THE MIMOLA DESIGN SYSTEM

A COMPUTER AIDED DIGITAL PROCESSOR DESIGN METHOD

G. Zimmermann

Institut für Informatik und Praktische Mathematik
Universität Kiel, D 23 Kiel 1, W-Germany

Summary

A top-down design method is presented. The design starts with an algorithmic description of the problems and attempts to find an optimal hardware structure for the solution of the problems. Comparisons with methods using hardware and functional descriptions are included. The application of the macro processor is explained.

Introduction

A computer aided design method may be used to cut down the time and cost of a design process, while trying to achieve the same results in performance and economics of the product as a human designer. This approach is of interest if the design costs are high compared with production costs. Examples are the design of digital circuits or simple microprocessors, using large scale integration techniques. This is one of the objectives of the RT-CAD project[1,2].

The objective of our system is the design and optimization of complex digital processors. In such tasks the human designer needs help in making the best choices in the vast design space. This help is given by our design system. Thus we do not speak of an automated, but of a computer aided, interactive design. The human intelligence and the computing facilities are combined to find better designs in a reasonable amount of time. This method will be explained in this paper; further details of the design system are given in another paper[3].

Problem Definition

Hardware Description

Many hardware design methods are based on CHDL's (Computer Hardware Description Languages). Using languages like CDL, DDL and ERES, given computer structures can be described for simulation purposes, however, the proof of correctness by syntactical checks is poor and the expression of algorithms is difficult, due to the nonprocedural behaviour. The main handicap of these methods is the wrong direction of the design: the starting point is a more or less detailed hardware design, not the problem description.

Our system allows nonprocedural hardware descriptions - DECLARATION's - during the stepwise, interactive design process. The design language is MIMOLA[4], a procedural CHDL that is also suited as a HLL to describe algorithms. The hardware level is the register transfer level; the action level is the state transition or microprogramming level. The description of this state transition of the synchronous automaton, which models the computer structure, is called a microstatement. It is commonly interpreted as a word of microcode in a program memory. In the MIMOLA syntax a microstatement begins with "L" (label) and ends with ";". Declarations use an extended subset of MIMOLA.

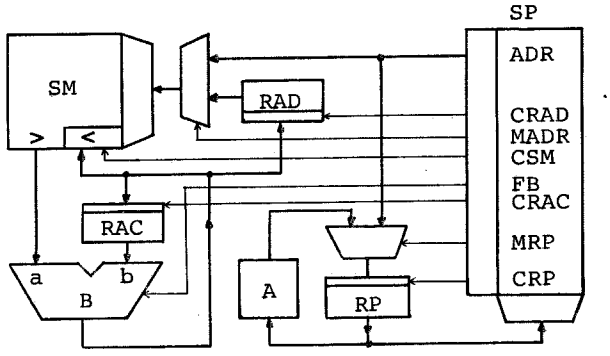

Figure 1: Simple Processor Example

```
DECLARE
ADDMODUL
    SM<A(16384:0).BIT(15:0), SM>A
    "main memory, 16k, 16 bits, 2 ports" ,
    SP(1024:0).BIT(21:0) "microprogr.mem.",
    RAC.BIT(9:0) "accumulator" ,
    RAD.BIT(15:0) "address register" ,
    RP.BIT(9:0) "program counter" ,
    A(.INCR) "monadic operator" ,
    B(+,-,.a,.b) "adder" ,
    I.BIT(21)CRP.BIT(20)MRP.BIT(19:18)FB
     .BIT(17)CRAC.BIT(16)CSM.BIT(15)MADR
     .BIT(14)CRAD.BIT(13:0)ADR
    "microprogram word, field names" ;
```

Figure 2: Modul Declaration

Fig. 2 shows the declaration of the modules of a sample computer in fig. 1. It is not possible to give a complete definition of MIMOLA here. In Appendix A some of the terms are explained. The examples can be understood by comparing software and hardware together with the "comments".

