THE MIMOLA DESIGN SYSTEM:
DETAILED DESCRIPTION OF THE SOFTWARE SYSTEM

Peter Marwedel
University of Kiel
Kiel, Germany

Summary

A software system is described which aids in the design of digital processors with a top-down method. It takes the design language MIMOLA as its input. Input may be either a high level functional or a structural description of the hardware. The output is a description of the hardware on the block level. In the case of a high level functional input, the output contains a listing of the hardware which is necessary to execute the input together with utilization factors of the hardware units. The amount of creatable hardware may be limited in a declaration. If there is not enough hardware to execute the input statements in parallel, the necessary intermediate steps are inserted by the system and a corresponding functional description is presented.

Introduction

The general motivation has been set up in a previous paper[1]. We will now give some details of our design system.

The complete system may be subdivided into different parts according to Fig. 1. Basic features of the macro processor have been described in[1] and the lexical analyser and the syntax analyser will be described in a forthcoming report[2]. We restrict ourselves to a description of the hardware allocator, the compiler, and the statistical analyser. A prerequisite for understanding the allocator is the knowledge of the hardware data tables.

Hardware Data Structure and Declarations

The hardware data structure (hds) contains all information about the available hardware. It is needed for determining if the current microstatement is executable in parallel and for storing design information. The structure has been specially developed for this purpose and is similar to the network model[3] of data bases. We did not use available data base systems because most of them are not portable and none is implemented in a language suitable for compiler construction. Because operations on the hds occur quite frequently, the hds has to remain resident in the primary storage.

In order to define a hds, hardware resources have been structured mainly hierarchically:

A group of modules consists of all modules with the same first letter in their identifying module names. Examples are the groups of registers and of stacks.

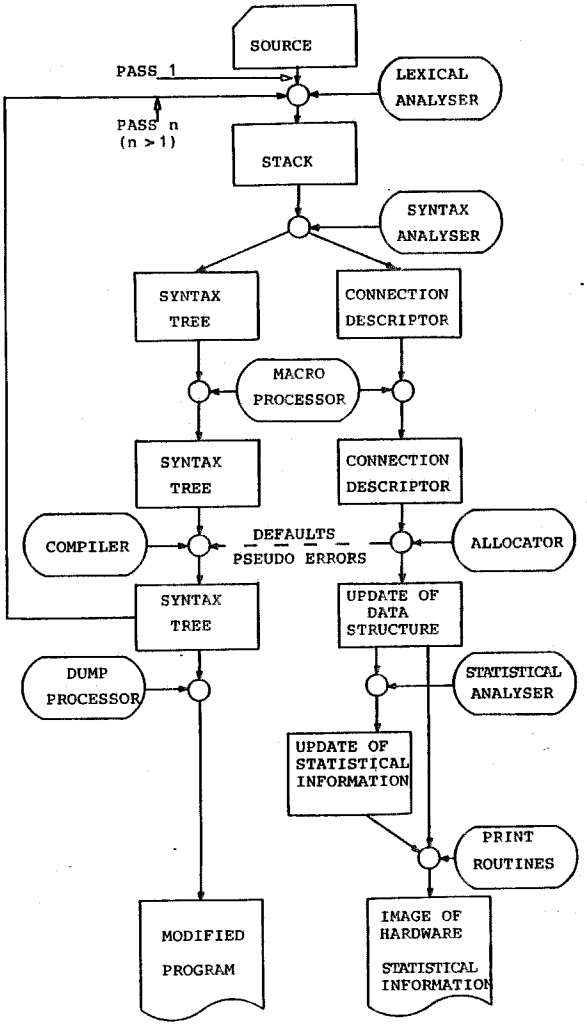


Fig. 1 Flow of data in the MIMOLA software system

Modules must have at least one port for doing input or output. Some groups have a fixed number of ports (e.g. dyadic operators have three: two for input and one for output), others have a varying number of ports (e.g. storages). Ports are identified by the direction characters < and > and by a name.

Each port has up to four inputs or three inputs and one output: function input, address input, control input and data input or output. Inputs and outputs are identified by the standard attributes .FCT, .ADR, .CON, and .DAT. The latter is assumed by default.

