A RETARGETABLE COMPILER FOR A HIGH-LEVEL MICROPROGRAMMING LANGUAGE

PETER MARWEDEL

Institut für Informatik und Praktische Mathematik
Universität Kiel, D-2300 Kiel, W. Germany

ABSTRACT

A compiler for the generation of microcode for a high-level microprogramming language is presented. The compiler is target machine independent. The input to the compiler consists of a hardware description, a high-level microprogram and a set of program transformation rules. The compiler is able to take advantage of optimization techniques which are used by microprogrammers because many of these can be represented by program transformation rules.

1. INTRODUCTION

During the recent years, microprogramming has become increasingly important. As processors become more complex, they tend to be microprogrammed. Most mainframes and larger microprocessors are now microprogrammed. The AMD 2900 series of chips has made microprogramming more popular. Sometimes, conventional machine instructions are never implemented for AMD-based designs. Microprogramming is the only programming done in these cases.

As a result of the increased use of microprogramming, there is an increased interest in tools for microprogramming. On the other hand, there is a lack of tools for microprogramming. Most microprogrammers still use assemblers. What are the reasons for not using microcode compilers?

There are some obvious problems in the construction of microcode compilers:

- In some cases timing has to be considered.
- The inherent parallelism complicates code generation.
- Many of the available microarchitectures are not easy to program because ease of programming has not been a design goal.
- There may be a mismatch between the data types of the target and those of the programming language.
- For most applications extremely efficient code is required.
- There are many more microarchitectures than there are machine architectures (each model of a machine may have its own microarchitecture). Only a few programs are written for each architecture. Therefore, the design of a separate compiler for each microarchitecture is too expensive.
- "Microcode development is at an early part of the critical path for a processor development project, therefore waiting until a compiler is developed would lengthen the critical path delaying the project"[1].

Retargetable microcode compilers have been recognized as one solution to the last two problems. The input language for such a compiler should be as machine independent as possible. Because of the need for highly efficient code, it is desirable also to allow machine dependent code sequences.

One attempt to design a suitable input language resulted in the design of the language schema S^* [2]. With this approach, a new member of a family of languages is designed for each new target machine.

The approach described in this paper is different; namely, a language is defined that may be extended (cf. deWitt[3]) to include target dependent operations. This language is called MIMOLA (machine independent microprogramming language). This language and a retargetable compiler allows combining the virtues of the two approaches to high level microprogramming which have been suggested by Davidson[1]:

1. machine dependent code generators for machine independent languages and
2. retargetable code generator and a range of machine dependent languages.

CH2088/84/0000/0267$01.00©1984IEEE

2. MIMOLA HARDWARE DESCRIPTION

For a retargetable compiler, a description of the target architecture is needed. In order to simplify the compiler, the same language is used for hardware descriptions and the description of algorithms. MIMOLA has been designed as a computer hardware description language (CHDL) and a high level microprogramming language (HLML). Hardware descriptions use a subset of the language.

The hardware model underlying MIMOLA hardware descriptions has been dictated by the use of MIMOLA in a hardware design system, the MIMOLA Software System (MSS). Because of this application of the language, the hardware model describes real hardware, i.e. hardware modules and data paths. For some applications this structural description level may be lower than necessary. However, it simplifies the generation of hardware descriptions using hardware block diagrams. It is not required to generate functional level descriptions (i.e. instruction set level descriptions).

Example:

Figure 1 represents a slightly simplified block diagram of an AMD 2901 bit-slice[4] :

