A RETARGETABLE MICROCODE GENERATION SYSTEM FOR A HIGH-LEVEL MICROPROGRAMMING LANGUAGE

PETER MARWEDEL

UNIVERSITY OF KIEL, W-GERMANY

ABSTRACT

A system for the generation of microcode from a high-level microprogramming language is presented. The system is independent of the target machine because it is table-driven by a separate hardware declaration. It is applicable for horizontally microprogrammed machines.

1. Introduction

The last decade showed an increasing importance of microprogramming. Manufacturers now use microprogramming for the interpretation of machine programs, for a higher regularity of microprocessor chips and for enhanced capabilities of diagnostic routines. Users are becoming interested in microprogramming because of the larger speed of microprograms. Increasing use of microprogramming is made possible by larger and writable microinstruction stores.

An inherent characteristic of micro architectures is the fact that differences between machines are larger on the micro level than they are on the machine level, e.g. there is a large number of machines with an identical machine level and a different micro level. Therefore microprograms and microprogramming tools should be as target machine independent as possible. Otherwise firmware costs hinder the use of microprogramming. Target machine independency of tools like assemblers and and compilers is known as retargetability. Machine independency of microprograms consists of two parts: First, the syntax of the used programming language must be machine independent. Second, semantics must be the same for all used machines.

Another consequence of the necessity to reduce firmware costs is the use of a high-level microprogramming language (HLML). The advantages of a high level of abstraction during programming are well known and need not be repeated here. However, the use of HLML requires a microcode compiler. Such a compiler seems not to exist. "At present there is no microcode translation system which may be called both high-level and machine independent"[1]. It is still even the question if suitable HLML's exists "The question is still whether it is possible to create efficient high level microprogramming languages (HLML's) that are also machine independent."[2] Therefore "a number of new approaches to the design of microprogramming languages are needed".[3] And "new techniques are also needed for microprogram compilers"[3]. A good summary of the problems found in creating microprogramming tools is contained in a recent paper by Dasgupta[4]. These include the presence of low-level parallelism, the need for optimal microcode, the presence of timing constraints and the large differences between the data path structure of different machines. To this list we add the requirement of retargetability and the need for tools which are available for a wide range of host machines, i.e. machines on which the code is generated.

The best known microprogram generator is the MPG system[5]. However, the input programs for the MPG system are target machine dependent because they contain names of target machine registers.

Based on our language MIMOLA (= machine independent microprogramming language) we implemented a software system called MSS (= MIMOLA Software System)[6]. This system was originally intended as a tool for cost/performance studies during the top-down design of microprogrammable machines[7]. It was able to show the cost-effectiveness

0194-1895/81/0000/0115$00.75 © 1981 IEEE

of parallel machines designed with MSS[8]. A parallel machine, called SPDM, which was designed with MSS, is about 30 times faster than a minicomputer using about the same amount of hardware[9]. MSS has the advantage of allowing the user to define the mapping between high-level constructs like WHILE, FOR etc. and the register transfer level (RTL). This mechanism can also be used for defining the meaning of extensions to the language.

We found that some extensions of the algorithms used by MSS will turn it into a retargetable microcode compiler. Code generation with the extended MSS is described in this paper. It a step towards an efficient retargetable microcode compiler for horizontally microprogrammed machines. We assume that the microcode controls the gate level without the use of complex decoders. According to Wendt's classification[10], we consider one level programming of the gate level. We do not assume that there is a machine level above the micro level.

Before we start with any details of our method we will summarize the notation used in MIMOLA.

A MIMOLA program is a sequence of parallel instructions called esb's (=elementary statement block). Every esb starts with a label (whose initial letter is a capital L) and ends with a semicolon. Esb's contain statements, separated from each other by a comma. All statements in an esb are assumed to be executed in parallel if enough hardware is available. Lower case letters denote identifiers, S(adr) denotes cell adr of memory S (=storage) and B<name>(<fct>) is a binary function box B<name> executing function <fct>. Notation for expressions is postfix. RP is the microinstruction counter and I is the current microinstruction.

Example

FORTRAN

```
      SUM = TRUE
      DO 10 I = J,K,L
10    SUM = A(I) .AND. SUM
      ...
```

MIMOLA

```
Lsetup  S(sum) := true,
        FOR i FROM S(j) BY S(k) TO S(l);
Lloop   DO(i)
        S(sum):=S(a[i])/S(sum)->B(.AND),
        OD(i);
Lend    ... ;
```

This example or parts thereof will be used troughout chapter 2.

