MIMOLA PRIMER

MIMOLA Software System Primer (Version MSSI)

CTC-83-7: 8214

Gerhard Zimmermann, Richard Cloutier, Richard Rudell
Mark Albert, Beth Hurd

Honeywell Corporate Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, Minnesota 55420
(612)887-4589

December 1982

TABLE OF CONTENTS

OTHER PAPERS AVAILABLE

Zimmermann, G. and Marwedel, P. "MIMOLA Report Rev. 1 and the MIMOLA Software System Users Manual, Christian-Albrechts-Universitat, Kiel, Bericht Nr. 2/79, May 1979.

Zimmermann, G. "The MIMOLA Design System, A Computer Aided Processor Design Method," *16th Design Automation Conference Proceedings,* San Diego 1979, 53-58.

Zimmermann, G. "MDS - The MIMOLA Design Method," invited paper, *Journal of Digital Systems,* Vol. 4, No. 3, 1980, pp. 337-369.

Zimmermann, G. "VLSI Design with the MIMOLA Design System," *Proceedings IEEE European Conference on Electronic Design Automation,* Brighton UK, 1981, pp. 277-280.

Zimmermann, G. "Computer Aided Synthesis of Digital Systems," *Proceedings 5th Intl. IFIP Conf. on Computer Hardware Description Languages and Their Application,* Kaiserslautern 1981.

Rahf, D.. Zimmermann, G. and R. Cloutier. "MIMD User's Manual," (translated from Master's thesis entitled "Ein Codegenerator und Speicherzuordnungs - und Parallelisierungs-strategien fur MIMOLA-Programme," University of Kiel, December 1981.) CCSC Document, October 1982.

Cloutier, R. and R. Rudell. "MIMOLA to DL Translator - Users Manual," CCSC Document, October 1982.

Cloutier, R. and R. Rudell. "MIMOLA to DL Translator - Computer Operation Manual," CCSC Document, December 1982.

I. INTRODUCTION

OVERVIEW

The MIMOLA Software System (MSS) is a new tool to help the designer of digital systems, mainly in the initial phases of the design. Therefore, it is called a front-end or planning tool. The microprogramming facility can also be used during implementation and use of the digital system.

This PRIMER is intended to introduce users of the MSS to the application of this new tool and to direct them to more detailed documentation. The PRIMER is written in a tutorial style, including examples. The reader is encouraged to follow the examples and to do the exercises.

The PRIMER also explains how some things work in the software system. This knowledge is not necessary to actually use the system, but it may help explain why certain things are done as they are. It also helps the designer understand the results s/he will achieve, which will enable the user to more fully control the software.

The MSS should always be viewed as an aid, not as an automatic tool.

MOTIVATION

Who should read the PRIMER? What will you get out of it?

Traditionally, we would address the digital systems architect, subsystem and logic designer. However, for VLSI systems we also address the "tall" designer, who carries the design through all levels. The MIMOLA System is especially suited for firmware and hardware design of processor-like structures, with complex data and control units.

The PRIMER will teach the initial use of the currently available tools of the MIMOLA System. New PRIMERS will be published for new software, as necessary. The information in this PRIMER will help the user evaluate the usefulness of the tools as they apply to his special needs. It will allow him to use the tool and get some hands-on experience. It will also prepare him to make full use of the MIMOLA Report /3/.

If you have no idea what MIMOLA is, the authors strongly recommend that you read the reprint of the paper "What is MIMOLA?" (Appendix A).

WHERE MIMOLA IS USED

We are trying to build a digital system that fulfills given requirements. The requirements mainly express the behavior, but can also include an objective function and other parameters, like cost and size. The first task in a hierarchical design is to find a digital systems structure, for example a design in the form of schematics. The second task is to map this structure onto a physical design, for example, a layout. As a third task, we have to fabricate (build) the system. Finally we have to test, program (if possible), install, and maintain the system.

With the MIMOLA System, we do part of the first task, called structural design. In a hierarchy of refinement levels, we do the structural design of synchronous subsystems (for example, processors), using primitives at the register-transfer (RT) level.

Now keeping these limitations in mind (structural design, subsystem to RT level), we first have to specify the requirements.

II. GETTING STARTED

We now have to decide what we want to design. This "what" will be called the "System Under Design" or SUD. Some thought should be given here to: "What is necessary to specify the SUD?" We do not want to specify too many details, and thus limit the design space. We also do not want to omit information that could cause unwanted behavior of the SUD. So, the proper specification is a balance of abstraction level and completeness.

Let us take an example. Suppose we want to design a microprocessor with the same instruction set as system xyz. The right abstraction level is the user's handbook that describes the instructions of xyz. It is also complete in the sense that it contains all instructions. Further specifications include the required performance for a given benchmark, and physical properties, such as the number and size of chips, etc. The objective function could be the maximum performance for a given physical configuration, such as a single chip implementation.

Another example might be a traffic light controller. We can completely specify it by a state transition diagram that shows the visible states, which would be the right abstraction level. Again we can have additional requirements like reliability, cost, or temperature range.

The completeness becomes more of a problem if the SUD is more generic, as for example, an image processing unit. Many image processing algorithms are possible, and we may want a machine that can execute all of them. We cannot possibly enumerate all of the algorithms, but each may be implemented in terms of more basic operators. This is also true for instruction set machines. We have learned that even with very primitive instruction sets we can execute any executable algorithm.*(1) Thus, primitive image processing operations can be easily implemented in the SUD. So the question of completeness is more: For which set of algorithms do we want to achieve an optimal or some other level of performance? It is now up to the architect to decide on a kind of benchmark of algorithms that should be considered, and weigh them. This, then, defines a complete set. The correct level of abstraction is typically a description of the algorithms in a high level programming language or a software design language.

Now let us consider an example that we will develop throughout this PRIMER. It should be simple, but still show enough different functions to exercise the basic features of the MIMOLA System. Unlike the top-down approach of a normal system design, we will proceed bottom-up in this example. That means we start with the primitive functions and add more complex ones later. This makes it easier to learn the use of MIMOLA.

DESIGN EXAMPLE:

Requirements: Design a microprogrammable system that can calculate sin(cyclic_frequency * time). We may later want to extend the function to a wave form synthesizer.