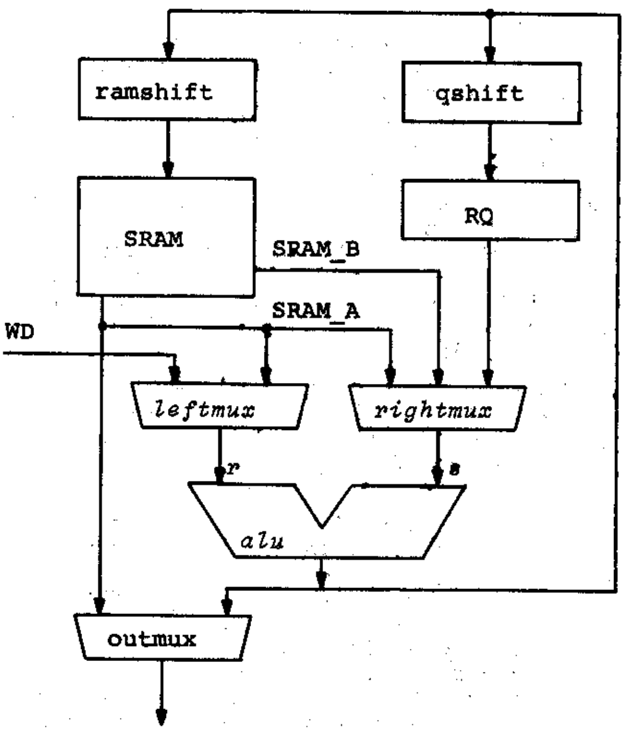


Figure 1. AMD 2901 bitslice (simplified)

Figure 2 represents the functional description of the ALU and the ALU source selection :

code	ALU function
0	r + s
1	s - r
2	r - s
3	r OR s
4	r AND s
5	r' AND s
6	r ⊕ s
7	NOT(r ⊕ s)

code	ALU source left	ALU source right
0	SRAM_A	RQ
1	SRAM_A	SRAM_B
2	0	RQ
3	0	SRAM_B
4	0	SRAM_A
5	WD	SRAM_A
6	WD	RQ
7	WD	0

Figure 2. ALU function and source control

Figure 3 explains how the above information may be represented in MIMOLA :

```
TARGET example;
 TYPE nibble = BIT(3:0);
  MODULE BALU(in r,s : nibble; in sel : BIT(2:0);
              out f:nibble);
   BEGIN
    f:= CASE sel OF
         0 : r "+" s;      1 : s "-" r;
         2 : r "-" s;      3 : r "OR" s;
         4 : r "AND" s;    5 : r "a'ANDb" s;
         6 : r "XOR" s;    7 : r "EQU" s
         END
   END;

  MODULE N2mux(in dat0, dat1 : nibble;
               in sel : BIT(2:0); out p : nibble);
   BEGIN
    p:= CASE sel OF
         0,1 : dat1; 2,3,4 : 0; 5,6,7 : dat0
         END
   END;

  MODULE N3mux(in dat0,dat1,dat2 : nibble;
               in sel : BIT(2:0); out p : nibble);
   BEGIN
    p:= CASE sel OF
         0,2,6 : dat2;   1,3 : dat1;
         4,5   : dat0;   7   : 0
         END
   END;

 ...
 PARTS
  alu      : BALU;
  leftmux  : N2mux;
  rightmux : N3mux;
 ...
 CONNECTIONS     (* structure information *)
  leftmux.p  -> alu.r ;
  rightmux.p -> alu.s ;
 ...

END_target ;
```

Figure 3. Hardware Description Example

The inclusion of data paths in the hardware declaration is required because they represent resources. The absence of a data path often explains why certain microoperations are illegal.

Strings enclosed in double quotes are operators. In addition to predefined operators like "+" and "*", non-standard operators can be defined. This extension mechanism allows adding target dependent operators to the language without changing the syntax or the compiler. A certain operator may be used by a program if either this operator is implemented in hardware or if the operator may be replaced by a combination of hardware-implemented operators (replacement rules are explained below). Therefore, machine dependent programs can take advantage of special purpose hardware.

The meaning of non-standard operators has to be declared. This declaration resembles function declarations in ADA[5].

Example:

```
OPERATION "SHIFTLC" (i : BIT(15:0)): BIT(15:0);
 BEGIN
  RETURN i.BIT(14:0) ! i.BIT(15)
  (* Exclamation marks denote concatenations *)
 END;
```

The body of *OPERATION* declarations is evaluated during simulations. The code generator uses only few properties (e.g. existence of neutral elements). These properties can be defined by the user.

The following convention holds for the MIMOLA language : Identifiers of hardware modules start with capital letters. Some characters denote special module types :

A:	unary function box
B:	binary function box
N:	general function box
I:	current instruction
R:	register
S:	random access storage
W:	bus (wire)

3. MIMOLA ALGORITHM DESCRIPTION

3.1 High-Level Language

The reasons for the design of high-level microprogramming languages include those reasons which led to the development of high-level languages (HLLs) and need not be repeated. Consequently, an earlier definition of MIMOLA[6] has been extended to include most of PASCAL's high-level language elements[7].