2. Phases of the Compilation

2.1 Overview

Stepwise translation of a HLL is a widely accepted technique for breaking down the complexity of the translation task into subtasks and has already been used in early microprogram generation systems[11]. Stepwise translation simplifies formal verification of each of the translation steps. In contrast to other translation systems, we do not use a separate 'intermediate language'. The language MIMOLA is rich enough to be used as a HLML as well as an intermediate language on a register tranfer level. This enables us to use parts of the compilation system for various steps. For example the macro processor can be used in all seven phases. Another advantage is that the user can directly write programs which can be used as input to each of the phases.

The following seven translation phases are used for the code generation with MSS :

1. Replacement of HLML constructs by register transfer level constructs.
2. Generation of trees which describe the flow of data and control (flow trees) in the program.
3. Program transformations generating a more parallel version of the program.
4. Machine-independent optimization.
5. Global storage allocation.
6. Allocation of hardware to software constructs, including necessary transformations of the flow tree. This step is also known as resource binding.
7. Generation of binary microinstructions (micro assembly phase).

2.2 Replacement of HLML constructs by register transfer level (RTL) constructs

Mapping the HLML input to the register-transfer level output during phase 1 requires only little more than string replacement with parameters if the HLML is carefully designed. But the parameter mechanism must be sophisticated and therefore even elaborate editors cannot serve as a substitute for our MIMOLA macro processor. Unfortunately, the syntax of currently available editors also depends on the host machine.

The replacement is controlled by substitution rules, so-called MACRO rules. These rules, which are read in prior to the program, can be changed by the user. They define the mapping between HLML- and RTL-constructs and they define the meaning of language extensions. A library of standard macros is available. Therefore, users who do not use language extensions, need not write macros. Possible language extensions are non-standard functions, data structures

and shorthands with parameters (macros in the usual sense).

Chapter 2.2.1 shows how macros are used to handle DO-loops:

2.2.1 Replacement of DO-loops

The following substitution rule means: Replace the string between 'MACRO' and '&&' by the string between '&&' and 'ENDMACRO'. &... denotes parameters (any string, which is a terminal or non-terminal symbol of the MIMOLA syntax, is accepted as an actual parameter).

```
MACRO FOR &id FROM &initial BY &increment
TO &limit && S(&id):=&initial,
S(&id_by):=&increment, S(&id_to):=&limit
ENDMACRO
```

This macro stores the initial value of a FOR loop in cell &id of memory S, the increment in cell &id_by (the concatenation of the actual parameter and '_by') and the upper limit in cell &id_to. This macro will expand

```
FOR i FROM S(j) BY S(k) TO S(l)    to
S(i):=S(j), S(i_by):=S(k), S(i_to):=S(l).
```

The library contains corresponding macros for the testing and incrementation of the control variable i. These macros will convert the sample program of chapter 1 to:

```
Lsetup S(sum):=true, S(i):=S(j), S(i_by):=S(k),
S(i_to):=S(l);
LLoop IF S(i) / S(i_to) -> B(>)
     THEN RP:= Lend  "RP is the program counter"
     ELSE S(sum):=S(a/S(i) ->B(+))/S(sum)->B(.AND),
          S(i)  :=S(i) / S(i_by) ->B(+), RP:=Lloop
     FI ;
Lend ...;
```

The macro processor is able to consider the nesting of DO-loops. It replaces the special parameter L&DO by the label of the esb containing the DO of the current nesting level. Furthermore it replaces the parameter L&OD1 by the label of the esb following the OD of the current nesting level.

2.2.2 Replacement of procedure calls

Compared to traditional compilers there exist three major differences in the code generated for procedure calls:

1. Program and data memories are normally separated, have a different width and in general the instruction memory can be addressed only by the program counter. Therefore the parameter blocks cannot be stored behind the procedure call in the instruction memory. Parameter information must either be stored in the data memory or must be included in the instructions ('in-line' parameter passing).
2. The speed of memories equipped with parallel ports can only be used if the parameter information is not moved in a loop which is executed once for every parameter. Some method of parallel transmission is needed if code generation for highly parallel computers is desired.
3. A run-time system may be too slow and a faster parameter passing may be required This may be passing in registers or an 'in-line' passing, at least for a small number of parameters.

The precise form of macros for procedure calls depends on the calling mechanism which is to be implemented. Macros have been written which make use of special parameters, representing counters and the name of the currently analysed subroutine.

