Verification of Hardware Descriptions by Retargetable Code Generation

Lothar Nowak
Nixdorf Computer AG
Pontanusstr.
D-4790 Paderborn, W. Germany

Peter Marwedel
Institut für Informatik
der Universität Kiel
Olshausenstr. 40-60
D-2300 Kiel, W. Germany

Abstract

This paper proposes a new method for hardware verification. The basic idea is the application of a retargetable compiler as verification tool. A retargetable compiler is able to compile programs into the machine code of a specified hardware (target). If the program is the complete behavioural specification of the target, the compiler can be used to verify that a properly programmed structure implements the behaviour. Methods, algorithms and applications of an existing retargetable compiler are described.

1 Introduction

The correctness of digital systems is relatively neglected by current design systems. It will become even more important, because the systems will become more complex and there will be more critical applications. Simulation, in general, cannot be used for correctness proofs, since the complexity of most systems makes exhaustive simulation impossible.

In the past, synthesis and verification have been proposed as means for the design of correct systems.

Synthesis has received considerable attention, since it is capable of generating correct designs rather quickly and does not require sophisticated training. On the other hand, synthesis algorithms do not fully exploit the capabilities of the implementation technology. Therefore, verification of manual design steps is important whenever there is a strong demand for optimal utilization of resources.

Both synthesis and verification have been used extensively at the geometric, transistor and gate levels. In this paper we consider the process of **verifying that a programmable register transfer (RT-) structure is able to implement a given procedural behaviour**. We do not consider gates and flip-flops. At the RT-level, synthesis is becoming popular, but verification has not yet been used extensively. Exceptions include the work of Uehara et al. [Ueh81], Eveking [Eve81], Milne [Mil86], Takagi [Tak84], Grass [Gra87] and Anceau [Anc87]. Uehara and Eveking check the correctness of a hardware description against a set of assertions. The assertions must be derived manually, a process requiring a skilled staff. Takagi checks structural data path description against its behavioural DDL description (in form of state transitions). The verifier is rule based and looks for validity of operations, data transfers and for resource conflicts. Because of the restriction to the data path the verifier's application is limited and it does not generate binary code.

Binary code, however, is needed because many of the current systems are programmable. That is, the behaviour of the system depends upon the contents of an instruction memory. A programmable system is correct with respect to a high level specification, if and only if the specification can be translated into binary instructions. Equivalence of a programmed hardware structure and a behavioural specification can only be established if the contents of the instruction store is known. Traditional proof systems would use the contents of the instruction store as precondition. The manual derivation of binary instructions, however, is an error-prone process. Therefore, code generation and verification of programmable systems are closely related.

2 Verification by Retargetable Compilation

The compiler integrated in the verifier has to generate code for the target specified by the structural description. Changing this description retargets the compiler and therefore it can be classified as a **retargetable compiler**. In fact, the verifier described in this paper consists of a retargetable compiler (c.f. fig.1), which generates error messages in case it cannot generate code. If code can be generated for the behavioural specifica-

© 1989 ACM 0-89791-310-8/89/0006/0441 $1.50

tion, the structure is correct. This implication cannot be reversed: a fail in code generation might result from insufficient semantic knowledge of the compiler.

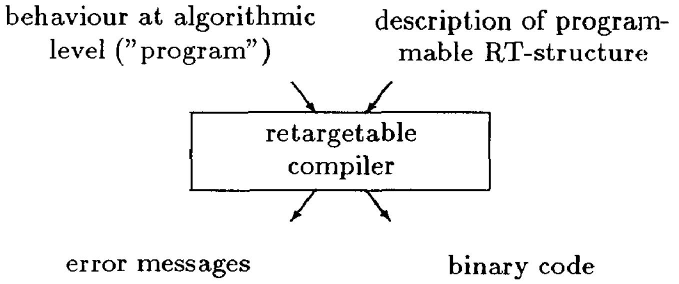

behaviour at algorithmic level ("program")
description of programmable RT-structure
retargetable compiler
error messages
binary code