3.1.1 Data Types

In contrast to PASCAL, MIMOLA is not a strongly typed language because for microprogramming this would be too restrictive. The basic data type of MIMOLA is the bitstring of arbitrary length. Other data types may be declared as subtypes of this type, e.g.

```
TYPE  INTEGER = BIT(31:0);
```

This declaration declares the type *INTEGER* as a string of 32 bits. It is assumed that 32 bits fit into one memory word. MIMOLA record declarations provide a means for the declaration of multi-word data types.

Declarations like these will typically be part of the declarations at the outermost declaration level. Changes of the representation of integers therefore are confined to a single declaration list at the beginning of the program.

3.1.2 Variables

Automatic storage allocation for high-level language variables is desirable. It is required if the input program is to be independent of the target. The programmer should have the choice between static and dynamic (runtime-stack) allocation of variables.

On the other hand, performance requirements do not always allow this approach and manual storage allocation may be required. Therefore, MIMOLA (as well as some systems implementation languages (SILs)) allows automatic and manual storage allocation. If a variable is to be allocated automatically, just its type has to be declared, e.g.:

```
VAR a: INTEGER;
```

For manually allocated variables the memory name and the address are included in the declaration, e.g.:

```
VAR b: SM(15).INTEGER; (*SM: memory; 15:address *)
```

For procedures, locations for parameter passing may be declared, e.g.:

```
PROCEDURE fac(c:Reg_1.INTEGER;
              REF r: SM(Reg_2).INTEGER);
```

In this example the first parameter is a value parameter and is passed in a register. The second parameter is a reference parameter. The address is passed in register Reg_2.

3.1.3 Control Flow

Required high level-control constructs such as procedure calls, for- and while-loops have been included in MIMOLA. MIMOLA allows the programmer to specify parallel and sequential execution of statements: separation of statements by a semicolon indicates sequential execution, separation by a comma indicates parallel execution. The sequencing specified by the programmer is not altered unless resource conflicts occur or the optional detection

of parallelism is used. Therefore, the programmer is able to write low-level microprograms.
HLMLs have the property that the programmer does not have to care about sequencing. The MIMOLA compiler frees the programmer from this task because it is able to override programmer-specified sequencing :

1. A compiler option is available which causes the compiler to detect sequences of statements which (from an algorithm's viewpoint) could be executed in parallel. This transformation is target machine independent.
   Example: The compiler would replace the semicolon in the sequence *r:=s; c:=p+2* by a comma (Except if *r* or *c* are formal reference variables).
2. If resource conflicts occur during hardware allocation for parallel blocks, commas are replaced by semicolons. If resource conflicts occur during hardware allocation for single statements, assignments to temporary locations are generated. Available temporary locations have to be declared.
   Example: *RESERVED_TEMPORARY SM(3:0);*
   This declaration is either a part of the hardware description or of the declarations in the program.

3.1.4 Supplementary Information

Some applications (some targets) require that additional information can be added to the algorithms without changing the syntax of the language. In MIMOLA two methods are available:
1. Property lists (items enclosed in angle brackets)
   Example:

   IF .. THEN <WEIGHT=0.3>..

   (* define estimated execution probability *)
2. Reserved words may be followed by "extensions" starting with an underline character.
   Examples:

   RETURN_FROM_INTERRUPT, CALL_FORTRAN-EXTERNAL GOTO_LOCAL, RESERVED_TEMPORARY, END_CASE

3.1.5 Programming Example

Last year, Davidson compared several microprogramming languages[1]. A multiplication procedure served as an example. The following program contains the MIMOLA version of this procedure:

```
PROGRAM callmultiply;
(* insert target description here *)
DECLARE
 VAR p,q,r : BIT(63:0);
 PROCEDURE multiply(a,b: BIT(63:0);
                    REF c : BIT(63::0));
 DECLARE
  VAR mcand,mier,result : BIT(63:0);
  BEGIN
   mier :=b;
   result:=((mier "AND" %100..0) "+" a) "AND" %100..0;
   mier :=mier "AND" %011..1;
   mcand := a "AND" %011..1;
   WHILE mier "< >" 0 DO
    IF mier.BIT(0)
     THEN result:= result "+" mcand FI;
    mier:= "SHIFTRL" mier;
    mcand:="SHIFTLL" mcand
   OD;
   c:=result
  END;
 BEGIN
  p:=7; q:=15; CALL multiply(p,q,r)
 END.
```