2.3 Generation of flow trees

The next step converts the MIMOLA program into trees, which express the flow of data and of control. The flow of control describes which operations may be done in parallel, which operations depend upon conditions etc. Fig. 2.1 shows the flow tree for Lloop listed in 2.2.1:

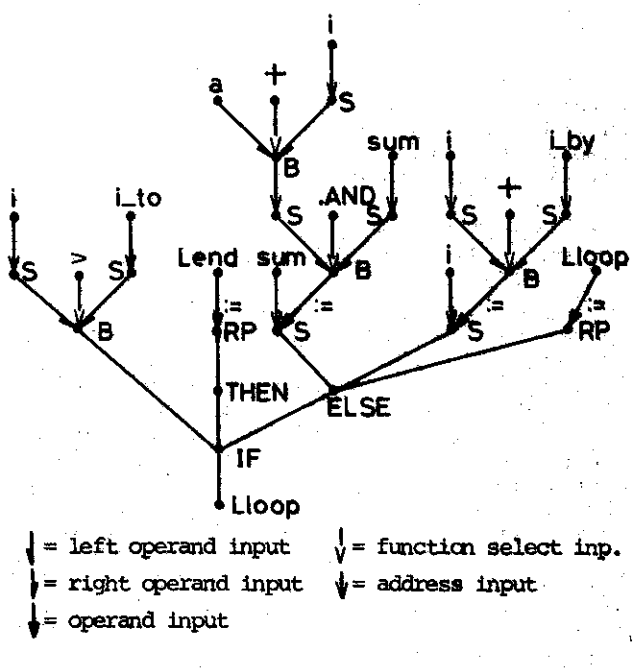


Fig. 2.2 Flow tree for Lloop

Our hardware adaptation works on these trees and not on linear lists which are normally used in microcode algorithms. Trees have the advantage of automatically expressing relations between sons of a

node ("bundling"). Such trees have already been useful for target-machine independent assembly-language code generation[12]. A common notation for these trees could possibly enable us to generate microcode for HLL's for which a tree-based assembly - language code generator exists. Schmidt and Völler[13] found that such a common tree language exists for diverse languages like FORTRAN, BASIC, PASCAL and COBOL.

Edges, except those which are related to the sequence of execution and conditional execution, all describe flow of data (or 'signals'). Associated with the edges is information about the inputs to which the data is rooted. In Fig. 2.1 this information is represented by different arrows.

Nodes describe hardware units like function boxes and memories. Single letters like 'S' and 'B' simply describe the type of hardware units. These letters will be replaced by the names of available hardware units during step 3.

We define:

A microoperation (MO) consists of a signal-describing edge of a flow graph and the source and destination node. The description of a MO includes the microcode which controls the data flow and the time which is required to complete the transport.

Linked PASCAL records are used for the internal representation of the flow trees. In order to pass these trees to the next compilation step, files are used which contain the trees in a bracket notation.

2.4 Program transformations generating a more parallel version of the input program

Microinstructions may specify many parallel operations. In order to make effective use of parallel hardware, the input program has to be made parallel if the user did not explicitly specify enough parallelism. We found out that is best first to make the program as parallel as possible and then reduce the degree of parallelism according to the available hardware. One reason is that the blocks of parallel operations form natural blocks for the code optimization. Optimizing one program block at a time and not the whole program at once is a way to reduce compilation times. This is necessary because generation of optimal code, even in simplified cases, is NP-complete[14].
The required program transformations have been published by Kuck[15] and may be applied to our flow trees.

2.5 Machine independent optimization

Optimization techniques published for the generation of machine code may also be applied to the generation of microcode. At present constant folding is implemented. Optimization is simplified by the use of flow trees.

2.6 Global storage allocation

If the input programs shall be independent of the target architecture, storage allocation must be done by the translation system (and not by the user).
For every memory in the target architecture we generate a priority list indicating which variables should be assigned to that memory. Priority computation is based upon the knowledge of the data paths, the functions of the function boxes and the expected frequency of use of that variable. An estimation of the latter may be included in the program by the user or a high-level simulator. Actual storage allocation is done during the next step, when the required number of temporary cells is known.

2.7 Allocation of hardware

This step assigns hardware to the nodes and edges of the flow tree. The way, how this is done, is most important for the quality of the generated code.