Fig. 3 completes this to a full declaration of the hardware. The MIMOLA design system transforms these declarations into a data base[3] which contains all the information about the hardware.

```
ADDCONNECTION
    "declaration of data, address and
    control paths"
    B<a <- SM>A,"memory out to adder input"
    B<b <- RAC,
    B.FCT <- I.FB "microprogram field to
    adder function control",
    RAC <- B "adder out to accumulator",
    RAC.CON <- I.CRAC "accu load enable",
    RAD <- B, RAD.CON <- I.CRAD,
    SM<A <- B, SM<A.CON <- I.CSM,
    SM.ADR <- I.ADR / RAD "address field
    and address register via multiplexer
    to address port of memory" ,
    SM.ADR.MPX <- I.MADR "address multi-
    plexer control" ,
    RP <- A / I.ADR, RP.CON <- I.CRP,
    RP.MPX <- I.MRP,
    A <- RP, SP.ADR <- RP;
ENDDECLARE
```

Figure 3: Connection Declaration

Functional Description

A top-down design should start with a definition of the function. The function of a digital automaton or processor can be expressed in a procedural formal language. If this language is a CHDL and if we are able to define a complete set of functions, we call this a functional description. A given set of machine instructions is a good example of a base for a functional description of a processor.

There are two ways to write a description:

1. If we define every detail of the execution on the microprogramming level, there is no free design space in the transformation from the functional description to the hardware description. This transformation is called the allocation process. Fig. 4 gives a functional description of 6 microstatements in MIMOLA. The allocator transforms this to the hardware of fig. 1.

2. In order to expand the allocation process design space, we have to raise the level of the functional description. On this higher level the designer can describe the semantics of the functions without giving execution details. Fig. 5 is an example of a transformation of Fig. 4 to a higher level.

If the original intention was to calculate i:=j+k and jump to "label", with i, j, k in memory SM, we can use an even higher level of functional description as shown in Fig. 6.

A correct transformation of a functional description to a hardware structure will assure that the hardware will execute the functions correctly. This is the main advantage over a hardware description. The correctness of the functional description is in the field of program correctness and beyond the scope of this paper. A simulation on the starting level might be useful.

The hardware design will be only as detailed as the description is. On the RT-level, modules like adders and memories are language primitives and are assumed to behave according to their description. If the available modules differ from the assumed ones, there exist three possibilities:

1. The designer designs new "ideal" modules, using a conventional design method. This is useful for theoretical investigations and for small series or it may lead to new commercially available modules.

2. If the first way is to expensive, the designer can try to add some hardware to existing modules to construct "ideal" interfaces.

3. The MIMOLA description has to be adapted to the available modules. This may lead to a performance degradiation and to unstructured designs.

```
Lload    RAC := SM(I.ADR) / X -> B(.a),
         RP  := RP -> A(.INCR);
           "load accumulator from memory"
Lstore   SM(I.ADR) := X / RAC -> B(.b),
         RP  := RP -> A(.INCR);
           "store accumulator in memory"
Ladd     RAC := SM(I.ADR) / RAC -> B(+),
         RP  := RP -> A(.INCR);
           "add to accumulator"
Lladr    RAD := SM(I.ADR) / X -> B(.a),
         RP  := RP -> A(.INCR);
           "load address"
Laddind  RAC := SM(RAD) / RAC -> B(+),
         RP  := RP -> A(.INCR);
           "add to accu indirect"
Ljump    RP  := I.ADR; "uncondit. jump"
```

Figure 4: Functional Description

```
Lload    RAC := SM(I.ADR);
Lstore   SM(I.ADR) := RAC;
Ladd     RAC := SM(I.ADR) / RAC -> B(+);
Laddind  RAC := SM( SM(I.ADR) ) / RAC
                -> B(+);
Ljump    RP  := I.ADR;
```

Figure 5: High Level Functional Description

```
Laddjump SM(i) := SM(j) / SM(k) -> B(+),
         GOTO Label;
           "symbolic addressing"
```

Figure 6: Algorithmic Description

The Description Level

Fig. 4 - 6 have shown that the level of the description can be changed. What is the highest possible description level?

In the "Lload" statement, the transformation from Fig. 4 to Fig. 5 minimized the number of data paths. This class of transformations does not change the semantics and the state transitions. The highest possible description level is equivalent to the shortest MIMOLA expression.