Figure 1: Inputs and results for a retargetable compiler

The compiler has to generate code (usually referred to as microcode) that will be interpreted by the hardware itself. Available retargetable microcode compilers [Mue83], [Mue84], [Veg82], [Bab81], however, are not suited for verification in a DA environment because they do not accept structural target descriptions. Instead, some manual preprocessing (like selection of paths for register transfers) is required. In a DA environment, there is usually no manpower to perform this task. Also, manual preprocessing is a source of secondary errors and some of the possible solutions may be lost. For verifiers it is absolutely necessary to retain the full set of solutions. Otherwise, there is the risk of rejecting correct designs. The same remarks apply to the use of retargetable compilers (c.f. [Gan82]) for classical machine instructions.

The verifier that comes closest to our approach is the "Formal Checker of Executability" (FORCE) by Anceau [Anc87]. It is intended for verification and it does generate the binary code. It also works with a similar graph representation of the target. FORCE, in contrast to our tool, requires that the user partitions the behaviour into register transfers and control steps.

Our compiler MSSC is designed to work as one of the tools in the MIMOLA hardware design system MSS [Mar84]. In addition, the MSS contains a hardware synthesis tool [Mar86], a self-test program generator [Krü86, Krü88], a simulator and a schematics generator. All the tools are capable of processing designs described in the language MIMOLA. MIMOLA allows the description of behaviour (programs) as well as the description of hardware structures. The program part of MIMOLA spans the range between high level and register transfer level. The high level part of MIMOLA is identical to PASCAL.

For a typical design application, MSSC is normally used for design refinements after synthesizing initial designs (c.f. fig. 2).

Manual modifications of synthesized structures are rather the rule than the exception and therefore the verification is required for reliable designs. Minor modifications are expected even for matured synthesis systems. Without verification, these modifications could result in incorrect designs. The code generator compiles programs into the machine code of a specified target machine. The target specification is given by its MIMOLA structure description. This description contains a list of all register transfer modules, their operations and their interconnections. The compiler uses various tools for preprocessing available in the MIMOLA Design System (e.g. memory allocation, transformation to RT-level, parallelization, branch optimization). The input to the compiler is a preprocessed RT-level program and the target's netlist.

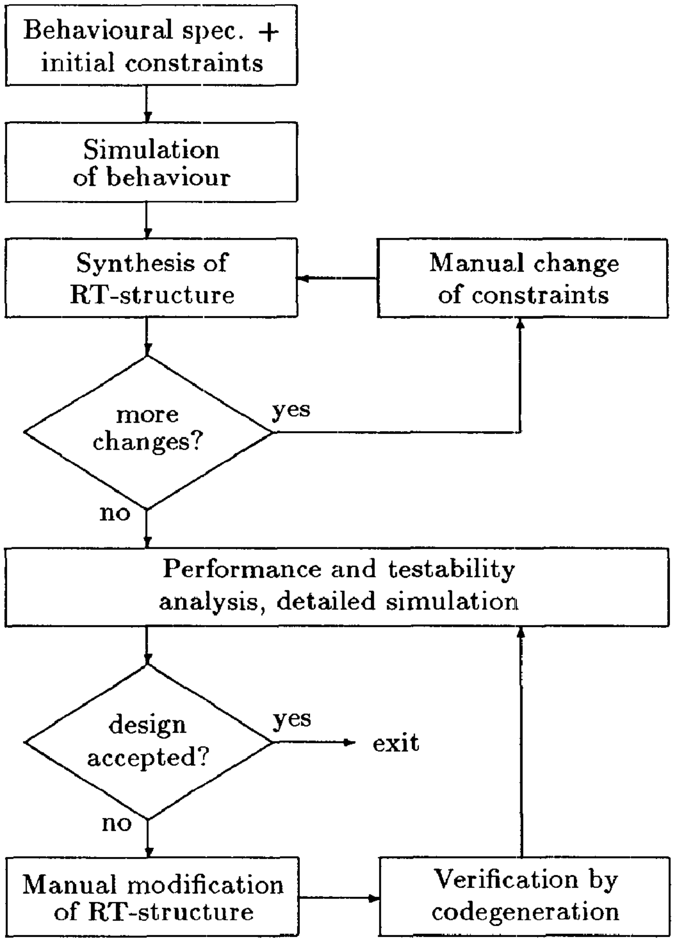

Behavioural spec. + initial constraints
Simulation of behaviour
Synthesis of RT-structure
Manual change of constraints
more changes?
yes
no
Performance and testability analysis, detailed simulation
design accepted?
yes
exit
no
Manual modification of RT-structure
Verification by codegeneration