Refinement (specification of algorithm): More specifically, we want to calculate sin in the following manner.

$$\sin(x) = x - \frac{x^{**}3}{3!} + \frac{x^{**}5}{5!} - \ldots$$

Let us call

```
cyclic_frequency   = c
time               = t
```

then we can write

$$\sin(c^*t) = \sum_{n=0}^{\infty} \frac{(ct)^{2n+1}(-1)^n}{(2n+1)!}$$

or a(0) = c * t
sin(0) = c * t

(1) *The Turing Machine gives theoretical proof for this.

```
a(n)      = -a(n-1)*(c*t)**2/(2n*(2n+1))
sin(n)    = sin(n-1) + a(n)
```

SELECTION OF THE ALGORITHMS

If the algorithms of the specified functions have not already been defined, we have to decide now. This decision will somewhat influence the structure of the SUD, and only experience with the MIMOLA System can really help. The guideline is: Use a fast algorithm, and use parallelism and loops as you would in a High Level Language (HLL).

Algorithm Refinement:

EXAMPLE: We will use an iterative loop based on the previous recursive equations. The iteration stops when the a(n) becomes less than 0.004 (we may choose to use 1/256 in order to make it simple in the binary number system).

```
---------------------------------------------
|  First, describe the algorithm in a High   |
|  Level Language (e.g., Pascal, PL/1,       |
|  Fortran, etc.)                            |
---------------------------------------------
```

Now we write a little BASIC program to verify our algorithm.

LIST PRIMER_1.B

```
10  print 'This Program calculates sin(c*t)'
20  print ' c = ';
30  input c
40  print ' t = ';
50  input t
60  n = 1
70  x = c * t
80  s = x
90  a = x
100 if abs(a) < .004 then goto 200
110 a = -a * x * x / (2 * n * ( 2 * n + 1 ) )
120 s = s + a
130 print 'n = ';n,'    a = ';a, '    s = ';s
140 n = n + 1
150 goto 100
200 print 'sin( ';x;' ) = ';s
210 print ' type: 0 for quit, 1 for new c and 2 for new t'
220 input i
230 if i = 1 then goto 20
240 if i = 2 then goto 40
250 end
```

A typical result is:

```
LIST PRIMER.RUN

   run

 BASIC- 3.0-08/02/0700  1982/12/08 0932:26                       PAGE 1
 ^ZDA>UDD>DA>ZIMMERMANN>PRIMER_1.B


 This Program calculates sin(c*t)
@ c = ? 1
@ t = ? 1
 n =  1              a = -.166667                    s =  .833333
 n =  2              a =  .833333E-2                 s =  .841667
 n =  3              a = -.198413E-3                 s =  .841468
 sin(  1  ) =  .841468
  type: 0 for quit, 1 for new c and 2 for new t
@? 0
```

Checking our pocket calculator gives

sin(1) = 0.841471

Our result is accurate within the error margin of 0.004.

```
-----------------------------------------
|  Note:  It is recommended that you try |
|  the exercises before checking the     |
|  answers in the appendix.              |
-----------------------------------------
```

EXERCISE 1
Try this algorithm on your system for the numbers 2, 4, and 8.

We see in this exercise that a variable can take on a wide range of values. The designers should be aware of this, to ensure that there are no underflow or overflow errors because of wordsize limitations.

```
-----------------------------------------
|  The selection of the algorithms to    |
|  describe the behavior influences the  |
|  structure of the System Under Design, |
|  SUD.                                  |
-----------------------------------------
```

DESCRIBING THE ALGORITHMS IN MIMOLA

In order to enter the selected algorithms into the MSS, we have to describe them in MIMOLA. MIMOLA is a procedural Hardware Description Language (HDL). It has been developed especially for the task of structural design, and therefore, differs slightly from a High Level Language, as we will see. We introduce here only the constructs needed for our example in a step-by-step fashion. A complete description can be found in the Report /3/. In the BASIC program lines 60-120 and 140-150 are the parts we want to map into hardware. The other lines are only necessary for exercising it.

```
-------------------------------------------
| The first character of all MIMOLA names |
| is case sensitive.                      |
-------------------------------------------
```

EXAMPLE: Let us start by converting line 60 of the BASIC program into MIMOLA.

```
  n = 1      --->     S(n):=1
```

(The "--->" symbol represents the conversion of BASIC into MIMOLA.)

In this conversion the variable called n, which implies a storage location in BASIC, is explicitly declared as a storage location in the MIMOLA statement. S(n) describes a storage device (a memory) that is addressed at the location defined by n. The name of the memory is S, other memories that could exist might be called: Smain, S1 or SINDEX. All memory names in MIMOLA start with an upper case S. Variable names, however, start with a lower case letter or the underscore character ("_"). We also see that ":=" means assignment (as is Pascal).

```
-----------------------------------------------
| S      :  Storage, addressable memory       |
| Sname  :  Specific storage module           |
| S(var):   Contents of storage location var  |
| var    :  Variable name, first letter lower |
|           case or "_"                       |
-----------------------------------------------
```

In MIMOLA, as in BASIC, we have labels for each instruction, and each label starts with L. The label name does not define an order in MIMOLA!

```
     60 n = 1        --->       L60 S(n):=1 ;
```

The ";" denotes the end of the instruction, which can continue over several lines. MIMOLA is free format, so lines have no special meaning.

Line 70 contains an expression in the infix notation used by BASIC. MIMOLA uses postfix notation.

```
     70 x = c * t        --->       L70 S(x):= S(c)/S(t)->B(*);
```

We assume that c and t were already stored in the memory locations S(c) and S(t) by some, as yet undefined, statement.

L70 contains some new symbols, so let us first consider "B(*)". B(*) describes a dyadic operator, in this case it performs the operation "*" upon its two inputs. The name of this dyadic operator is B; other names could be Balu, B1 or BMUL. Note that the first character of a dyadic operator's name is always an upper case B. The inputs for the operator are immediately to the left of the "->" symbol that points to the operator name. The inputs are also separated from each other by the slash character ("/").

```
-------------------------------------------------------
|  Expressions:  Expressions are postfix notation     |
|                without parentheses                   |
|  op / op -> b-operator:  dyadic expression (two     |
|                           operands "op")              |
|  b-operator:  B(function), executes "function" on   |
|                the next two operators to the left    |
|  function:     "+", "-", "*", ".name", ....          |
-------------------------------------------------------
```

EXERCISE 2
Write the following infix expressions in the MIMOLA postfix notation.

```
a + b
2 + (c + 5)*5
[(3 + 4)*(1 + 2)] + [3*(a+b)]
```

(The solution is in Appendix B.)