In a second class of transformations the level is raised by decreasing the number of

microstatements or state transitions. This can be done by parallel execution of sequential order-independent statements or by building more complex expressions, avoiding the storage of intermediate results. This process may be supported by reordering the statements and restructuring conditional branches.

As far as these transformations do not alter the essential state transitions of the automaton, they will be considered as permissible. Thus the highest level of the description will only contain the essential state transitions.

The problem has now been shifted to the question: what are the essential states?

In the case of a traffic light controller the answer is easy: Every state transition changing a light is essential. If the function of machine instructions are described, every instruction must be represented by at least one state transition. It is unessential on the ASSEMBLER-level if "shift accu 8 times left" is executed in one or 8 state transitions. Some complex instructions will need more than one state.

Problems arise if we have to deal with more complex functions. A common problem is the calculation of the Fourier transformation. We could look at the whole transformation as one state transition. Normally the FFT algorithm is used. It is possible in MIMOLA to express the FFT in one microstatement, but such a description would be at such a high level that the realization of the hardware would be too difficult.

An algorithm is normally executed in program loops. In a synchronous automaton a loop has at least two states: the begin - identified by a label - and the end of the loop with a "GOTO label". Both states may be identical. Similar cases are subroutine CALLs and RETURNs. In our design system we call this "reasonable level" MIMOLA-Ø. Its exact position is determined by the programmer.

Algorithms can be expressed without exlicit state transitions by data flow languages[5], functional languages as proposed by Backus[6], or the "value trace" in the RT-CAD project [2]. We can define these descriptions as the highest possible level. These high level descriptions have not yet directly been included in our design system.

Asynchronous Operations

Our present intention is to design synchronous automatons. Asynchronous operations may occur on three levels:

1. MIMOLA can express concurrent statements and spatial sequential operations in one microinstruction. The internal tree representation of a microinstruction has a great similarity to a data flow graph. The difference is in the clock transition that synchronizes all storage operations. Thus deadlocks and races are prevented.

2. Asynchronous concurrencies on the RT-level overlapping several state transitions are forbidden in MIMOLA because deadlocks and races cannot be excluded in general. Such constructions are allowed in many CHDLs and cause trouble in the hardware implementation.

3. On a processor level these synchronization problems arise on a level transparent to the programmer. High level synchronization tools can be included in concurrent programs and built in hardware. This solution is preferred in our system as long as algorithms for the distribution of programs into concurrent processes do not exist. Thus the problem can be reduced to the independent design of several processors as in 1.

Algorithmic Description

In our example of a FFT-function we have already reached a high problem level. The description is not given in terms of machine instructions but in terms of an algorithm. If the goal is to design a processor only capable of executing this function, we can still speak of a functional description. In the more general case the problem will be to design a processor for one or several classes of algorithms, where only very general specifications of the classes exist. Typical classes are arithmetic problems, signal processing, text processing, information systems, compilers and operating systems. This is the problem we attack by our design system. It is no longer possible to define a complete set of functions or algorithms, but we must select a set of representatives of the classes weighted according to their importance in the proposed application of the processor.

This set must be large enough to decrease the influence of selection or estimation errors in the statistical mean. Thus the description will become very redundant. This redundancy together with the weights will be used for statistical analysis in the design.

The description is now no longer functional and we call it algorithmic. Special cases with limited function sets as our FFT example will be included here.

A language to describe algorithms should be powerful and descriptive. Therefore several HLL structures are included in MIMOLA:

```
FOR   FROM  BY  TO    WHILE  DO  OD

IF  THEN  ELSE  FI

CASE  OF  THEN  THEN ...  ELSE  ESAC

CALL , RETURN , GOTO
```

These instructions have no unique hardware representation on the RT-level. Their semantics are not fixed in MIMOLA but should be close to other HLL. The semantics have to be declared by macros throughout the design process. In this way the design space can be extended by macros.