There may exist several possibilities for the assignment of hardware to the flow tree of Fig. 2.1. There may be several alternatives for sources and destinations, fields of the microinstruction or delay time. For example reading of a memory cell can be done by different ports of a memory, adding two inputs can be done by different arithmetic function boxes and a certain microcode can be supplied by various fields of the microinstruction. Each alternative of a MO is called a version (cf. Mallet[14]).

Formally, a MO consists of an index, identifying the MO's location in the flow tree, and the list of versions:

$$<k,(D_1^k, S_1^k, F_1^k, V_1^k, T_1^k),\ldots,(D_n^k, S_n^k, F_n^k, V_n^k, T_n^k)>$$

where: k is the index of the MO,

n is the number of versions of MO k,

D_i^k is the ordered set of destination bits,

S_i^k is the ordered set of sources bits,

F_i^k is the ordered set of microinstr. bits,

V_i^k is the ordered set of the values of the microinstruction bits,

T_i^k is the execution time for version i.

We include all microinstruction bits, which are involved in the execution of MO k, in the sets F_i^k and V_i^k. This is necessary in order to recognize all conflicts due to a limited width of the microinstruction. Further information about our formal definition of MO's is given a report [16].

Optimal selection of versions is known to be NP-complete [14]. Mallett gives an overview over existing microcode compaction algorithms. They are closely related to our selection problem but cannot be applied directly because of different assumptions, boundary conditions and the omission of tree transformations like the insertion of buffers.

On the other hand, algorithms used for retargetable machine-language code generators [12,17] are also not applicable, because they assume sequential machines. For sequential machines it is possible to complete code generation of one branch of the flow tree, before code generation of the next branch is started. If we have a large degree of parallelism, then we have to start with the code generation of the leaves and then proceed down to the root. Using this order, the input esbs are split into horizontal slices (c.f. Fig. 2.2) instead of vertical slices which correspond to a slower microprogram.

The target hardware is defined by a hardware declaration. This declaration specifies how certain patterns of the flow tree may be substituted by the "patterns" (the data paths etc.) of the real hardware. To this list of valid substitutions we add automatically valid algebraic substitutions, e.g. the rules of commutativity.

The target declaration needs to be written once for every target.

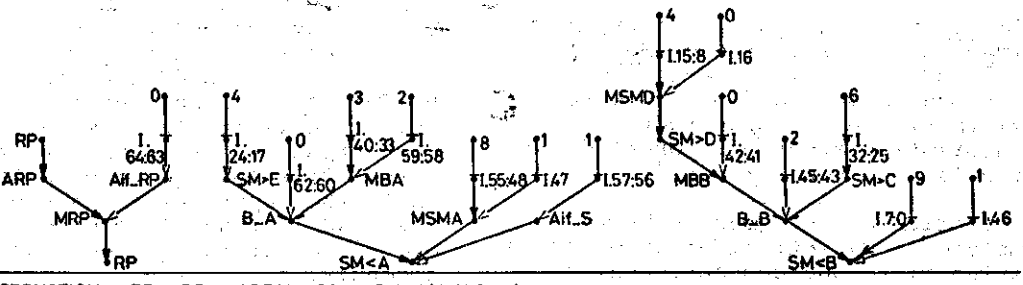
Our allocation-algorithm first puts an input esb into the input buffer and then computes which data paths of the hardware description are available for the transportation of data described by the MO's of the flow tree. The list of possible paths and related timing- and microinstruction-information is added to the flow tree (list of versions). If this list is empty, then we apply algebraic substitutions until at least one version exists. Some of the hardware patterns define detours for unimplemented direct data paths. There are non-storing and storing detours, like paths through arithmetic units and scratch-pad memories. We replace unimplemented data paths in the flow tree by detours. Then we move MO's into the output buffer (starting with the highest leaf) until the MO to be moved next cannot be included in the same microinstruction. When this occurs, all temporary results of MO's in the output buffer have to be written into temporary memory cells. If these are not available, then MO's are moved back from the output buffer to the input buffer until temporary memory cells for the remaining MO's exist. The output buffer is then written to the output file and the process is repeated until the input buffer is empty. Then, the next esb is read. A more detailed version of this algorithm is contained in the appendix.