The next lines are easy to translate:

```
     80 s = x        --->       L80 S(s) := S(x);
     90 a = x        --->       L90 S(a) := S(x);
```

So far, the translation seems to be fairly straightforward. Before we proceed, we will peek into the MSS. Because:

```
-------------------------------------------------
| "The Description of the Algorithm influences |
|  the Structure of the SUD"                    |
-------------------------------------------------
```

III. A CLOSER LOOK AT THE MSS

HARDWARE ALLOCATION

Let us see how the MSS allocates hardware for the behavioral expressions we have described so far. This process is also called resource binding, because we can think of the hardware modules and connections as resources that are bound to primitive operations in our instructions.

The MSS scans each instruction from left to right. As necessary, hardware modules are added to the hardware database. If possible, hardware modules that already exist in the database, but are not in use for the current instruction, are reused. After each instruction is processed (a ";" indicates the end of the instruction) all of the modules in the database are released for future use. The ";" indicates that all assignment operators for the current statement are to occur and that a state transition has occured.

```
     EXAMPLE
       L60 S(n) := 1;
```

Let us skip the translation of the label "L60" for now. We will come back to it when we discuss control.

S(n) is a destination, so the Allocator creates a module S with a data input "A". This can be shown to the designer as:

In the PROGRAM Text
  S<A(n) - indicates that the operand S(n) is now bound to input A of memory S (i.e., Port A).

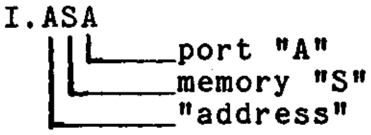
In the hardware description section
  S<A(...) - indicates that a port called A exists for the module S.

The address n has to come from somewhere. The default assumption is that any constant (such as n) is assigned to a microprogram field. The microprogram vector is called I and the fieldname

"I.fieldname" is generated from the module name, with the destination type as a first letter.

Destinations (first letter) in I.fieldname:	
A	Address
D	Data
F	Function
C	Control
M	Multiplexer Addr.

So, the field for n tranlates into



The "1" in our expression is a constant and is directly assigned to the data port S<A. Therefore, the "1" is stored in another microprogram vector field DSA.

```
          1 translates to I(1).DSA
```

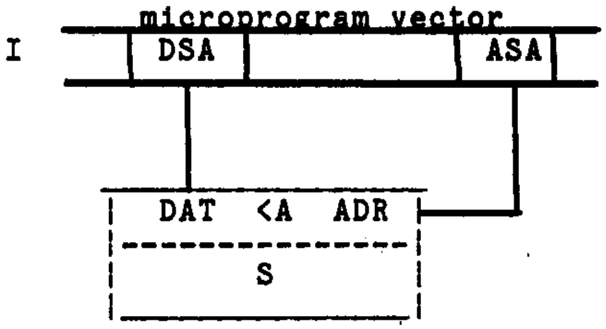
So far we have, in a long bound form,:

```
          L60 S<A(I(n).ASA) := I(1).DSA ;
```

This implies connections, which we can show in structural MIMOLA as (destination <- source_list).

```
          S<A.DAT <- I.DSA
          S<A.ADR <- I.ASA
```

or as schematic



When the MSS produces structural MIMOLA it is placed in the "SCR" file.

Here again we have seen the use of the construct:

```
          port.attribute
```

Where DAT and ADR stand for the data and address connections to the "A" port. Therefore, an attribute can be used to define fields of a vector and also types of ports.

S name < portname	:	Input port of S name
S name > portname	:	Output port of S name
I	:	Microinstruction control vector
port.name	:	Attribute, e.g., field of a vector or port
.DAT	:	Data field attribute
.ADR	:	Address field attribute

Let us now see if the MIMOLA Software System yields the same result. We use the Allocator called MIMB and look at the SCR file. (The examples were generated on a Honeywell L66/GCOS system.):

```
*LIST/GEZ/PRA.SCR

$ADDMODULE
S(0:0),
S<A.DAT.WORD,
S<A.ADR.WORD;
$ADDCONNECT
S<A.DAT.WORD         <-   I>.DSA,
S<A.ADR.WORD         <-   I>.ASA;
```

Under the key-word ADDMODULE we see a list of the modules and respective ports. S(0:0) means a memory S with an addressing range 0:0. This is because we did not tell the system how large the memory should be, and it defaults to the smallest possible size. We also see that for the ports DAT and ADR the system assumed a default bitsize called word, which is undefined at this point.

The ADDCONNECT section, which shows the connections between the hardware devices, is exactly what we predicted.

This uncertainty about the bitsize and the zero number of words in the memory may confuse the system (it could lead to zero length vectors, which are then suppressed). At this point in a design, we probably have some idea about these parameters. We can specify a value now and change it later, if necessary.

EXAMPLE:
Let us assume an 8K memory and a 16 bit wordsize. We specify this by preceding our program with

```
$ADDMODULE
S(8191:0).BIT(15:0)WORD ;
```

We use the same syntax as in the SCR file that we looked at before. Actually, the SCR output is designed to be used as an input in the next iteration. We only have to add or delete details to create an input specification that reflects what we want to describe.

BIT (a:b) name is a bit range attribute and can be used in both the structure and the program part. The optional "name" defines an attribute called "name" that can be used instead of BIT(a:b) from there on.

MORE SYNTAX

To run our design through the MSS, we need some more information. The behavioral section must start with $PROGRAM, and it must be enclosed by a BEGIN END pair (to give it the appearance of an HLL). The $ tells the system that PROGRAM is a "monitor keyword".

EXAMPLE
The file INP, the input to MIMB, looks like this:

```
*LIST/GEZ/PRC.INP

$ADDMODULE
S(8191:0).BIT(15:0)WORD;
$PROGRAM
BEGIN
L60  S(n) := 1;
END
```

Now, running MIMB again, we get as a structural output:

```
*LIST/GEZ/PRC.SCR

$ADDMODULE
I>.BIT(12:0)ASB,
        I>.BIT(28:13)DSB,
        S>A(8191:0),
        S>A.BIT(15:0),
        S<B.DAT.BIT(15:0),
        S<B.ADR.BIT(12:0);
        $ADDCONNECT
        S<B.DAT.BIT(15:0)   <-  I>.DSB,
        S<B.ADR.BIT(12:0)   <-  I>.ASB;
```

This is much more than we expected, but we will see that it makes sense. The first two lines define two fields of the microinstruction word: Address ASB with 13 bits - for 8K memory - and DSB with 16 bits, as a data input to S.

The third line defines an output >A of the memory. This is a system default, because if a memory is named without a port (line 3 of our input program), the port >A is assumed. The address range and word size are what we defined.

The input port is now <B, which changes our naming, but nothing else. Also, the address port ADR for S<B is 13 bits, as it should be.