Figure 2: Control flow during a design project with MSS

With respect to verification, the semantic knowledge of the compiler is important. Our compiler uses different mechanisms to get this knowledge. On the highest level a set of hardware independent transformation rules is applied (e.g. the replacement of repeat-until). This set is accessible to the user and can be extended. Lower level rules can be supplied by the user too (e.g. the rules of De Morgan). On the lowest level the compiler has a fixed knowledge about invariant operation semantics, like commutativity, sign extension, converse operations, neutral elements and constant generation.

3 The Retargetable Microcode Generator

The basic idea used in our compiler MSSC is the mapping of all resource conflicts to instruction field conflicts. This can be done since the selection of modules and their operations is defined (directly or indirectly) by the instruction word. Conflicts for the use of hardware resources are detected by a check for code compatibility. Special treatment is given to resolving programmable bus conflicts.

Experience with an earlier version of the compiler [Mar84a] indicated that an adequate data structure for instruction field settings is the key to a comfortable compilation speed. Therefore the data structure of the current version will be explained in some detail.

3.1 I-Trees: A Representation of Instruction Field Settings

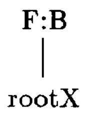
The handling of instruction fields and their conflicts is mapped to the operations of a special Boolean algebra. The elements of this algebra are represented by I-Trees [Now87a]:

An **I-Tree** is a tree structure with the following characteristics:

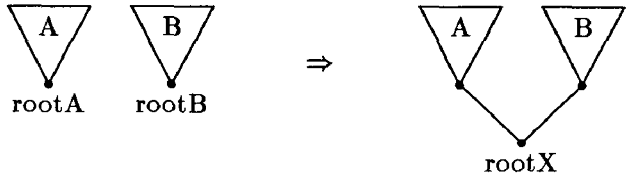
- each node is associated with a field of the instruction word. It contains a bitstring consisting of '0','1' or 'X', which denotes the current field setting.
- nodes on the same path are related by the AND-operation: the field settings are simultaneously valid. If two nodes correspond to the same field, their bitstrings must be compatible.
- neighbouring subtrees indicate alternate instruction word settings (OR-operation, "versions").
- a complete path from the root to a leaf defines one possible instruction word setting. Fields not contained in this path are don't care by definition.
- the root node always contains don't cares and is used only for tree consistency.

The operations defined on I-Trees are:

SET is a mapping Fields × Bitstring → I-Tree. SET sets field F of the instruction word according to bitstring B:

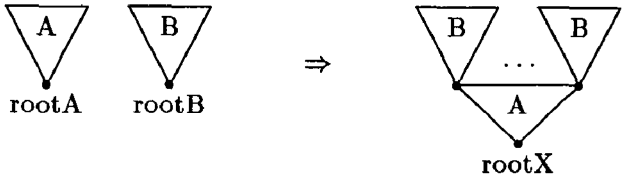


MERGE is a mapping I-Tree × I-Tree → I-Tree. MERGE merges the I-Trees A,B into the I-Tree X (the alternative A or B):



CUT is a mapping I-Tree × I-Tree → {I-Tree,{}}. CUT maps two I-Trees A,B into the I-Tree X (the cut of A and B). The result can be empty.

CUT is implemented by two steps. Using the distributive law, B is first copied to all leaves of A. Then all incompatible paths are deleted. A path is said to be *incompatible* if it contains conflicting field settings.