This algorithm performs the following transformations of the flow trees:

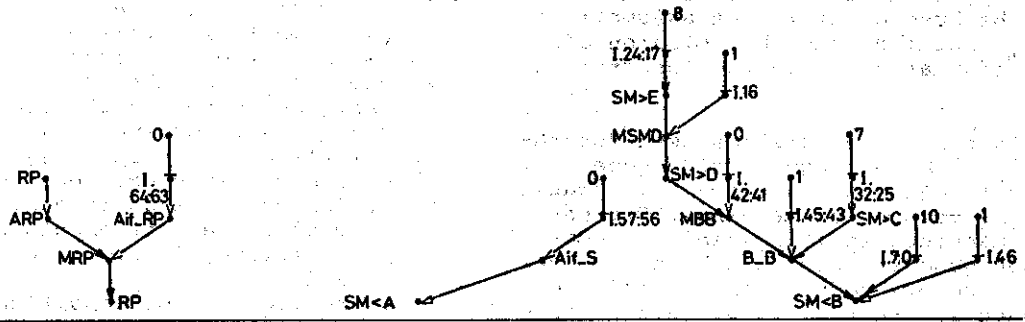
1. Assignment of hardware-units to the nodes of the flow tree
2. Replacement of the control flow by assignments to the program counter and conditional loading of memories.
3. Splitting of the input esb where necessary. This may lead to:
   a. Insertion of additional jumps (if conditional statements are splitted),
   b. insertion of buffering operations and
   c. replacement of conditional store operations by unconditional store and conditional move operations.
4. Insertion of detours (additional MO's) where data paths are missing.
5. Omission of multiple computations of common subexpressions.
6. Migration of conditional operations (conditional loading of data, conditional expressions and conditional jumps).
7. Insertion of default bitnumbers (as far as these are determined by the width of the target's hardware modules).
8. Replacement of names of functions (e.g. '+') by function codes and insertion of codes controlling multiplexers. These codes are known by the hardware declaration and, together with immediate values given by the user and binary address constants they represent the micoinstruction.

Assume that our target hardware is given by Fig. 2.3. Then our algorithm will convert the flow tree shown in Fig. 2.1 to the flow tree shown in Fig. 2.2.
For some architectures it is necessary to apply Baba's algorithm [5] for addressing the control memory.

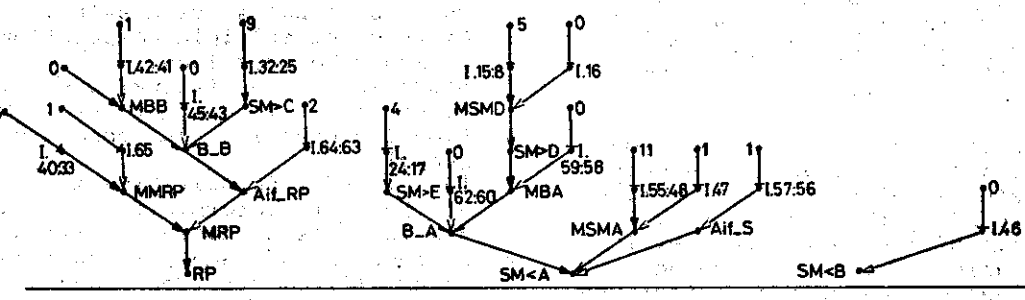
1st INSTRUCTION : RP := RP->ARP(.INCR), SM<A(t1) := SM>E(i) / a ->B_A(+),
SM<B(t2) := SM>D(i) / SM>C(i_to) ->B_B(>);



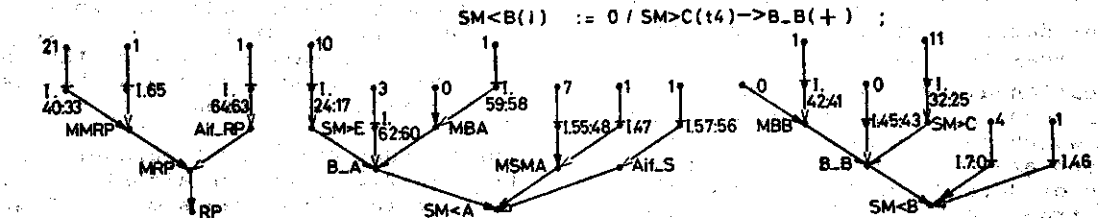
2nd INSTRUCTION : RP := RP->ARP(.INCR), SM<A(.NOLOAD),
SM<B(t3) := SM>D(SM>E(t1)) / SM>C(sum)->B_B(.NAND);