Assume that *Reg_x, Reg_y* and *Reg_z* are registers declared in the target description. Variables mier, mcand and result are bound to these registers if their declaration is replaced by

```
VAR  mcand : Reg_x.BIT(63:0),
     mier  : Reg_y.BIT(63:0),
     result: Reg_z.BIT(63:0);
```

3.2 Low-Level Language

It is commonly accepted, that certain critical code sections have to be manually bound in order to generate efficient microprograms. This is also possible with MIMOLA. Manual binding of variables and sequencing have already been described. Manual binding of operations is possible with a modified functional programming style.

Example:

x:= BALU(a,b,"+")(BALU is a hardware resource *)*

On some occasions, microcode bits must be set explicitly by the programmer. The letter *I* represents the current instruction and can be used for this purpose.

Example:

I(%1).BIT(63) (Set bit 63 to 1 *)*

With these features, it is possible to specify an algorithm in high-level MIMOLA and hand-translate critical code sections.

4. REPLACEMENT RULES

Implicitly compilers make use of program transformations in order to bind programs to a certain hardware. For machine-dependent compilers these rules are not made explicit. They are built into the compiler ('hardwired-software'). Since many of these transformations are target-dependent, this approach is unacceptable for a retargetable compiler. The target dependent subset of these rules must be made explicit. Therefore, transformation rules may be defined in MIMOLA. Transformation rules are used for the following transformations:

1. Replacement of high-level constructs by a set of equivalent register-transfer (RT) level statements.
   Example:

   ```
   REPLACE goto L&a (* L&a matches all labels *)
   WITH  RP:= L&a (* RP is the program counter *)
   END
   ```

2. Replacement of unimplemented operations by a set of implemented operations.
   Example (identifiers starting with & are parameters):

   ```
   REPLACE &a   "<" &b
   WITH "<0"(&a "-" &b)
   END    (* "<0" is a unary operation *)
   ```

3. Replacement of expressions by simpler expressions (optimization).
   Example:

   ```
   REPLACE &a "+" 0 WITH &a END
   ```

4. Replacement of expressions or statements by others such that binding the program is feasible.
   Example:

   ```
   REPLACE 0 WITH 0 "AND" RQ END
   ```

   This rule apparently complicates expressions. It is required e.g. for AMD 2901 chips, however, in order to generate a zero constant at the output of the ALU. Without this rule, zeros could only be generated at the input. This example explains how 'tricks', which are used by microprogrammers, may be described and used by the compiler.

Most of these rules will be applied unconditionally. If a match occurs for such a rule, the program will be transformed and the original program is not saved.

By means of extensions to the keyword REPLACE, it is possible to define conditional replacement rules. These are applied like inference rules in expert systems. A "tiny expert system" within the compiler handles these rules. The rule which was defined in the last example should not be applied unconditionally, i.e. it should not be applied for the AMD 2901 if 0 is a memory address. Therefore, this rule has to be defined as a conditional rule.

Replacement rules are either defined in the hardware description section or in conjunction with the first declaration of types and variables. Hence they are (with exeptions) global for the whole program. Several sets of rules are collected in a library.

We found that program transformation rules helped handling a number of special cases and believe that they represent one of the main ideas which made a retargetable compiler possible.