Inputs and outputs have fields of one or more bits. Fields can also be described by symbolic names; in this case the corresponding bitnumbers may be defined in a more refined design step. Input fields can be viewed as multiplexers if there is more than one source for a field. In this case the field has an associated multiplexer address input field. This field can be identified by a .MPX attribute.

Sources in their most general form are concatenated subranges of fields of output ports. Concatenation can be expressed by the star operator.

By writing all the names and attributes one after the other, one can create and address all nodes in the data structure. Additional attributes and syntactic rules allow the modification of entries in the nodes. By this means the user has complete control over the data structure in the declaration part and therefore can fully specify all the hardware. It is up to him to create meaningful hardware structures.

The declaration has been divided into two parts:

1. Declaration of connections (sources) using the ADDCONNECTION command,
2. Declaration of the other data structure nodes using the ADDMODUL command.

We include an example in order to demonstrate the relations between the various descriptive tools.

The following block diagram

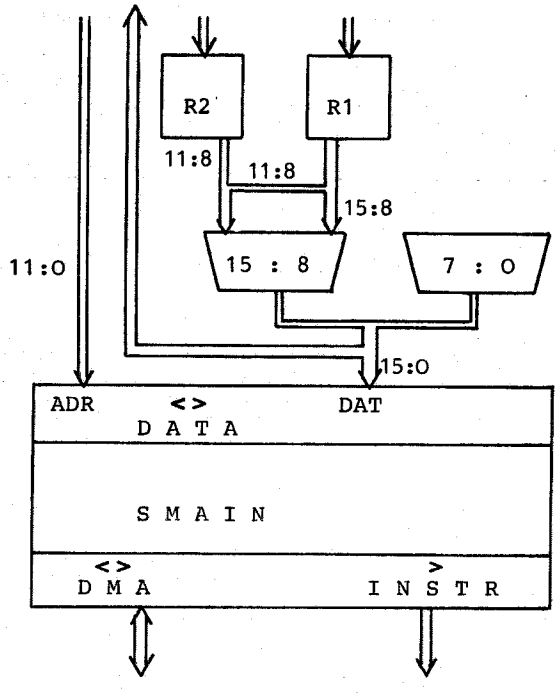


Fig. 2 Hardware block diagram

may be declared in MIMOLA as

```
ADDMODUL
SMAIN<>DMA , SMAIN >INSTR,
SMAIN<>DATA.ADR.BIT(11:O),
SMAIN<>DATA.DAT.BIT(15:8,7:O),
R1> ,R1< ,R2> ,R2< ;

ADDCONNECTION
SMAIN<>DATA.DAT.BIT(15:8)   -> R1.BIT(15:8) /
R2.BIT(11:8) * R1.BIT(11:8);
```

The allocator converts this into following data structure:

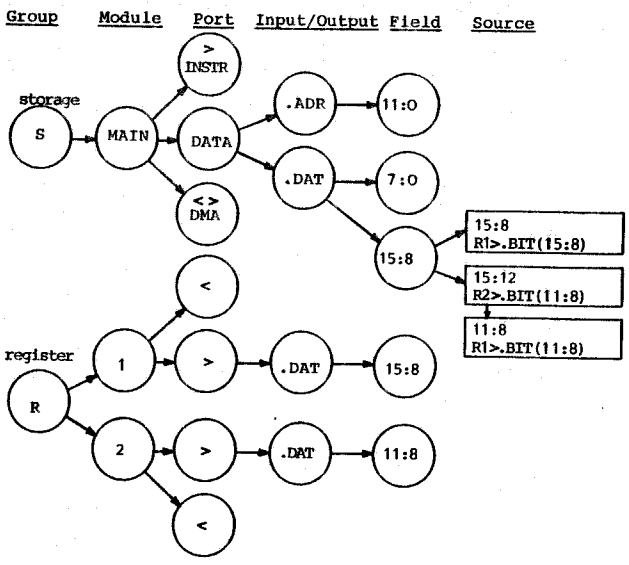


Fig. 3 Hardware data structure

Allocator

When the software system analyses a program part, it must find the correspondence between the language expressions and the hardware, represented by the hardware data structur hds. This is done by the hardware allocator which does the resource management. If there is not enough hardware to execute the µstatement in parallel, the allocator determines what additional hardware may be defined. The amount of additional hardware is restricted by the declaration. If the necessary extensions are not allowed, the allocator gives a pseudo error indication, thereby stimulating the compiler to divide the microstatement. Other errors, like writing concurrently into the same destination, are also indicated but do not stimulate the compiler.