In addition to these built-in macros user macros can be inserted and declared. This is a powerful tool in MIMOLA to introduce new HLL-structures like semaphores, monitors, or complex addressing schemes. Besides keeping the program source small, it allows the designer to experiment with different algorithms.

The Design Method

The design goal is to find an optimal hardware structure for a given set of algorithmic descriptions under given constraints. Measures for the optimum and constraints are

execution speed, costs, space and power consumption and program and microprogram storage requirements.

The design space contains the parameters: number of physical modules (memories, alu's, shifters), number of memory ports, function sets, address space, number of data paths, multiplexers and buses, microinstruction wordlength and module delay times. In addition there exists an algorithmic design space.

At the state of the art it is impossible to find an analytical solution of this optimization problem. In our opinion we are far from an algorithmic solution. We use an interactive approach to get enough experience to transfer interactions to automatic actions.

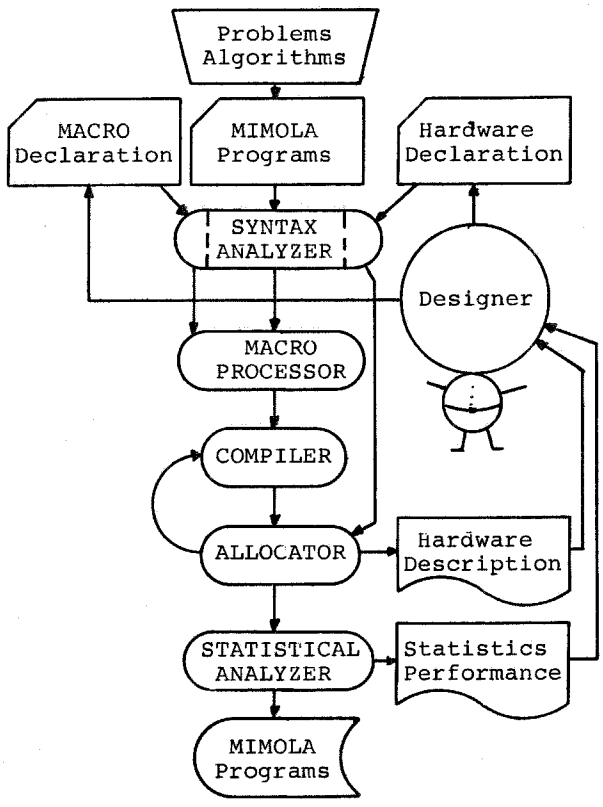

Figure 7: Design Method Flow Chart

Fig. 7 shows the designer's view of the software system. Starting with a MIMOLA-Ø program set and unrestricted hardware resources, the syntactical correctness is checked. In addition to error printouts warnings are generated pointing out errors that influence the program execution but not the hardware. Examples are "undefined label" and "undefined macro". As long as no hardware constraints are in effect, the COMPILER action is zero. The ALLOCATOR will add new resources to the data base as long as no frees can be found there. The resources are mainly modules and connections. After every state transition the resources are free again and can be used in the next microstatement. Thus the data base contains the union of all resource needs for all microstatements in the input program set. Local optimization strategies are used to minimize the union.

At this point the data base contains a hardware description of a processor that can execute the programs in the original number of state transitions and with our definition of MIMOLA-Ø. This is the minimum number of transitions or the performance maximum. Usually cost constraints will forbid this solution.

The task of the designer is to restrict the hardware in a way that he tries to go backwards through the design space always as near to the performance maximum as possible. He can estimate the performance reduction due to resource restrictions from statistical values, generated by the STATISTICAL ANALYSER. Measured is the number of times F(r) a resource r is used in the program set. These measurements will be realistic only if FACTOR's are included in the calculation that represent the probability of the occurrence of a statement in the intended environment of the processor. If the microstatement i in the program k is executed n(i,k) times in a loop and k is called m(k) times in the mean in a time period T, the factor is

$$FACTOR(i,k)=n(i,k) \times m(k) \quad .$$

The frequency F(r) is calculated from

$$F(r) = \sum_k \sum_i FACTOR(i,k) \times B(i,k,r) \quad ,$$

B(i,k,r) is 1 if the resource r is used in the microstatement i,k; 0 otherwise.

$$N = \sum_k \sum_i FACTOR(i,k)$$

is the number of microprogram steps or clock cycles that occur in time T. This is a rough measurement of the processor time consumption of all sample tasks that are processed in T. The exact time is

$$t = \sum_k \sum_i FACTOR(i,k) \times \tau(i,k)$$

where $\tau(i,k)$ is the executions time of the microinstruction i,k. It depends on the longest data path in the instruction, and can be calculated from individual delays in the resources.