These operations can be used to preset instruction fields (SET), test for resource conflicts (CUT) or generate different versions (MERGE) in case of some alternative choices. If for example an ALU-operation is directly controlled by an instruction field alu_ct, the addition will be executed if the instruction is set to SET(alu_ct,add_code). The result can be stored in register REG with load-control field reg_ct if CUT(SET(reg_ct,load_code),SET(alu_ct,add_code)) is not void, even if the fields reg_ct and alu_ct intersect. The MERGE-operation has to be applied if, e.g. the addition can be selected by two different control codes: the second I-Tree in the CUT-operation above has to be replaced by MERGE(SET(alu_ct,add_code1),SET(alu_ct,add_code2)).

These examples show how module activations and combinations of them can be expressed by I-Trees. Even the activation of complete microorders can be represented by I-Trees. Resource compatibility can be checked by the CUT-operation again.

3.2 Compilation Phases

The compilation process is devided into three phases: preallocation, assignment allocation and microorder scheduling.

3.2.1 Preallocation

The **preallocation phase** generates the Connection-Operation-Graph (CO-Graph). This internal data structure is derived from the MIMOLA hardware description of the target architecture. Such description is shown in fig. 3 (using MIMOLA version 4.0). It is the description of a processor containing an ALU, two data registers, a multiplexer, a program counter and a control store.

```
MODULE Processor (OUT res:(15:0);
                  IN ClockIn:(0));
 STRUCTURE AtRtLevel OF Processor IS
  TYPE
   word = (15:0);          (*16 bits per word  *)
   Instr=FIELDS            (*instruction format*)
         (*absolute positions of control fields*)
          Alu : (1:0);  Mux : (2);
          R0  : (3);    R1  : (4);
          R2  : (5);    Imm : (21:6);
          NextAddr : (37:22);
         END;
  PARTS                    (*components        *)
   Alu : MODULE AluT(IN i1,i2: word;
                     OUT outp: word;
                     FCT ct: (1:0));
          BEHAVIOUR AtRtLevel OF AluT IS
           BEGIN
            CASE ct OF
             %00 :  outp <- i1 + i2   AFTER 10 ;
             %01 :  outp <- i2 - i1   AFTER 10 ;
             %10 :  outp <- i1        AFTER  5 ;
           END;  END;
   Mux : MODULE M2x16(IN i1,i2: word;
                      OUT outp: word;
                      FCT ct: (0));
          BEHAVIOUR AtRtLevel OF M2x16 IS
           BEGIN CASE ct OF
            %0 :  outp <- i1 AFTER 5;
            %1 :  outp <- i2 AFTER 5;
           END; END;
   R0,R1,R2:
         MODULE R16(IN i: word; OUT outp:word;
                    FCT ct:(0);CLK c:(0));
          BEHAVIOUR AtRtLevel OF R16 IS
           VAR cell : word;
           CONBEGIN
            AT c DO CASE ct OF
                     %0 :  cell := i;
                     %1 :  ;
                    END;
            outp <- cell;
           CONEND;
   I: MODULE EPROM(ADR a:word; OUT f:Instr);
          BEHAVIOUR AtRtLevel OF EPROM IS
           VAR
            cells : ARRAY [0..(2**16)-1] OF Instr;
           BEGIN
            f <- cells[a] AFTER 7;
           END;
  CONNECTIONS
   Mux.i1 <- I.f.Imm;   Mux.i2 <- R0;
   Mux.ct <- I.f.Mux;   Alu.ct <- I.f.Alu;
   Alu.i1 <- Mux.outp;  Alu.i2 <- R1.outp;
   R0.ct  <- I.f.R0;    R0.i   <- Alu.outp;
   R1.ct  <- I.f.R1;    R1.i   <- Alu.outp;
   R0.c   <- ClockIn;   R1.c   <- ClockIn;
   R2.c   <- ClockIn;   R2.i   <- I.f.NextAddr;
   R2.ct  <- I.f.R2;    res    <- R0.outp;
   I.a    <- R2.outp;
 END_structure_of_processor;
RESERVED Space OF Processor IS
  FOR ProgramCounter USE R2;
  FOR Instructions   USE I;
 END_Reserved;
```

Figure 3: MIMOLA description of a simple processor

Descriptions of processor components are first mapped into an **M-graph**. The root and the leaves of M-graphs represent the output and the input of the component, respectively. In between there is one tree of depth 2 for every operation that is listed in the MIMOLA description (c.f. fig. 4). "dat" is the identity operation in MIMOLA.