In ADDCONNECT only WORD has been replaced by the correct bitsize.

Looking at the GEN file, we see the same syntax as in the INP file. Thus, this could be used as an input if necessary. We also see that in the bound program the input <B has been selected, as expected.

```
*LIST/GEZ/PRC.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
      $PASS    1
      $PROGRAM


BEGIN


L60.1
          S<B(n):=1;


END
```

IV. BACK TO OUR EXAMPLE

MORE HARDWARE ALLOCATION

Now let us see what the next line of the BASIC program, line 70, adds to the hardware structure. Of course we will try to reuse the structure we have so far:

```
     L70 S(x) := S(c)/S(t) -> B(*);
```

Scanning it from left to right, we first assign existing Port <B to S(x), then port >A to S(c), then create a new port, >C, for S(t). We have to interrupt here to make a note: If we define a module or port in ADDMODULE, this implies that it is the only one allowed, unless we state otherwise. Thus, the only allowed ports of S would be >A and <B. We can give more freedom by adding the standard attribute "MOREPORT" to S.

```
     S(8191:0).BIT(15:0)WORD.MOREPORT
```

Now back to the L70 example. An ALU called B is needed to perform a multiplication, so we create one. We also have to create new inputs of B, called B<a and B<b.

Input S<A had been used before. Can we still use the data or address connections to it? If we look at the last SCR file, the address input ADR still seems to be ok, because we again address S with a variable name. The data input DAT was connected to a constant before, and now needs to be connected to the output of B.

MULTIPLEXERS

A multiplexer (MUX) must be introduced at the input to S<B.DAT. (The term muiltiplexer is used here; however, a bus structure could work as well.) Multiplexer names are automatically generated by the system. Every MUX name starts with a "U" followed by a character that indicates the information being switched by the MUX (D, A, F or C for the data, address, function or control). The rest of the name identifies the module and port that are connected to the output of the MUX. For the current example, the MUX at the data input port "B" of the memory called "S" would be UDSB:

```
          UDSB
          ││││
          │││└───────Port "B"
          ││└────────Module "S"
          │└─────────"data"
          └──────────"MUX"
```

The INP input to MIMB:

```
*LIST /GEZ/PRD.INP

$ADDMODULE
S(8191:0).BIT(15:0)WORD.MOREPORT;
$PROGRAM
BEGIN
L60  S(n) := 1;
L70  S(x) := S(c) / S(t) -> B(*);
END
```

generates the following GEN file.

```
*LIST /GEZ/PRD.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN


L60.1
          S<B(n):=1;

L70.1
          S<B(x):=S>A(c)/S>C(t)->B(*);


END
```

Note that the MUX does not appear in this file. L70.1 in the output is the bound instruction, as we would expect. In the SCR file, we now see the new structure, which includes the multiplexer.

```
*LIST /GEZ/PRD.SCR

$ADDMODULE
B>(*[1,0]),
B>.BIT(15:0),
B<a.DAT.BIT(15:0),
B<b.DAT.BIT(15:0),
I>.BIT(12:0)ASB,
I>.BIT(28:13)DSB,
I>.BIT(41:29)ASA,
I>.BIT(54:42)ASC,
I>.BIT(55:55)MDSB,
S>A(8191:0),
S>A.BIT(15:0),
S>A.ADR.BIT(12:0),
S<B.DAT.BIT(15:0)
    "?,UDSB<a.DAT.BIT(15:0),
    UDSB<b.DAT.BIT(15:0),
    UDSB>.MPX.BIT(0:0),
    UDSB>.BIT(15:0)?",
S<B.ADR.BIT(12:0),
S>C.BIT(15:0),
S>C.ADR.BIT(12:0);
$ADDCONNECT
B<a.DAT.BIT(15:0)    <-   S>A.BIT(15:0),
B<b.DAT.BIT(15:0)    <-   S>C.BIT(15:0),
S>A.ADR.BIT(12:0)    <-   I>.ASA,
S<B.DAT.BIT(15:0)    <-   "?UDSB>.BIT(15:0),?"
    "?UDSB<a.DAT.BIT(15:0)    <-  ?"I>.DSB/
    "?UDSB<b.DAT.BIT(15:0)    <-  ?"B>.BIT(15:0)
    "?,UDSB>.MPX.BIT(0:0)    <-   I>.MDSB?",
S<B.ADR.BIT(12:0)    <-   I>.ASB,
S>C.ADR.BIT(12:0)    <-   I>.ASC;
```

DETAILS OF THE SCR FILE

More information has now been added that must be explained. The first line of the ADDMODULE section describes the dyadic operator "B".

```
B>(*[1])
|  |  |___execution time for "*" (default is 1)
|  |_____function "*" of function list of "B>"
|________output of "B"; no name has been added
          to B, because it is the first copy.
```

The next lines describe the three ports of B.

The microinstruction I has been extended by two more address fields, ASA (for port S>A) and ASC (for port S>C), and for the MUX address field called MDSB. MDSB, you will recall, refers to the Multiplexer select field for the Data port of the storage module S called <B.

The next new module is the MUX UDSB with two input ports, one address port and the output port.

The "? ... ?" comments the MUX out for the purpose of using this description as input to other MSS tools. In some the MUX U is not explicitly known. For our purpose of looking at the structure, we simply ignore the comment characters.

ADDCONNECT shows all the connections necessary to execute L60 and L70 sequentially. Note the connections made to the MUX inputs.

EXERCISE 3
Draw a block diagram that describes (or is described by) the above SCR file.

The next two MIMOLA statements should not cause us any problem.