3rd INSTRUCTION : RP := (IF.0 / SM>C(t2)->B_B(+) THEN Lend ELSE RP->ARP(.INCR) FI),
SM<A(t4) := SM>E(i) / SM>D(i_by)->B_A(+), SM<B(.NOLOAD) ;



4th INSTRUCTION : RP := Lloop , SM<A(sum) := SM>E(t3) / 0 ->B_A(.NOT),
SM<B(i) := 0 / SM>C(t4)->B_B(+) ;



LEGEND	FUNCTION CODES		ASSUMED VALUES OF ADDRESS CONSTANTS		
↓ = LEFT DATA INPUT OF ALU	(DEFINED IN TARGET DESCRIPT.)		a		= 3
↓ = RIGHT DATA INPUT OF ALU	.NOLOAD (OF MEMORY)	= 0	i		= 4
↓ = DATA INPUT	.LOAD (OF MEMORY)	= 1	i_by		= 5
↓ = ADDRESS INPUT OF MEMORY	.LOAD_IF_TRUE	= 2	i_to		= 6
↓ = FUNCTION INPUT OF ALU	.LOAD_IF_FALSE	= 3	sum		= 7
↓ = CONTROL INPUT OF MEMORY	RP:=RP+1 (Alf_RP)	= 0	t1		= 8
↓ = ASSIGN VALUE TO μINSTR.FIELD	+	= 0	t2	TEMPORARY	= 9
	.NAND	= 1	t3	LOCATIONS	= 10
	>	= 2	t4		= 11
	.NOT	= 3	Lloop = 21, Lend = 25		

FIG. 2.2 FLOW TREES FOR HARDWARE-ADAPTED SAMPLE PROGRAM

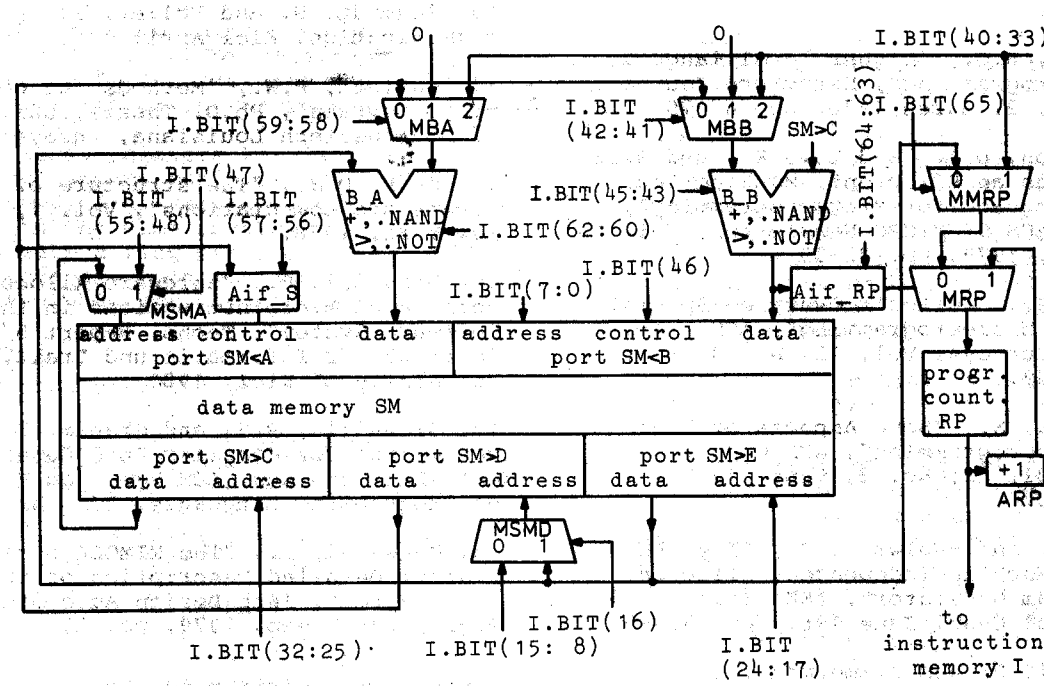


Fig. 2.3 Example of a target machine

2.8 Generation of binary microinstruction

Complete microinstructions may be assembled by collecting microinstruction bits in the flow tree ($\perp$-edges). It is like 'picking apples'. The following four instructions are collected from Fig. 2.2 (x = don't care):

	microinstruction bits															
	65	64 63	62 60	59 58	57 56	55 48	47	46 43	45 41	42 33	40 25	32 17	24 16	15 8	7 0
1.)	x	0	0	2	1	8	1	1	2	0	3	6	4	0	4	9
2.)	x	0	x	x	0	x	x	1	1	0	x	7	8	1	x	10
3.)	1	2	0	0	1	11	1	0	0	1	25	x	4	0	5	x
4.)	1	1	3	1	1	7	1	1	0	1	21	11	10	x	x	4

The flow tree may also be converted back to MIMOLA. This has the advantage of creating an easier to read instruction listing.