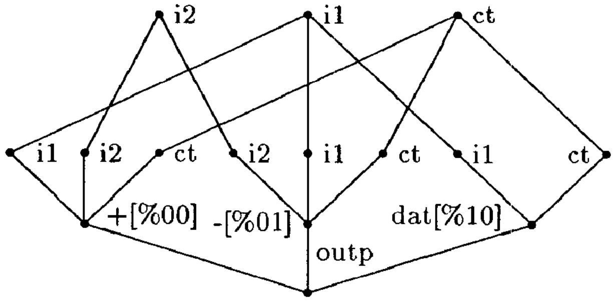


Figure 4: M-graph for component "Alu"

Next, the M-graphs are combined to form the **CO-graph**, representing the complete hardware structure. Registers are divided into two M-graphs, representing all load and all read operations, respectively. Fig. 5 shows the CO-graph for the example of fig. 3. In this figure, the internals of all M-graphs corresponding to each node of the CO-graph have been omitted.

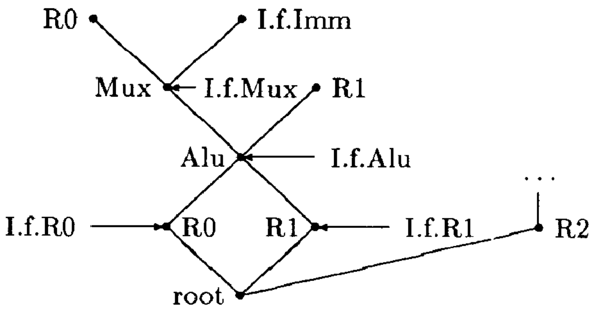


Figure 5: Hardware structure and corresponding CO-graph

In the next step, local transformations are applied to the CO-graph. For example, additional paths are generated for commutative operations and for so-called **via-operations**. A via-operation is an operation which can be used for propagating values from an input to the output of a component. One possibility is to create nodes for via-operations by using operations with neutral elements at some other input. Fig. 6 contains such an entry for 0 as the neutral element of +.

In the following, the requirement to supply a certain value at the input of a hardware component is called