EXERCISE 4
Bind instructions L80 and L90, using as much hardware as is described in the last SCR file. Write the expected additional lines in the GEN and the SCR files. Then compare your result with the following listings.

```
*LIST /GEZ/PRE.INP

$ADDMODULE
S(8191:0).BIT(15:0)WORD.MOREPORT;
$PROGRAM
BEGIN
L60  S(n) := 1;
L70  S(x) := S(c) / S(t) -> B(*);
L80  S(s) := S(x);
L90  S(a) := S(x);
END
```

```
*LIST /GEZ/PRE.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN


L60.1
          S<B(n):=1;

L70.1
          S<B(x):=S>A(c)/S>C(t)->B(*);

L80.1
          S<B(s):=S>A(x);

L90.1
          S<B(a):=S>A(x);


END
```

```
*LIST /GEZ/PRE.SCR
```

```
$ADDMODULE
B>(*[1,0]),
B>.BIT(15:0),
B<a.DAT.BIT(15:0),
B<b.DAT.BIT(15:0),
I>.BIT(12:0)ASB,
I>.BIT(28:13)DSB,
I>.BIT(41:29)ASA,
I>.BIT(54:42)ASC,
I>.BIT(56:55)MDSB,
S>A(8191:0),
S>A.BIT(15:0),
S>A.ADR.BIT(12:0),
S<B.DAT.BIT(15:0)
    "?,UDSB<a.DAT.BIT(15:0),
    UDSB<b.DAT.BIT(15:0),
    UDSB<c.DAT.BIT(15:0),
    UDSB>.MPX.BIT(1:0),
    UDSB>.BIT(15:0)?",
S<B.ADR.BIT(12:0),
S>C.BIT(15:0),
S>C.ADR.BIT(12:0);
$ADDCONNECT
B<a.DAT.BIT(15:0)    <-   S>A.BIT(15:0),
B<b.DAT.BIT(15:0)    <-   S>C.BIT(15:0),
S>A.ADR.BIT(12:0)    <-   I>.ASA,
S<B.DAT.BIT(15:0)    <-   "?UDSB>.BIT(15:0),?"
    "?UDSB<a.DAT.BIT(15:0)     <-  ?"I>.DSB/
    "?UDSB<b.DAT.BIT(15:0)     <-  ?"B>.BIT(15:0)/
    "?UDSB<c.DAT.BIT(15:0)     <-  ?"S>A.BIT(15:0)
    "?,UDSB>.MPX.BIT(1:0)     <-   I>.MDSB?",
S<B.ADR.BIT(12:0)    <-   I>.ASB,
S>C.ADR.BIT(12:0)    <-   I>.ASC;
```

Simple, isn't it? This may be the time to try running MIMB on your own input file. Refer to Appendix D for a listing of available systems and documentation.

V. PARALLELISM

WHAT CAN BE DONE IN PARALLEL?

In instruction L70 we saw that three memory ports were used simultaneously. Is this all we can do in parallel? Definitely not. Let us have a look at the last INP file /GEZ/PRE.INP. There is no data dependency between instructions L60 and L70, so we will combine them.

```
          L60P S(n) := 1, S(x) := S(c)/S(t) -> B(*);
```

The comma separates parallel statements; both are executed in the same clock cycle. Note that the cycle ends when storing the results into S(n) and S(x). All storage operations occur at the same instant, so an exchange of values between two locations in a memory may be expressed in MIMOLA as:

```
 Lex        S(a) := S(b),
            S(b) := S(a);
```

Now let us run L60P through MIMB:

```
*LIST /GEZ/PRF.INP

$ADDMODULE
S(8191:0).BIT(15:0)WORD.MOREPORT;
$PROGRAM
BEGIN
L60P  S(n) := 1,
      S(x) := S(c) / S(t) -> B(*);
END


*LIST /GEZ/PRF.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN


L60P.1
           S<B(n):=1,
           S<D(x):=S>A(c)/S>C(t)->B(*);


END
```

The GEN file shows us that another port S<D was created and the mux was not needed; otherwise nothing has changed. It should be obvious that the execution time is now less than in the sequential version.

```
*LIST /GEZ/PRF.SCR

$ADDMODULE
B>(*[1,0]),
B>.BIT(15:0),
B<a.DAT.BIT(15:0),
B<b.DAT.BIT(15:0),
I>.BIT(12:0)ASB,
I>.BIT(28:13)DSB,
I>.BIT(41:29)ASA,
I>.BIT(54:42)ASC,
I>.BIT(67:55)ASD,
S>A(8191:0),
S>A.BIT(15:0),
S>A.ADR.BIT(12:0),
S<B.DAT.BIT(15:0),
S<B.ADR.BIT(12:0),
S>C.BIT(15:0),
S>C.ADR.BIT(12:0),
S<D.DAT.BIT(15:0),
S<D.ADR.BIT(12:0);
$ADDCONNECT
B<a.DAT.BIT(15:0)   <-   S>A.BIT(15:0),
B<b.DAT.BIT(15:0)   <-   S>C.BIT(15:0),
S>A.ADR.BIT(12:0)   <-   I>.ASA,
S<B.DAT.BIT(15:0)   <-   I>.DSB,
S<B.ADR.BIT(12:0)   <-   I>.ASB,
S>C.ADR.BIT(12:0)   <-   I>.ASC,
S<D.DAT.BIT(15:0)   <-   B>.BIT(15:0),
S<D.ADR.BIT(12:0)   <-   I>.ASD;
```

PARALLELISM WITH DATA DEPENDENCIES

Now, let us go back to our sequential program, /GEZ/PRE.INP. It looks like L80 cannot be executed before L70 has been completed, because the new value of S(x) has to be used. However, there is an alternative: Instead of using S(x), we can use the expression of L70 as a source! This is possible only because MIMOLA allows multiple assignments to occur in a single statement. The same technique may be used with L90 also. There are no data dependencies, and thus, we can execute all four statements in one instruction:

```
*LIST /GEZ/PRG.INP

$ADDMODULE
S(8191:0).BIT(15:0)WORD.MOREPORT;
$PROGRAM
BEGIN
L60PP  S(n) := 1,
       S(x) := S(c) / S(t) -> B(*),
       S(s) := S(c) / S(t) -> B(*),
       S(a) := S(c) / S(t) -> B(*);
END
```

This looks like it may be very expensive! Three B operators and a total of 10 memory ports could be used. The MSS system is intelligent enough to recognize that the same expression is calculated in three places and that the result from one could be shared with the other two. If we look at the corresponding GEN file, we see that the allocator has assigned the same memory ports and B operator to the three expressions. Since this all occurs in a single cycle, the sharing of hardware is implied.

```
*LIST /GEZ/PRG.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN


L6OPP.1
          S<B(n):=1,
          S<D(x):=S>A(c)/S>C(t)->B(*),
          S<E(s):=S>A(c)/S>C(t)->B(*),
          S<F(a):=S>A(c)/S>C(t)->B(*);


END
```