The resources which the allocator must distribute are input fields and the code of the microinstruction. Output fields can be accessed concurrently; the allocator only indicates their use for statistical purposes. Output port allocation is necessary, however, because most output ports contain function or address inputs at which conflicts might occur.

Conflicts normally are recognized by an attempt to use the same bit of the same input field by different sources not under mutually exclusive conditions. Conflicting jumps are only a special case in that they would result in a conflict at a program counter input field. Two problems unfortunately are undecidable for the allocator:

1. In some cases mutual exclusion cannot be checked before run time. In this case the allocator assumes there is no mutual exclusion and possibly indicates too many errors.

2. If more than one input port of a storage or stack with non-constant addresses is present in the same microstatement, the allocator does not know whether they are equal. It assumes they are not equal, because experience has shown that even warnings by the allocator would be too numerous to be individually examined by the user.

By 'microinstruction' we mean one instruction word of the program storage. There is no state-sequencing while one instruction is valid; therefore it is called microinstruction. We do not assume a machine language level exists. A microinstruction consists of fields of bits. Fields are used to steer the address inputs of storages, function inputs of operators, enable inputs, and multiplexer addresses. Normally each field is dedicated to one such input; sharing one field among different inputs may be declared.

By 'microstatement' we mean the description of the entire state transition from one clock cycle to the next.

A microoperation is the transition of a subset of bits, e.g. S(a):=R1.

Resource management for the microinstruction consists of assigning a symbolic value to the microinstruction's fields the first time these fields are needed in the current microstatement. In the case of symbolic addresses for example, the alphanumeric identifier is assigned to the corresponding microinstruction field. While assigning a different identifier in the same microstatement is a conflicting use of the resource 'microinstruction' a second use of the same identifier is allowed. This assignment process combines resource management and microprogram generation in a natural way. Therefore the allocator is capable of acting like a microprogram assembler. However, it is advisable to use a different program for code generation for the sake of program modularity and a less complex software system.

The allocator reads its input out of small intermediate connection-describing records. The original intermediate structure is completed by insertion of default values for directions and for bitnumbers/names. If the microinstruction is the source, a symbolic subrange depending on the name of the destination is taken as the source bit specification. Otherwise if the input does not have a different field width, the standard attribute .WORD is taken.

Automatic horizontal migration on the module and port level is inherent to the idea of a MIMOLA design system. Two such features have already been included in the software system:

1. The omission of port names for stacks and storages indicates that the allocator shall select suitable ports. The allocator then will first try to find a port where the address connection already exists and is usable. If no such port exists, the user may choose to take the first free port or to optimize data connections. In the second case the allocator tries to find a port where the required data connection already exists and is usable. If there is more than one port with the required data connection, the one with the most frequently used connection is taken. This optimization is essential for the design process because it tries to create few frequently used connections instead of scattering uses over the set of all possible connections. The designer's part of the optimization process is to delete some unfrequently used connections whose usefulness could not be foreseen by scanning only once over the entire program.

The automatic allocation of ports is one of the degrees of freedom in the design space which is used by MIMOLA.

The allocator will create new ports if the current number is not sufficient and if creation is allowed. The maximum number of ports may be declared in the declaration. If no new ports are allowed, the allocator will stimulate the compiler to split the microstatement.

2. If more than the present number of operators is needed, the allocator tries to create duplicates of existing operators. The number of allowable duplicates may be defined in the declaration.Compiler stimulation is used as above.

The compiler may add the default bitnumbers, portnames and names of duplicates to the syntax tree. Later dumping of the terminal symbols in the tree then results in a more refined version of the source program.

Proceeding this way from microstatement to microstatement, the allocator creates an image of hardware which may execute the given input. 'Used' - flags are the input to the statistical analyser, and pseudo-errors cause the splitting of certain microstatements. If a high level functional description is input, the allocator will convert the inherent parallelism into parallel hardware. Using a high level input for the allocator avoids the machine level bottleneck. Conventional machine languages can also be used as input in order to study utilization factors of conventional hardware structures.