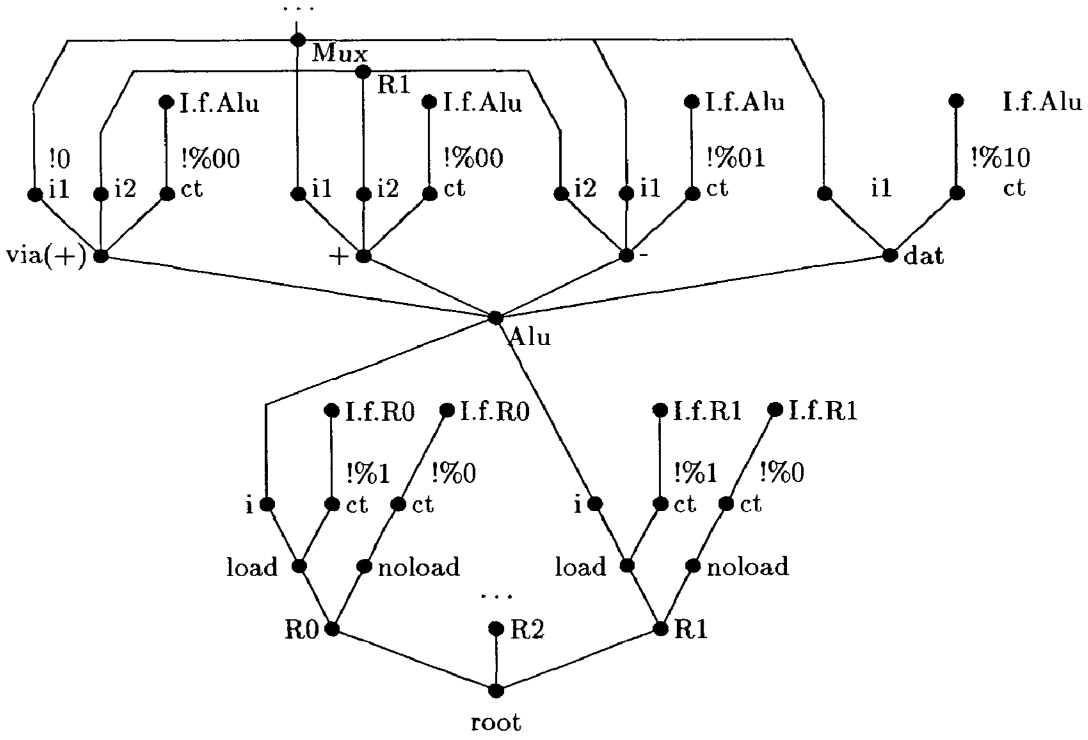


Figure 6: CO-graph with preconditions (% denotes binary numbers)

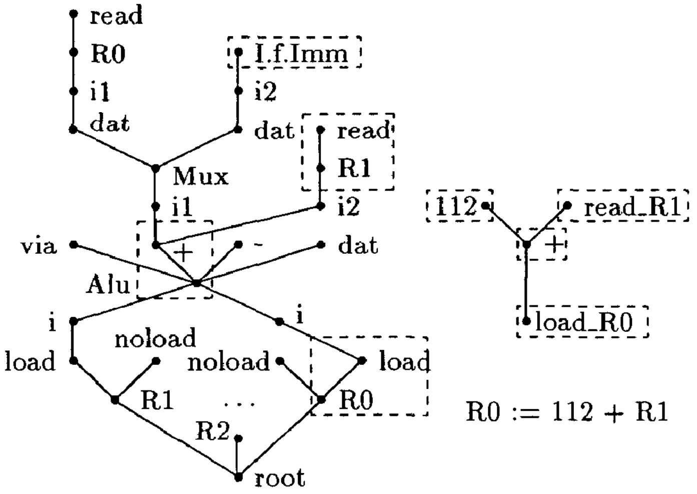
a **precondition**. In the CO-graph, preconditions are depicted by an exclamation mark. Preconditions are constants, which have to be generated by the hardware. The most common way of generating such a constant is by appropriate values in the instruction register, that is, by an appropriate binary program. However, constants can also be generated indirectly by other hardware components.

Most preconditions are met by appropriate instruction field settings. These settings are represented by I-Trees that are linked to the nodes of the CO-graph.

3.2.2 Code generation

Code generation for MIMOLA programs is done by a pattern matching process. During this process, program flow trees are compared with the CO-graph. The code generation allocation algorithm searches through the CO-Graph for subtrees which are equivalent to the dataflow trees defined by the assignment. Equivalence means a one to one correspondence between operations in the program and nodes in the M-graph ignoring some redundant M-graph nodes like multiplexers and input nodes (nodes labelled i, i1 and i2). Further it is required that all the arguments are present in the same order.

An example of two matching graphs is shown in fig. 7. The required procedural behaviour in this case is "R0:=112+R1". The two graphs match and therefore verification succeeds. The resulting I-Tree is the cut of all I-Trees found in the CO-Graph subtree. If alternative ways of matching the graphs exit, all alternatives will be expressed as an OR-list. Hence, the full generality of possible instruction bit settings is retained. This breadth-first-search technique avoids backtracking for single binary instructions.



CUT(itree(load_R0),itree(dat_Mux),itree(+_Alu), SET(I.f.Imm,112),itree(read_R1))

Figure 7: Allocation of an assignment

If the allocation fails (void I-Tree), the algorithm returns the reason and the probable error location. This information is used by MSSC to choose a proper location for a temporary variable. The assignment is split and the allocation continues recursively. If R1 were available as a temporary register, MSSC would split the statement

```
R0:=R0+5
```

into two statements which can be implemented on the hardware:

```
R1:=R0  ;
R0:=R1+5;
```

Hence in this case verification would succeed.

In contrast, verification for "R0:=(R0-5)+R1" would fail due to a lack of temporary registers.