If t_o is measured on the MIMOLA-Ø level,

t_o / t is a relative performance measure.

The ratio U(r)=F(r)/N denotes the utilization of a resource. As the allocator first tries to minimize the use of resources and then will select resources in a definite order, the utilization distribution for equal resources, e.g. different ports of a memory module, will show strong differences. Small U(r) will indicate candidate resources for deletion. Further information is provided by joint usage statistics.

The factors n and m cannot be calculated by the system as no simulation is done. They must be supplied with the programs. In the case of loops they can often be calculated or estimated. In other cases measurements will be appropriate. With operating systems as input problems, it would be impossible to run a simulation in a correct environment on the MIMOLA-level. If we use an existing operating system the most correct achievable factors can be obtained by measurements. Many implementations offer some of these data for tuning purposes.

The task of the designer will be to reduce the costs of the MIMOLA-Ø level hardware structure. He will start with high-cost low-utilized resources, e.g. memory ports. He can delete these resources from the data base by DECLARATION's. At the same time further declarations of this resource can be forbidden.

When the program set is processed the next time, more resources will be demanded in some microstatements than are available. In most cases the compiler can solve these problems by sequentialization, introduction of storage cells for intermediate results or restructuring. Unresolvable situations will be reported to the designer. In such a case he can revise his declaration or try to find a solution not known to the ALLOCATOR or the COMPILER. The last can be performed for the short term by macros and will later lead to software systems improvements.

The effect of the compilation and primary of the declaration can again be seen in the statistics. If the hardware reduction is done in small steps, it is possible to keep the design always as near to the performance optimum as possible. The iteration stops when the design constraints are fulfilled.

The Macros

The standard macros of MIMOLA have already been mentioned. Macros generally make it possible to postpone decisions on execution details until a later cycle in the design process. The semantics of the macros are fixed by convention or by the programmer.

An example of the first type is the reference to an array element in the memory S. In MIMOLA this can be expressed e.g. by

S(array-name [S(i),S(j)]) .

There exists a design space in the choice of an algorithm to calculate the effective address. A simple one is:

S(array-name/S(i)->B(+)/S(j)/dimension ->B(*)->B(+))

Now the hardware to execute this expression can be allocated.

The tool for expanding the macro is a macro declaration. The MACRO PROCESSOR compares all syntax trees of the programs with the tree of the macro to be replaced. The replacement is made in the tree structure. The PROCESSOR is not limited to standard or user macros but can replace nearly every expression in a microstatement. Thus the algorithmic design space is extended. This tool must be handled very carefully to avoid changing the semantics or introducing errors.

Another example is the

CALL subroutine (parameters)

macro statement. The semantics will depend on the type of the subroutines and parameters. Recursive and nonrecursive subroutines will require different algorithms. This is a decision about the task of the processor. In the design itself general hardware decisions have to be made. Different stack concepts are known. The macro facility gives the designer the possibility of experimenting with different concepts to find a solution that fits into the already designed structure.

User-defined macros open a new possibility. In cases where the language is inadequate and the programmer wants to express an operation with known semantics but unknown execution details, a macro can be used. Again experiments with different replacements can be made. Examples are semaphore operations: Mp(..) , Mv (..), or a special call: Mcall monitor (..).

Thus the macros are valuable tools for exploring the algorithmic design space and for defining the details that will lead to a complete hardware design. An example of the latter is the definition of the control part of the processor. The choice of a special algorithm will have the result that the hardware will have the best performance if this algorithm is used in the later application. This is the responsibility of the compiler designer. It does not mean that other algorithms cannot be executed. To assure more flexibility, different algorithms can be defined concurrently by duplicating the statements and using appropriate weights.