3. Final remarks

3.1 State of the implementation

Phases 1, 2, 4 and 7 are implemented. For the use of the MIMOLA Software System as a hardware design tool, a preliminary version of the code generation system with less capabilities, exists [18]. A high-level simulator accepting the output of phase 4 as its input, has just been completed.

All programs are implemented in a portable subset of PASCAL and have been compiled on several machines.

3.2 Quality of the generated code

Hand-compiled examples indicate that fast microinstruction sequences are generated. The algorithm computes optimum sequences for the six (small) examples in Malletts dissertation. There is a case where the current system generates an instruction sequence requiring 49 time units and one temporary buffer, regardless of the number of buffers available. If three buffers are available then the new system will generate a sequence requiring 23 time units. For our SPDM-processor we studied a sequence of 20 manually generated instructions and found that the new system would generate only 18.

Acknowledgment

Our special thanks is directed to the computer architecture group and the programming languages group at the University of Kiel.

References

1. Bushell, R.G., "Higher level language for microprogramming", Euromicro Journal, Vol. 4, No. 2, March 1978, pp. 67-75.

2. Patterson, D.A., and Lew, K., and Tuck, R., "Towards an Efficient, Machine-Independent Language for Microprogramming", MICRO-12, ACM SIGMICRO Newsletter, Vol. 10, No. 4, Dec. 1979, pp. 22-35.

3. Habib, S., "Editor's Overview-Special Section on Microprogramming", ACM Computing Surveys, Vol. 12, No. 3, Sept. 1980, p. 259.

4. Dasgupta, S., "Some Aspects of High Level Microprogramming", ACM Computing Surveys, Vol. 12, No. 3, Sept. 1980, pp. 295-323.

5. Baba, T. and Hagiwara, N., "The MPG System: A Machine-Independent Efficient Microprogram Generator", IEEE Trans. Comput., Vol C-30, June 1981, pp. 373-395.

6. Marwedel, P. and Zimmermann, G., "MIMOLA Report Revision 1 and MIMOLA Software System User Manual", Techn. Report 2/79, Institut für Informatik und Prakt. Mathematik, University of Kiel, 1979.

7. Zimmermann, G., "The MIMOLA Design System: A Computer Aided Digital Processor Design Method", 16th Design Automation Conf. Proc., San Diego, June 1979, pp. 53-58.

8. Zimmermann, G., "Cost Performance Analysis and Optimization of Highly Parallel Computer Structures: First Results of a Structured Top-Down Design Method", Proc. 4th Int. Symp. on Computer Hardware Description Languages, Palo Alto, Oct. 1979, pp. 33-39.

9. Marwedel, P., "The Design of a Subprocessor with Dynamic Microprogramming with MIMOLA", in Informatik Fachberichte, Vol. 27, G. Zimmermann, ed., Springer, Heidelberg, 1980, pp. 164-177.

10. Wendt, S., "Models and structures for microprogramming", Euromicro Symp., Venice, Oct. 1976, pp. 35-42.

11. Eckhouse, R.H., "A high-level microprogramming language (MPL)", Proc. Spring Joint Comp. Conf., 1971, pp. 169-177.

12. Cattell, R.G.G., "Formalization and Automatic Derivation of Code Generators", Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, 1978.

13. Schmidt, U. and Völler, R., private communication, Kiel April 1980.

14. Mallett, P.W., "Methods for Compacting Microprograms", Ph.D. Thesis, University of Southwestern Louisiana, Lafayette, 1978.

15. Kuck, D.J., "The structure of computers and computations", Vol. 1, Wiley, New York, 1978.

16. Marwedel, P., "Hardware Allocation for Horizontal Microinstructions in the MIMOLA Software System", Techn. Report 5/80, Institut für Informatik und Prakt. Math., University of Kiel, 1980.

17. Glanville, R.S. and Graham, S.L., "A New Method for Compiler Code Generation", Conf. Rec. 5th Ann. ACM Symp. on Principles of Programming Languages, ACM, NY, 1978.