Insertion of temporary registers is possibly the most complex task of MSSC. It depends on several heuristics for defining proper temporary storage locations and temporary values. Backtracking is used if the allocation fails for a certain location or value. Therefore, a type of depth-first-search is used for the binding of temporaries.

In practice, behavioural descriptions are larger than the simple examples. Frequently, they consist of a complete instruction set interpreter and they make reference to i/o-ports.

The result of the allocation phase is a list of microorders (assignments of values to single storage units) represented by I-trees.

3.2.3 Scheduling

In the **scheduling phase** microorders have to be assigned to microinstructions (control store words). An assignment of a set M (m_1,..,m_n) of microorders to a microinstruction I is valid, if and only if:

1. the partial ordering defined by datadependencies and data-antidependencies is preserved,
2. all microorders are compatible:
itree(M)=CUT(itree(m_1),..,itree(m_n)) is not void,
3. all unused memories and registers (denoted by noop(M)) are disabled:
itree(I)=CUT(itree(M),itree(noop(M)) is not void.

The result of the scheduling phase is a list of all packed instructions. Due to the I-Tree representation the algorithm handles versions of microorders at once. Even the resulting instructions are represented by I-Trees, offering the chance to choose the best version out of all possible ones.

4 Limitations and Results

MSSC is implemented by three separate Standard-PASCAL programs containing a total of about 34,000 lines of code.

The current implementation has the following limitations:

- Instructions must have a fixed wordlength.
- Delayed branches are not yet supported.
- Asynchronous hardware components (e.g. multi-cycle read operations) are not yet supported.

The last restriction can be bypassed by adding virtual memory buffers to the hardware description. All three restrictions can be eliminated by extending MSSC without violating the principles of this paper.

MSSC as well as the previous compiler MSSV [Mar84a] have been applied to real design problems. These include:

- One example is the SAMP processor, containing 4 ALUs, two 4-port memories and a horizontal instruction format [Now87]. An old version [Zim80] of the synthesis system was used to generate an initial hardware structure. It has been possible to reduce the number of interconnections between hardware components by about 50% and to verify with MSSV, that the specification was still met. Recently, MSSC was used to implement Warren's abstract PROLOG-machine (WAM) on the SAMP [Sch88]. This was done by writing an interpreter for the WAM in MIMOLA and compiling this interpreter into machine code for the SAMP. For a set of 8 benchmark programs the SAMP is 1.1 times faster than a WAM implemented in VAX8600 microcode. This performance could only be achieved because MSSC generates rather compact code.
- The applicability of the hardware model was demonstrated at a contest at the 20th Annual Workshop on Microprogramming. MSSC was the only tool that could handle the benchmark hardware.
- A separate group of the university of Kiel is currently using the MSS to design a reduction machine. Automatic synthesis generated a machine with 4346 wires at the RT-level. Using MSSC, it has been possible to reduce the number of wires to 3319 without affecting the performance. Typically, MSSC compiles the 1248 statements of this example into 780 binary instructions with a total of 12,237 versions, using about 1100 seconds of Apollo DN-3500 CPU-Time. This indicates that versions are an important concept. The compilation speed is much lower than that of traditional compilers but acceptable for the intended applications.

5 Conclusion

Equivalence of a programmed hardware structure and a behavioural specification can only be established if the contents of the instruction store is known. Compilers are capable of generating binary instructions fast and reliably and it is therefore suggested to use compilers for verification. The application requires that retargetable compilers accepting structural hardware descriptions are used. This paper presents such a compiler.

In general, the implementation of behavioural operations in a given hardware is not unique. Hence, the support of implementation versions is essential. The presented compiler uses a compact representation of versions. This compact representation allows retaining the full generality of solutions during the first phases of the compilation procedure by using breadth-first-search. Backtracking is employed for the last phases of the procedure in order to avoid rejection of correct designs.

The compiler has successfully generated code for various complex processors and has been used for verifying design refinements of synthesized structures.