Compiler

The compiler is triggered by pseudo-errors from the allocator when splitting of the current microstatement is necessary. It takes the syntax tree as its input, and the output is a syntax tree of microstatements which may run on the hardware. The compiler calls the statistical analyser when a microstatement is free of pseudo-errors.

The output program is in a canonical form due to the splitting algorithm[4]:

```
if there are pseudo errors
then call testrecursiv;
     check if splitting requires buffering
      in order to maintain program semantics;
     find optimal sequence of µstatements
      such that buffer requirements are mini-
      mal;
     call testrecursiv "buffering can cause
      pseudo errors";
     minimize buffer use
fi;
insert default bitnumbers, portnames, dupli-
cate names;
call dump processor to put out new program;
call statistical analyser;
stop;
```

```
procedure testrecursiv
begin
0. let the first nodes of the syntax tree be-
   long to non-jumping unconditional μopera-
   tions, followed by conditional μoperations
   and finally the jump class μoperations.
1. if μoperations are in error because of a
   use of the required hardware in the same
   μoperation, store the source of the connec-
   tion in a temporary cell and continue with
   step 1 as long as there are such errors.
2. if there are error-free unconditional
   μoperations, make a separate μstatement
   out of them together with error-free
   unconditional μoperations else goto 4.
3. call testrecursiv for the remaining μope-
   rations.
4. if there are conditional μoperations left
   then take the errorless part of the first
   conditional μoperation and make a μstate-
   ment out of it
   else return
5. if the THEN part contained errors make a
   microstatement out of it.
6. if the ELSE part contained errors make a
   microstatement out of it.
7. make a μstatement out of the rest of the
   original μstatement (e.g. jumps)
8. call testrecursiv of steps 5 - 7.
end;
```

The user may choose to use registers or a RAM storage module with at least three ports as temporary storages.

An example is given for the resulting new programs:

INPUT:

```
ADDMODUL SA>A(#FFFF:0), SA<>B(#FFFF:0);
         "storage with an output and a bidi-
          rectional port, other module and
          the connections will be declared
          by appearance in the program"

PROGRAM  "one port missing in each μstatement"
         BEGIN
 Lone SA(a):=SA(b)/SA(c)->B1(+);
 Ltwo IF SA(R1)->A1(.INCR).BIT(3)
       THEN R1:=SA(b), SA(b):=R1, GOTO Lone
       ELSE R1:=SA(d), R2:=SA(e)
      FI, R3:=F1;
END
```

OUTPUT:

```
PROGRAM BEGIN
Lone.1 Rtmp_101:=SA>A(b)/SA>B(c)->B1(+);
Lone.2 SA<B(a) :=Rtmp_101;
Ltwo.1 Rtmp_101:=R1,   R3:=F1,
       IF SA>A(R1)->A1(.INCR).BIT(3)
        THEN R1:=SA>B(b)
        ELSE R1:=SA>B(d), GOTO Ltwo.3
       FI;
Ltwo.2 SA<B(b):=Rtmp_101, GOTO Lone.1;
Ltwo.3 R2:=SA>A(e);
END
```

Statistical Analyser

The hardware allocator identifies all connections which have been used in a microstatement. If all pseudo-errors have been resolved by splitting the microstatement, a statistical analyser uses this information in order to compute the necessary information for the design process. The unequal importance of different microstatements may be expressed by defining different weighting factors. The following information is computed:

1. frequency of module uses
This information is required in order to know all the modules which shall be deleted in the next design step.

2. frequency distribution of the number of ports required in μstatements
It is counted, how many times parallel execution of a μstatement requires n ($0 \leq n \leq 9$) ports of a storage module. The number of concurrent uses of different ports of the same module determines the number of independant ports in the final design.

3. frequency of uses of a particular port
This information determines the utilization factor of particular ports.

4. frequency of connection uses
Connections are basic cost factors and the frequency of their use is needed in an economic design.

5. joint frequency distribution of concurrent module and port uses
Often a hardware unit may be substituted by another hardware unit, e.g. an incrementer by an arithmetic function box (horizontal migration). The joint distribution indicates whether the substitution of two or more units by a single unit would significantly increase the number of microstatements. The same is true of ports of storages and stacks. For example, two ports with different direction which are rarely used concurrently may be substituted by a single bidirectional port.

6. Estimation of run-time
It is assumed that addressing of storage ports, propagation through operators, microinstruction reading and the write cycle of storage ports each require one unit of time. Overlapping is considered correctly.

