Sigmicro Newsletter, Vol. 15*, pages 267–274, 1984.

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

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

[11] P. Marwedel. MSSV: Tree-based mapping of algorithms to predefined structures. Technical Report 431, Computer Science Dpt., University of Dortmund, 1993.

[12] P. Marwedel and L. Nowak. Verification of hardware descriptions by retargetable code generation. *26th Design Automation Conference*, pages 441–447, 1989.

[13] F. Mavaddat. Data-path synthesis as grammar inference. *IFIP-Workshop on Control Dominated Synthesis from A Register Transfer Description, Grenoble*, 1992.

[14] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proc. of the IEEE, Vol. 78*, pages 301–318, 1990.

[15] R.A. Mueller and J. Varghese. Flow graph machine models in microcode synthesis. *17th Ann. Workshop on Microprogramming (MICRO-17)*, pages 159–167, 1983.

[16] L. Nowak. Graph based retargetable microcode compilation in the MIMOLA design system. *20th Annual Workshop on Microprogramming (MICRO-20)*, pages 126–132, 1987.

[17] L. Nowak. SAMP: A general purpose processor based on a self-timed VLIW-structure. *ACM Computer Architecture News, Vol. 15*, pages 32–39, 1987.

[18] W. Schenk. A high-level synthesis algorithm based on area oriented design transformations. *IFIP Working Conference On Logic and Architecture Synthesis, Paris*, 1990.

[19] G. Zimmermann. Top-down design of digital systems. *in: E. Hörbst (ed.): Logic Design and Simulation, Advances in CAD for VLSI, Vol. 2, North Holland*, pages 9–30, 1986.