References

[Anc86] F. Anceau: FORCE: A Formal Checker for Executability, in: D. Borrione (ed.): From HDL Descriptions to Guaranteed Correct Circuit Designs, Proc. of IFIP WG 10.2 Working Conf., North Holland, 1987

[Bab81] T. Baba, H. Hagiwara: The MPG System: A Machine-Independent Efficient Microprogram Generator, IEEE Trans.Comp., Vol. C-30, 6(1981), pp. 373-395

[Eve81] H. Eveking: The Application of CONLAN Assertions to the Correct Description of Hardware, in: M. Breuer, R. Hartenstein (eds.): Proc. 5th IFIP Conf. on Computer Hardware Description Languages and Their Applications, North Holland, 1981, pp. 37-50

[Gan82] M. Ganapathi, C.N. Fisher, J.L. Hennessy: Retargetable Compiler Code Generation, ACM Computing Surveys, Vol. 14, 4(1982), pp. 573-593

[Gra87] W. Grass, R. Rauscher: CAMILOD: A Program System for Designing Digital Hardware with Proven Correctness, in: D. Borrione (ed.): From HDL Descriptions to Guaranteed Correct Circuit Designs, Proc. of IFIP WG 10.2 Working Conf., North Holland, 1987

[Krü86] G. Krüger: Automatic Generation of Self-Test Programs - A New feature of the MIMOLA Design System, Proc. 23rd Design Automation Conference, 1986, pp. 378-384

[Krü88] G. Krüger: A Tool For Hierarchical Test Generation, Proc. ICCAD, 1988

[Mar84] P. Marwedel: The MIMOLA Design System: Tools for the Design of Digital Processors, 21st Design Automation Conference, 1984, pp. 587-593

[Mar84a] P. Marwedel: A Retargetable Compiler For A High-Level Microprogramming Language, 17th Annual Workshop on Microprogramming (MICRO-17), 1984, pp.267-274

[Mar86] P. Marwedel: A New Synthesis Algorithm for the MIMOLA Software System, Proc. 23rd Design Automation Conference, 1986, pp. 271-277

[Mil86] G. Milne: Towards verifiably correct VLSI design, in: Milne, Subrahmanyam (eds.): Formals Aspects of VLSI Design, Proc. Workshop on VLSI, North Holland, 1986

[Mue83] R.A. Mueller, J. Varghese : Flow Graph Machine Models in Microcode Synthesis, 16th Annual Microprogramming Workshop (MICRO-16), 1983, pp. 159-167

[Mue84] R.A. Mueller, J. Varghese, V.H. Allan : Global Methods in the Flow Graph Approach to Retargetable Microcode Generation, 17th Annual Microprogramming Workshop (MICRO-17), 1984, pp. 275-284

[Now87] L.Nowak : SAMP: A General Purpose Processor Based on a Self-Timed VLIW Structure, ACM Comp. Arch. News, Vol.15, No.4, Sep. 1987, pp. 32-39

[Now87a] L.Nowak : Graph Based Retargetable Microcode Compilation in the MIMOLA Design System, Proc. 20th Annual Workshop on Microprogramming (MICRO-20), Dec.1987, pp. 126-132

[Sch88] W. Schenk: Eine Prologimplementierung für einen Rechner sehr großer Befehlsbreite, master's thesis, Institut für Informatik und Praktische Mathematik der Universität Kiel, 1988

[Tak84] S. Takagi : Rule Based Synthesis, Verification and Compensation of Data Paths, Proc. IEEE Conf.Comp.Design (ICCD'84), New York, Oct.1984, pp. 133-138

[Ueh81] T. Uehara, F. Maruyama, T. Saito, N. Kawato : DDL Verifier, in: M.Breuer, R.Hartenstein (eds.): Proc. 5th IFIP Conf. on Computer Hardware Description Languages and Their Applications, North Holland, 1981, pp. 51-61

[Veg82] S.R. Vegdahl: Local Code Generation and Compaction in Optimizing Microcode Compilers, PhD thesis and report CMUCS-82-153, Carnegie-Mellon University, Pittsburgh, 1982

[Zim80] G. Zimmermann: MDS - The MIMOLA Design Method, Journal of Digital Systems, Vol.4, 3(1980), pp. 337-369