5. THE COMPILER

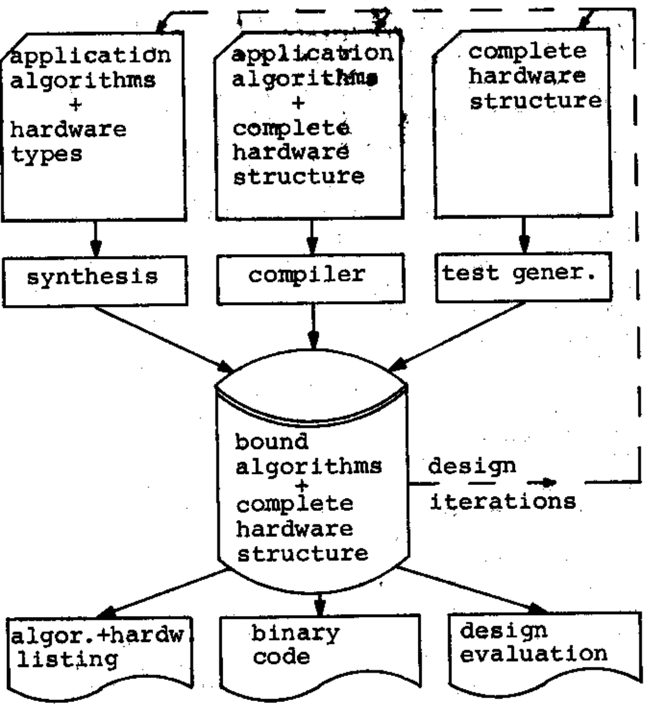
5.1 Context

The microcode generator is just one part of the MIMOLA Software System (MSS). The main goal of the MSS is to support the hardware design process. Hardware designs using the MSS usually start by selecting typical application algorithms for the hardware to be designed. These algorithms are written in MIMOLA and the synthesis part of the MSS is used in order to automatically generate a hardware structure which matches the structure of the algorithms. The first pass through the synthesis system normally does not generate a cost effective hardware structure and the ideas of human designers are needed in order to improve the structure. We take advantage of these ideas during design iterations. For each of these design iterations the designer may declare different types of hardware resources. He may, for example, change the number of memory ports or the types of available ALUs. The synthesis system will then generate the required data paths, path-switching circuits and the required control. After some initial design changes the structure becomes more and more concrete and the synthesis part is not required any more. Now, the designer only wants to change minor structural details. Standard text editors provide a means for manually changing generated hardware descriptions. After these changes, it is necessary to check if the original program can be bound to the modified hardware. This is the task of our microcode compiler. If it is able to generate code, the hardware is still correct. If it is unable to do so, the designer probably made an error. The code generator is also valuable for performance prediction and the detection of bottlenecks.

The compiler was primarily designed as a part of the hardware design system. Its ability to be used for machines, which were not designed with the MSS, is caused by the fact that it was designed for a wide range of design changes. In fact, the design may be changed such that it is completely different from the automatically generated design. For this application retargetability was a necessity.

Because of the increased importance of testing VLSI chips the MSS also incorporates a tool which generates test programs (micro-diagnostics) for a given hardware structure. For example, all data paths are tested for stuck-at errors and for shorts between adjacent lines.

Figure 4 contains a survey of possible applications of the MSS:



application algorithms + hardware types
application algorithms + complete hardware structure
complete hardware structure
synthesis
compiler
test gener.
bound algorithms + complete hardware structure
design iterations
algor.+hardw listing
binary code
design evaluation

Figure 4 Applications of the MSS (simulation not shown)

This paper focuses on microcode generation. Other papers describe synthesis and hardware design with MSS[8,9]. The MSS is completely written in PASCAL and is independent of the host.

5.2 Microcode Generation

Retargetable code generators have been around for a while. A good survey has been written by Ganapathi[10]. With the exception of Baba's MPG system[11] these systems have not been used to generate microcode. Interested readers should refer to Baba's paper for a survey on available microcode generation systems. Unfortunately, Baba's system does not match very well with the intended applications for the MSS. For example, it requires that a functional hardware description is generated. For the MSS a structural hardware description is required.

Recently Mueller and Varghese[12] described a code generation method which uses a structural hardware description. Their method is similar to the one which has been developed for the MSS.

It has been recognized that microcode generation can be modelled as parsing the source program for a grammar defined by a structural description of the target[13,14]. This interesting approach was rejected because the grammar is ambigious, does not describe resource conflicts, and is hard to parse.

In order to improve the modularity of the MSS, microcode generation was broken down into a sequence of program transformations. The first set of transformations is the same for all applications of the MSS.

Transformation 1: translation from MIMOLA into intermediate language

All software tools require that MIMOLA programs and hardware descriptions have been translated into the intermediate language TREEMOLA (tree microoperation language).