Everything in the file has been explained before. This may be a good time to review some of the earlier explanations if you do not understand all of it.

```
*LIST /GEZ/PRG.SCR

$ADDMODULE
B>(*[1,0]),
B>.BIT(15:0),
B<a.DAT.BIT(15:0),
B<b.DAT.BIT(15:0),
I>.BIT(12:0)ASB,
I>.BIT(28:13)DSB,
I>.BIT(41:29)ASA,
I>.BIT(54:42)ASC,
I>.BIT(67:55)ASD,
I>.BIT(80:68)ASE,
I>.BIT(93:81)ASF,
S>A(8191:0),
S>A.BIT(15:0),
S>A.ADR.BIT(12:0),
S<B.DAT.BIT(15:0),
S<B.ADR.BIT(12:0),
S>C.BIT(15:0),
S>C.ADR.BIT(12:0),
S<D.DAT.BIT(15:0),
S<D.ADR.BIT(12:0),
S<E.DAT.BIT(15:0),
S<E.ADR.BIT(12:0),
S<F.DAT.BIT(15:0),
S<F.ADR.BIT(12:0);
$ADDCONNECT
B<a.DAT.BIT(15:0)    <-   S>A.BIT(15:0),
B<b.DAT.BIT(15:0)    <-   S>C.BIT(15:0),
S>A.ADR.BIT(12:0)    <-   I>.ASA,
S<B.DAT.BIT(15:0)    <-   I>.DSB,
S<B.ADR.BIT(12:0)    <-   I>.ASB,
S>C.ADR.BIT(12:0)    <-   I>.ASC,
S<D.DAT.BIT(15:0)    <-   B>.BIT(15:0),
S<D.ADR.BIT(12:0)    <-   I>.ASD,
S<E.DAT.BIT(15:0)    <-   B>.BIT(15:0),
S<E.ADR.BIT(12:0)    <-   I>.ASE,
S<F.DAT.BIT(15:0)    <-   B>.BIT(15:0),
S<F.ADR.BIT(12:0)    <-   I>.ASF;
```

Sure, we added two more memory ports, addresses and the necessary data paths, but the hardware will also execute much faster. The actual increase will be determined later.

VI. WHAT HAVE WE LEARNED SO FAR?

We have seen two types of devices, memories (S) and dyadic operators (B), and how we define their properties, such as bit width and number of ports. The behavioral descriptions we have seen use a postfix notation and may express both parallel (indicated by a comma) or sequential (indicated by a semicolon) operations.

We have learned how to interpret the GEN file, which contains the bound program (the microprogram). We also learned how to read the structural description in the SCR file. We saw how the Allocator transforms behavioral descriptions into structures that can execute the behavior (microprogram).

So far all changes that were made to the design were due to modification of the hardware structure or the combination of sequential operations into parallel. These changes were made manually. What we will see now is an automatic process performed by the MSS.

VII. AN INTRODUCTION TO THE COMPILER

The Compiler is the part of the MSS that sequentializes an elementary statement block (ESB), if there is not enough hardware available to execute it. One method of restricting hardware is to define it in the ADDMODULE section. Other methods will be discussed later.

REGISTERS

Before providing an example of the Compiler 'in action', we introduce the use of registers. Registers are single vector storage units, and their names always begin with a capital 'R'. For example, R1, Rtemp, and RSTATUS are all valid register names.

A NEW EXAMPLE

We will present a new example in this discussion of the Compiler for a number of reasons. First, the reader will see the use of registers. Second, we do not want to obscure the focus of the sin example. Finally, the sin example is used later to demonstrate a slightly different aspect of the Compiler.

First we look at a very simple example. We want to sum the contents of R2, R3, R4, and R5 and put the result in R1. Obviously, this requires three adds, and thus, three B-operators in hardware.

```
*LIST /MGA/G.INP

$PROGRAM
BEGIN
LO  R1:=R2 / R3 -> B(+) / R4 -> B(+) / R5 -> B(+);
END
```

After running this file through the MSS, the GEN file is what we would expect.

```
*LIST /MGA/G.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
 $PASS   1
 $PROGRAM


BEGIN


LO.1
          R1:=R2/R3->B(+)/R4->B_A(+)/R5->B_B(+);


END
```

Note that the three B-operators are named B, B_A, and B_B. Remember that the first letter of the device defines what kind of device it is. Thus, B is a two-input device. The hardware is bound with the names _A, _B, _C, ... in alphabetical order. Since we did not name the B-operator, it is just called B. If we had named it, say Badd, the bound devices would have been called Badd, Badd_A, Badd_B,...

We said that we could limit the amount of hardware generated in the ADDMODULE section. To limit the number of B-operators, we use the attribute .DUPLICATE(n), where n is the number of duplicate modules the compiler is allowed to create:

```
*LIST /MGA/G1.INP

$ADDMODULE
  B(+).DUPLICATE(1);

$PROGRAM
BEGIN
LO  R1:=R2 / R3 -> B(+) / R4 -> B(+) / R5 -> B(+);
END
```

This says we will be using a two-input operator, called B, that can perform the function addition. Further, we are limiting the number of those devices to two. The first is B, which we get by declaring it. The second would be B_A, if it was needed.

If the device is not declared, or if .DUPLICATE is not specified, up to 27 devices are available, as needed (B, B_A,...,B_Z). Also available is the attribute .NODUPLICATE, which is equivalent to .DUPLICATE (0).

Now we have limited the number of adders to two. Let's see what happens to the .GEN file:

```
*LIST /MGA/G1.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN


LO.1
          RHLP_101:=R2/R3->B(+)/R4->B_A(+);

LO.2
          R1:=RHLP_101/R5->B(+);
```

```
END
```

We now have two ESBs when we had started with one. We also have something called RHLP_101. The name of this register is HLP_101; HLP means HELP, and the _101 is a numbering system. The next RHLP will be _102. The Compiler created a register to hold temporary values. This was necessary to store the intermediate result of adding the contents of registers R2, R3, and R4. This addition required all adders that are allowed.

So, we see that in one statement we could add R1, R2, and R3 in parallel and put the result in a temporary register, RHLP_101. In another statement we could add R5 to this temporary result and put the final answer in R1. Note that in this case, we could have added R2, R3, and R4, and put that result in R1 and then added R5 to R1. This would eliminate the need for an extra register. However, this cannot be done in all cases, and the compiler is not smart enough to know when it has to use the temporary register and when it can store an intermediate result in the destination. At a later stage in the design, the user could detect this special case and modify the .GEN file to reduce the use of intermediate storage registers.

We now use .DUPLICATE(0) to allow only one adder:

```
*LIST /MGA/G2.INP

$ADDMODULE
  B(+).DUPLICATE(0);

$PROGRAM
BEGIN
LO  R1:=R2 / R3 -> B(+) / R4 -> B(+) / R5 -> B(+);
END
```

The .GEN file shows that the Compiler created three sequential statements to do our addition. This makes sense; we can only do one add with one adder, and we have three adds to do.

```
*LIST /MGA/G2.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


 BEGIN


 LO.1
           RHLP_101:=R2/R3->B(+);
```

```
L0.2
          RHLP_101:=RHLP_101/R4->B(+);

L0.3
          R1:=RHLP_101/R5->B(+);


END
```

By now you are probably wondering where the L0.1, L0.2 and L0.3 came from. The L0 is the label of the statement in the input file. The 1, 2, or 3 appended to L0 shows a count of the statements generated by the single input statement. In the no-limitations case, the GEN file had only statement label L0.1 (only one bound statement). In the last example, we see three bound statements. More will be said about labels later.