18. Marwedel, P., "The MIMOLA Design System: Detailed Description of the Software System", 16th Design Automation Conf. Proc., San Diego, 1979, pp. 59-63.

Appendix : Algorithm for hardware adaptation

After construction of trees like in Fig. 2.1 our algorithm will add to the nodes the list of possible versions and default values for bitnumbers. Next, we replace IF-THEN-ELSE conditions by flow of data from the condition operand to control or multiplexer inputs. The condition operand may either steer the control input of the memory at the bottom of the data flow tree, or a multiplexer somewhere else in the tree. The latter is possible if THEN- and ELSE- parts contain similar trees and only part of the trees depend on the condition. If the hardware does not allow proper insertion of data switching operations then conditional jumps must be inserted. The mapping from the condition operand and the microinstruction to the enable-input of memories is assumed to be done in function boxes which have a name beginning with 'Aif'. For all possible values of condition operands, jumps must always be the last executed operation.

Next, we compute for all MO's the minimum number of time units they must start before the final store operation is completed. This may simply be the height of the destination nodes of the MO's in the tree but this number may also include delay-time characteristics of hardware units (known by the declaration). In the latter case, time units are version dependent. The following algorithm will try to select the fastest version. Therefore the algorithm will select fast, special purpose hardware if it is declared in terms of

standard functions with low execution time.

Computation of time units also includes the estimated probability of condition operands being true or false. The user may include these probabilities in the input program.

Now we compute pointers to identical sub-expressions and to conflicting versions for all versions, i.e. versions which cannot be included in the same instruction. These pointers have a flag bit and can be used in both directions. Pointers and time units are updated whenever the tree is modified.

Algorithm

0. Clear flag bits and buffer for output esb.
1. Select a 'current' MO from the set of unallocated MO's. Use a MO whose fastest version possesses the maximum number of time units.
2. Mark all versions for which the data path is not present in the target hardware.
3. Find alternative paths if all versions are marked. Try to apply algebraic rules in order to find a flow tree which matches the hardware pattern. Then try non-storing detours like buses, ALU's adding zero etc. and combinations of these. If not successful try to find storing detours. If still unsuccessful then treat the situation like a missing temporary cell (goto 6) else add detour to tree, correct time units and pointers and select the first inserted MO as current MO.
4. Test if current MO will fit into the current output instruction. Move current MO temporarily to output esb and temporarily allocate resources for MO's in the output buffer. First allocate resources for MO's with only one version, i.e. set the flags on all pointers to conflicting versions. Then allocate resources for all versions which do not point to a conflicting version in the output buffer. For all MO's with a dyadic function box as destination try to commute the inputs. Find out if all remaining unallocated MO's can be allocated. Versions, for which the flag bit on least one conflict pointer is set, may not be selected.

   Timing check:

   Functions of the special register RCLK are used to declare possible durations of microcycle and their corresponding microcodes. Before inclusion of the current MO in the output esb it is checked if it does not cause the duration of the instruction in the output buffer to be longer than the maximum allowed one. Functions of general modules may be declared with the attribute 'overlapping'. This means that the operation within a module need not be completed at the end of an instruction cycle. This feature is used e.g. for memory writes which last longer than a microinstruction. The MSS maintains down-counters for all modules which are set such that they indicate the delay time after which the module may be used again. Counting assumes the worst-case jumping between the output instructions belonging to the current input esb. Operations in all modules must be completed before jumping to other input esb's occurs.

   The amount of testing in step 4 depends upon the optimization quality the user desires.
5. If the current MO does not fit into the current instruction then remove it from the output buffer and try to find a temporary buffer for the intermediate result.
6. If a temporary buffer does not exist then move up in the tree and try to find edges where buffers can be inserted. The tree below the inserted buffer must be removed from the current output esb and its resources must be deallocated by simply removing the flags from the pointers to conflicting versions (easy backtracking!). If necessary buffers cannot be found, then the complete subtree must be deallocated and marked with a flag in order to prevent an immediate retry of allocation.
7. If there are unallocated branches of the current destination node then select the MO's of the unallocated branches as next MO's and goto 2.
8. If the MO below the current MO does not allow insertion of temporary buffers then take MO as current MO and goto 2.
9. If all leaves in the input buffer have been tested then write out the current output esb, clear the flags and remove tests in the input buffer which became redundant due to a jump in the current output buffer. Goto 1 if the input buffer contains unallocated MO's.