Transformation 2: translation from high-level language into register-transfer level language

This transformation replaces high-level language elements (in TREEMOLA representation) by equivalent sets of register-transfer level elements. This transformation is defined by unconditional replacement rules.

Transformation 3: storage allocation

All variables, which have not been explicitly bound, are bound by the storage allocator (this does not include temporaries). The storage allocator takes flow of data and of control into account[15] and is able to correctly assign several variables to the same location. Since all variables can be bound manually, this transformation is optional.

Transformation 4: detection of parallelism

This step replaces sequences of statements by parallel blocks of statements. This step is independent of any resource constraints. Blocks which are generated by this step are the basic units which are considered during program binding. This step simplifies compaction and is required for the synthesis of parallel architectures. Parallel blocks can be specified by the programmer and therefore this step is an optional step.

Transformation 5: generation of versions

During this step the MSS computes a set of bound statements for each statement of the source program. Every bound statement is called a version of the corresponding source statement[16]. Therefore this step is called version generation.

In order to reduce execution times, this step includes a prepass which is used to analyse the target description.
The prepass
- computes control signals which must be generated in order to activate the operations within the hardware modules.
- analyses which modules are capable of passing information unchanged from the inputs to the outputs.
- computes global connectivity information. For every output the set of inputs and outputs is computed to which information may be passed unchanged.

After this prepass is completed, versions are generated for each statement of the program. Statements are represented by flow trees. Details about these flow trees can be found in a paper describing an older version of the MSS[17]. Nodes of the flow trees represent constants, operators, memory and register references.
For every node of the tree the set of matching hardware resources is computed. Memory ports are resources matching memory references. Instruction fields, hardwired constants and decoders are resources matching constants. Function boxes match with operators that are able to perform the required operation. The set of matching resources is assigned to each node of the flow tree.

Next, the tree is traversed from the leaves to the root (depth-first-search) and a scan is made for matching data paths. For every resource matching a certain node (the source node), we try to find paths to the inputs of the resources matching the node below the current node (the sink node). If a direct path exists, a partial bound tree is assigned to the sink node. Partial bound trees are collected until the sink node becomes the next source node. Each of the partial bound trees represents the flow of data to <u>one</u> of the <u>inputs</u>. When the sink node becomes the next source node, the partial trees are <u>bundled</u>. After bundling they represent the flow of data to the <u>output</u> of the new source node. Further scanning for paths starts at the root of these bundled trees. During bundling, it is checked whether or not the partial trees are resource compatible with each other.

If a path to a multiplexer or a bus is found, a node which represents this so-called detour is inserted between the current node and the sink node. The same is done for function boxes which are able to perform operations with neutral elements (e.g. +0). Then, the algorithm is called recursively in order to find a path from the detour to the sink.

If a path is found to a temporary location, the source is assigned to that location and temporarily replaced by a 'read-temporary' operation. The algorithm is then called recursively in order to find versions for the remaining flow tree.

For every node of the flow tree, a scan for matching conditional replacement rules is made. If a rule is found, a transformed copy of the tree temporarily replaces the original tree. The algorithm is then called recursively for the modified tree. Finally the original tree is restored.

Several heuristics and cut-offs have been implemented in order to achieve an acceptable compilation speed. For example, the maximum number of detours between any two nodes is limited. This limit may be set by the user.

The output of this transformation step consists of the source program and the corresponding versions.

<u>Transformation 6: selection of versions and compaction</u>

The next step selects a version for every statement and packs versions into microinstructions. Several algorithms for microinstruction packaging are known. A modified version of the linear pairwise comparisons algorithm[15] is used in the MSS. This step also adds so-called NOOP-statements to microinstructions. These NOOP-statements insure that modules, which are not needed for a particular instruction, do not perform undesired actions. For example, unused memory input ports must be set to "LOAD-INHIBIT". Selecting instruction-compatible versions for required NOOP-statements unfortunately is a computationally complex problem.

The output of this transformation step consists of completely bound programs.

<u>Transformation 7: extraction of binary code</u>

Instruction bit patterns are generated by scanning completely bound programs for instruction bit definitions.

6. APPLICATIONS

Early applications used relatively small and simple targets. Some of these were based on AMD 2901 bitslices.