Conclusion

The language MIMOLA is capable of describing hardware, functions and algorithms. The hardware description is the result of an interactive design process starting with a functional or an algorithmic description. It describes the modules on an RT-level and all connections of data, functions, addresses and control lines.

Starting with a functional description that defines every state transition in detail, the MIMOLA Design System software will produce a full hardware description in one pass. This fulfills the demand for a low cost and fast design process.

A higher level functional description of a given instruction set of a computer opens a greater design space. An optimal solution will require several passes through the system and a human designer's interaction. A project with the IBM 370 instruction set is in progress to allow a comparison of our design with real machines.

As the instruction sets build an artificial bottleneck, our main purpose is to start on the higher problem level and go down to the great possibilities of microprogramming. The step back to an intermediate language that can be interpreted by the hardware is a future goal of our system. This will only be done if a code compression can be achieved on this way.

Today many people speak about computer networks, multi-minis and multi-micros as a solution towards higher performance. In our opinion the single processor has not yet reached its optimum performance. If we can find structures that will increase the performance, many difficulties with computer networks can be avoided.

The state of the project is as follows: The MIMOLA Software System, written in PASCAL, has been implemented and is used for several applications. Further improvements in the COMPILER and the MACRO PROCESSOR will be added. A FORTRAN-MIMOLA compiler[7] exists, and a MIMOLA-microcode translator is under construction. The main goals at the

moment are applications. A special purpose processor for a scientific subroutine library has been constructed using the method[8]. The translation of operating systems to MIMOLA has been started.

Appendix A

In this section some of the MIMOLA notations are listed. The complete syntax of the LR(1) Grammar, the semantics and examples are collected in the Report[4].

Modules:

XYYdZZ (aaa).FFF

X module group: 1 letter
- S storage
- R register
- K stack
- A monadic operator
- B dyadic operator
- V bus

YY module name, free

d port direction
- < input
- > output
- <> bidirectional

ZZ port name, free

aaa address, function: output ports, constants, identifier

FFF attribute, e.g. description of subfields

Connections:

Explicit in the declaration part:

III <- OOO,OOO,...
- III Input port, connected via multiplexer or bus with
- OOO output ports.

Implicit in the program part:

OOO -> A(aaa) expression: input ports of monadic /

OOO/OOO - B(aaa) dyadic operators are connected to the output port(s) of module(s). Complex expressions in postfix notation can be constructed

DDD:= expression microinstruction; the value of the expression is stored in the destination DDD

Time dependence:

Laaa microinstruction, microinstruction,..;
microstatement: all storage operations in the microinstructions are executed synchronously (state transition of the automaton). The program counter is incremented automatically if not otherwise stated.

References

1. L.J. Hafer, A.C. Parker, "Register-Transfer Level Digital Design Automation: The Allocation Process". Design Automation Conference Proceedings, Vol. 15, 1978.

2. E.A. Snow, D.P. Siewiorek, D.E. Thomas, "A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations, and Design Tradeoffs", Design Automation Conference Proceedings, Vol. 15, 1978.

3. P. Marwedel, "The MIMOLA Design System: Detailed Description of the Software System" 16th Design Automation Conference Proceedings, 1979.

4. G. Zimmermann, "Report on the Computer Architecture Design Language MIMOLA", Bericht des Instituts für Informatik und Praktische Mathematik, Universität Kiel Nr. 4/77/1977.

5. J.B. Dennis, "First Version of a Data Flow Procedure Language", Lecture Notes in Computer Science, Vol. 19, pp. 362-376, Springer-Verlag, 1974.

6. J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", Comm. of the ACM, Vol. 21, 8, p. 613-641 (1978).

7. K. Kasprzyk, "Entwicklung eines FORTRAN-MIMOLA Compilers für den Entwurf von Rechnerstrukturen", Diplomarbeit, Universität Kiel, 1978.

8. K. Schultze, "Systematischer Entwurf eines Prozessors mit der Sprache MIMOLA", Diplomarbiet, Universität 1978.