This section demonstrates what the Compiler does: split an ESB with a number of parallel operations. Depending on the hardware limitations imposed by the designer, various degrees of sequentialization can be achieved. The next sections explain more about how the compiler works. It also discusses other ways to limit the hardware.

VIII. CONTROLLING THE ALLOCATOR

Remember when we added the MOREPORT attribute? We said that we wanted the Allocator to create more ports. Let us see what would have happened if we did not allow more ports. If we do this in two steps it will be easier to understand.

First we limit the number of ports to three by allowing only two more ports:

```
*LIST /GEZ/PRH.INP

$ADDMODULE
S(8191:0).BIT(15:0)WORD.MOREPORT(2);
$PROGRAM
BEGIN
L60PP  S(n) := 1,
       S(x) := S(c) / S(t) -> B(*),
       S(s) := S(c) / S(t) -> B(*),
       S(a) := S(c) / S(t) -> B(*);
END
```

The result is a bound microprogram:

```
*LIST /GEZ/PRH.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN


L60PP.1
          S<B(n):=1;

L60PP.2
          S<B(x):=S>A(c)/S>C(t)->B(*);

L60PP.3
          S<B(s):=S>A(c)/S>C(t)->B(*);

L60PP.4
          S<B(a):=S>A(c)/S>C(t)->B(*);


END
```

It has four sequential steps, as did the sequential version we started with (/GEZ/PRE.INP). The difference is in the third and fourth lines. The expression is evaluated three times, instead of being copied out of S(x). From a resource standpoint, this is not bad. The ports and B operator are there anyway. We even save a data path from S>A to S<B (L90 in /GEZ/PRE.INP) by doing it this way. The penalty lies in the longer execution time of L60PP.3 and.4, compared with L80 and L90.

Here we see how the insertion of a single limit "(2)" in the defined structure controls the Allocator very efficiently.

IX. THE MSS COMPILER

HOW THE MSS COMPILER WORKS

When the Allocator tried to bind resources to the second statement in L60PP it could have created a second input port, then one output port. When it then tried to create the necessary second output, it hit the resource limit. Since nothing useful could be done in parallel to the first statement, the Allocator called the Compiler for help. The Compiler decided to start a new instruction and gave the ball back to the Allocator.

With all resources available again, because it was now working on a new instruction, the Allocator could find enough resources for the second statement. However, it had to call the Compiler again for the third statement. Control alternates back and forth until we obtain the result shown.

Naturally, the Compiler task is not always so easy and much more checking for data dependencies has to be done. For instance, recall the exchange example.

```
 Lex       S(a) := S(b),
           S(b) := S(a);
```

Here the Compiler would have to insert an intermediate storage cell.

MORE WORK FOR THE COMPILER

Now we go one step further and allow only one input and one output port. We could do this by setting MOREPORT to 1. However, a better way is to name the ports, otherwise we might end up with two output ports and no input, if it happened that two outputs were required before an input. We define ports in ADDMODULE by simply listing them. The INP file shows how:

```
*LIST /GEZ/PRI.INP

$ADDMODULE
S<A(8191:0).BIT(15:0)WORD,
S>B;
$PROGRAM
BEGIN
L6OPP   S(n) := 1,
        S(x) := S(c) / S(t) -> B(*),
        S(s) := S(c) / S(t) -> B(*),
        S(a) := S(c) / S(t) -> B(*);
END
```

Now we are curious about what the Compiler does with only two ports. The expression with the B-operator now cannot be executed as shown; the Compiler has to split the expression. Instead of guessing what the Compiler will do, we can just look at the results.

```
*LIST /GEZ/PRI.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN
```

```
L60PP.1
           S<A(n):=1,
           RHLP_101:=S>B(c);

L60PP.2
           S<A(x):=RHLP_101/S>B(t)->B(*);

L60PP.3
           S<A(s):=RHLP_101/S>B(t)->B(*);

L60PP.4
           S<A(a):=RHLP_101/S>B(t)->B(*);


END
```

This solution is not too bad. Only four instructions are necessary. The only thing that has changed, compared with PRH.GEN, is that the value of the variable c is stored in RHLP_101 in the first instruction L60PP.1. That is exactly what a normal HLL compiler would do. A rather inefficient feature is that the same expression is evaluated three times.

We have already seen RHLP used for temporary storage when a statement is being split. In this example we are limited by the number of memory ports allowed. In the example where we added four numbers, we were limited by the number of B operators. It does not matter to the Compiler what the hardware limitations are; it only knows that the ESB must be split.

Let us now have a look at the SCR file:

```
*LIST /GEZ/PRI.SCR

$ADDMODULE
B>(*[1,0]),
B>.BIT(15:0),
B<a.DAT.BIT(15:0)
B<b.DAT.BIT(15:0),
I>.BIT(12:0)ASA,
I>.BIT(28:13)DSA,
I>.BIT(41:29)ASB,
I>.BIT(42:42)MDBa,
I>.BIT(43:43)MDSA,
RHLP_101<.DAT.BIT(15:0),
RHLP_101>.BIT(15:0),
S>B(8191:0),
S<A.DAT.BIT(15:0)
    "?,UDSA<a.DAT.BIT(15:0),
    UDSA<b.DAT.BIT(15:0),
    UDSA>.MPX.BIT(0:0),
    UDSA>.BIT(15:0)?",
S<A.ADR.BIT(12:0),
S>B.BIT(15:0),
S>B.ADR.BIT(12:0);
$ADDCONNECT
B<a.DAT.BIT(15:0)   <-   "?UDBa>.BIT(15:0),?"
B<b.DAT.BIT(15:0)   <-   S>B.BIT(15:0),
RHLP_101<.DAT.BIT(15:0)  <-   S>B.BIT(15:0),
S<A.DAT.BIT(15:0)   <-   "?UDSA>.BIT(15:0),?"
    "?UDSA<a.DAT.BIT(15:0)    <-  ?"I>.DSA/
    "?UDSA<b.DAT.BIT(15:0)    <-  ?"B>.BIT(15:0)
    "?,UDSA>.MPX.BIT(0:0)    <-   I>.MDSA?",
S<A.ADR.BIT(12:0)   <-   I>.ASA,
S>B.ADR.BIT(12:0)   <-   I>.ASB;
```

This should be compared to file PRE.SCR, which showed the structure for the sequential version of our behavioral description. The difference is not very large; we saved a memory port at the cost of an additional register. In most technologies this would be a cost saving. Compared with PRE.SCR, this version requires 13 fewer bits for the microinstruction vector.