In a recent application the compiler was used for the design of an horizontally microprogrammed processor. It detected errors in the manual design of the control logic and is currently used in order to reduce the number of data paths and the instruction width. For a special hardware configuration, the system generated 125 instructions. For manual code generation, about 100 instructions were expected. The main reasons for these 25 additional instructions are a complex control logic and the heuristics used in the compaction.

In another application an AMD 29201-based machine was described. This machine is commercially sold as part of a larger system. It is a complex system with a large number of busses. It took about a month to describe the machine. This was longer than expected. One of the reasons was poor documentation. Another reason was the use of busses. Earlier applications all used multiplexers and busses had not yet been implemented.

The resulting code is somewhat slower than manually generated code if PASCAL programs are directly translated into MIMOLA. We expect that the code will be about as fast as manually generated code if MIMOLA programs are written more carefully, e.g. partially bound manually. The code for the first program is just being tested on the real machine.

The code generator was also a valuable tool during a course on computer architecture. It helped understanding microprogramming a simple architecture.

A total of about 20 different targets were programmed.

CONCLUSION

It has been demonstrated that the design of a retargetable microcode compiler is feasible. Key ideas for this success are the use of program transformation rules and of a language which is extendable and allows manual binding of variables and operations.

Problems which have been found include the existence of a large number of versions and the necessity to handle NOOP-statements.

Future work will concentrate on the speedup of the algorithms and an improved handling of temporary locations. Recent work on synthesis algorithms gave some valueable hints on how to do this.

ACKNOWLEDGEMENT

Development of the MSS was a task that involved the cooperation of a number of people over the past several years. However, I want to especially express my gratitude to R. Jöhnk for his work on the implementation of the MSS and to L. Nowak for testing the MSS.

REFERENCES

[1] S. Davidson, High Level Microprogramming - Current Usage, Future Prospects, MICRO - 16, 1983, pp. 193-200

[2] S. Dasgupta, Some Aspects of High-Level Microprogramming, Computing Surveys, 12 (1980), pp. 295-324

[3] D.J. deWitt, Extensibility - A New Approach for Designing Machine-Independent Microprogramming Languages, MICRO - 9, 1976, pp. 33-41

[4] Advanced Micro Devices Corporation, Bipolar, Microprocessor, Logic and Interface, Sunnyvale, 1983

[5] United States Department of Defense, Reference Manual for the ADA Programming Language, 1980

[6] P. Marwedel and G. Zimmermann, MIMOLA REPORT Revision 1 and MIMOLA Software System User Manual, Report 2/79, Inst. für Informatik der Universität Kiel, Kiel, 1979

[7] R. Jöhnk and P. Marwedel, MIMOLA Language Reference Manual (in preparation)

[8] P. Marwedel, The MIMOLA Design System: Tools for the Design of Digital Processors, Proc. 21th Design Automation Conf., June 1984

[9] P. Marwedel, The MIMOLA Design System: A Design System Which Spans Several Levels, in: W. Giloi (ed.): Methodologies for Computer System Design, North Holland, 1984 (in print)

[10] M. Ganapathi, C.N. Fisher and J.L. Hennessy, Retargetable Compiler Code Generation, Computing Surveys, Vol. 14, 1982, pp. 573-592

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

[12] R. A. Mueller and J. Varghese, Flow Graph Machine Models in Microcode Synthesis, MICRO-16, 1983, pp. 159-167

[13] C.J. Evangelist, G. Goertzel and H. Ofek, Using the Dataflow Analyzer on LCD Descriptions of Machines to Generate Control, Computer Hardware Description Languages, Palo Alto, 1979, pp. 109-115

[14] F. Anceau, P. Liddell, J. Mermet and Ch. Payan, CASSANDRE: A Language to Describe Digital Systems, Software Engineering, COINS III, Proc. 3rd Symp. on Computer and Information Sciences, Miami Beach, 1969

[15] M. S. Hecht, Flow Analysis of Computer Programs, North Holland, 1977

[16] P.W. Mallett, Methods for Compacting Microprograms, Ph. D. Thesis, University of Southwestern Louisiana, Lafayette, 1978

[17] P. Marwedel, A Retargetable Microcode Generation System for a High-Level Microprogramming Language, MICRO-14, 1981