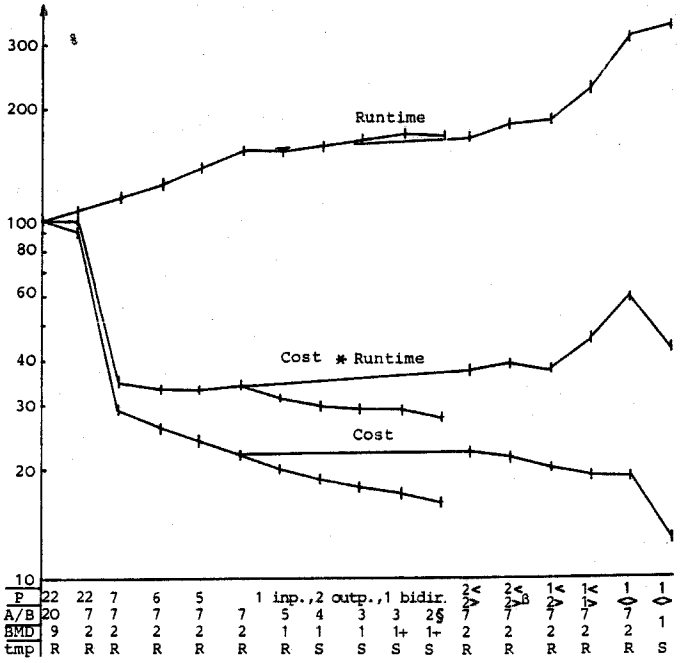
Manual setting of weighting factors for the μstatements is important for this computation.

First Results

A selection of subroutines of the IBM scientific subroutine package has been manually converted to MIMOLA[5]. This selection has then been used as input to the MIMOLA Design System. The number of allowable storage ports has been varied from unlimited to one. The number of operators has been varied, holding the number of ports fixed to one and four. Cost factors of storages, operators, multiplexers and connections have been estimated and the resulting costs have been computed for the different designs. Fig. 4 shows the resulting curves for runtime, cost, and the product of both. Notice that there is a minimum for four ports of the main storage, far away from classical architectures. Improving the minimization of connections, which is our next task, will decrease the costs of multiport architectures while the cost of the one-port architecture will remain fixed.

Final Remarks

A software system has been built which aids in the design of digital processors with a top-down method described in[1]. It can process the large number of input statements necessary to get significant statistical information. The system now will be applied to various design problems and will be further extended if there is a need to do so. It is written entirely in Standard PASCAL and there-



P	22	22	7	6	5	1	inp.,2	outp.,1	bidir.	2< 2>	2< 2>ß	1< 2>	1< 1>	1 ◇	1 ◇
A/B	20	7	7	7	7	7	5	4	3	3	2§	7	7	7	7	7	1
BMD	9	2	2	2	2	2	1	1	1	1+	1+	2	2	2	2	2	
tmp	R	R	R	R	R	R	R	S	S	S	S	R	R	R	R	R	S

P : number of ports of main memory
A/B: number of simple operators
BMD: number of units for multiplication or division
tmp: intermediate storage method: R = register, S = RAM
+ : shift-operations replaced by multiplication/division
§ : 2 function-boxes instead of separate add-,subtract-,and compare-units
ß : 1 common address

Fig. 4 Cost and speed of different architectures

fore is portable. It has been implemented on a SIEMENS 7.760, a DECSYSTEM-10, and a UNIVAC 1100. Storage requirements are 400 - 500 k bytes on the SIEMENS and 100k words on the UNIVAC.

References

1. G. Zimmermann: The MIMOLA Design System: A Computer Aided Digital Processor Design Method, Design Automation Conference Proceedings, Vol. 16, 1979
2. P. Marwedel: Algorithms and Data Structures for a MIMOLA Design System, Report of the Institut für Informatik und Praktische Mathematik, Kiel, 1979 (forthcoming)
3. G. Schlageter, W. Stucky: Datenbanksysteme: Konzepte und Modelle, Stuttgart, 1977
4. U. Zimmermann: Diploma Thesis, Kiel, 1979 (forthcoming)
5. K. Schultze: Systematischer Entwurf eines Prozessors mit der Sprache MIMOLA, Diploma Thesis, Kiel, 1978