Also gained through the parallel description of the algorithm is the flexibility to generate three different hardware structures (PRG, PRH, and PRI), with only a small change in the ADDMODULE section of the input. The three versions represent different solutions for cost and speed. In real examples, the number of possible versions is naturally much higher.

X. TRANSLATING THE LOOP

LOOK FOR PARALLELISM

So far we have only translated four lines of our BASIC program. The next line, 100, is the beginning of a loop that ends at line 150.

```
*LIST PRIMER_1.B

10  print 'This Program calculates sin(c*t)'
20  print ' c = ';
30  input c
40  print ' t = ';
50  input t
60  n = 1
70  x = c * t
80  s = x
90  a = x
100 if abs(a) < .004 then goto 200
110 a = -a * x * x / (2 * n * ( 2 * n + 1 ) )
```

```
120 s = s + a
130 print 'n = ';n,'    a = ';a, '    s = ';s
140 n = n + 1
150 goto 100
200 print 'sin( ';x;' ) = ';s
210 print ' type: 0 for quit, 1 for new c and 2 for new t'
220 input i
230 if i = 1 then goto 20
240 if i = 2 then goto 40
250 end
```

Since we have seen that MIMOLA can express much more parallelism than BASIC, we first analyze the loop for possible parallel execution. In line 100 we use variable 'a', and in line 110 we change 'a '. Since the change of 'a' occurs synchronously at the end of instruction 110, 'a' has its old value during the instruction, and thus, we can execute 100 and 110 in parallel.

Lines 110 to 150 are executed only if "ABS(A)<.004" is false. If it is true, control jumps to line 200. This can be expressed as:

```
*LIST /GEZ/PRJ.T

L100   IF S(a) -> A(.abs) / .004 -> B(>=)
         THEN  "EXECUTE LINE 110 IN BASIC PROGRAM"
       FI;
```

Let us explain this line by line. First we see the "IF...THEN...FI" construct. The only unusual thing is the "FI", which closes the THEN clause. Other HLL's use the BEGIN-END construct for this purpose. In MIMOLA the "FI" must be in the same instruction, because the condition value is not stored!

We have exchanged the BASIC THEN GOTO with a THEN followed by the executable code. Therefore, we must invert the condition from < to >=.

Let us digress at this point to discuss operators.

MONADIC EXPRESSIONS

The function abs(a) has been translated into a monadic expression. This means that there is only one operand, S(a), for the monadic operator A.

```
 ---------------------------------------------------
 |  monadic expression:  operand -> A-operator     |
 |  A-operator         :  A name (function)        |
 ---------------------------------------------------
```

The function of the operator A has been selected as ".abs". (We do not need to worry about how ".abs" is implemented.)

```
          abs(a)      --->      S(a) -> A(.abs)
```

DYADIC OPERATORS

The comparison translates into a dyadic expression. The first operand is now the expression "S(a) -> A(.abs)". In hardware we could think of using the output of the functional unit A as the left input of the functional unit B.

```
abs(a) < .004      --->      S(a) -> A(.abs)/.004 -> B(>=)
```

EXERCISE 5
Translate another expression:
```
(a+b) * c + d
```
Do it operator by operator from left to right!

See Appendix B for the solution. No problems yet? Then try some more examples.

EXERCISE 6
Translate

```
- a + b
abs (a + b)
(- abs (a + b)) * c
```

Do it stepwise!

Any problems? Then look at the solutions in Appendix B and try to simulate the MIMOLA expressions by hand. Remember, this is postfix notation. That means that the operands are followed by the operator. Thus, start scanning from left to right until you find the first operator (A or B), take one (or two, respectively) operand to the left and execute the function. Then replace operands plus operator with the result, which is now an operand, and proceed to the next operator. If the expression was correct, only one operand will be left at the end.

GOTO

The rest of line 100 is the GOTO 200, which simply translates into GOTO L200 in MIMOLA. GOTO is considered a MACRO and is not converted to hardware by the Allocator. Later we will have to replace it by an expression modifying a program counter. We will handle that when we design the control part.

```
---------------------------------------------
|   GOTO is a MACRO                          |
|   MACROS do not generate hardware          |
|   MACROS have to be replaced or expanded   |
|   Other MACROS are:  WHILE                 |
|                      FOR FROM BY TO DO OD  |
|                      CALL                  |
---------------------------------------------
```

THE FIRST TRY

Now we are ready to translate line 110 of the BASIC program:

```
     110  A = -A * X * X / (2 * N * ( 2 * N + 1 ) )
```

With the above procedure we get:

```
*LIST /GEZ/PRJ.T1

          S(a) := S(a) -> A(-) / S(x) -> B(*)
                  / S(x) -> B(*)
                  / 2 / S(n) -> B(*)
                  / 2 / S(n) -> B(*) / 1 -> B(+)
                  -> B(*)
                  -> B(/)
```

Line breaks have no significance in MIMOLA. Now we can combine lines 100 and 110 to get:

```
*LIST /GEZ/PRJ.T2

L100   IF S(a) -> A(.abs) / .004 -> B(>=)
          THEN S(a) := S(a) -> A(-) / S(x) -> B(*)
                       / S(x) -> B(*)
                       / 2 / S(n) -> B(*)
                       / 2 / S(n) -> B(*) / 1 -> B(+)
                       -> B(*)
                       -> B(/)
       FI;
```

As we did before, we try to extend the parallelism again. Although line 120

```
     120  s = s + a
```

uses the result "a" of line 110, we already know how to deal with such problems (see Parallelism With Data Dependencies, above). All we have to do is replace "a" by the expression for "a" in line 110. This could require a large amount of typing, but there is a short hand method supported by the MSS.

TEMPORARY STORAGE

We can give the result of an expression any name starting with "V" and then use this V name instead of the expression in following expressions. It is important to note that "Vname" is only valid throughout the instruction in which it is defined! "V" is not a storage device. In hardware, "V" can be interpreted as a temporary name for a data path or signal. The syntax for assigning a temporary value (of an operand) to V is

operand = Vname

and in MIMOLA, any expression can be an operand. After its definition, V can be used as if it were an operand.

In our example we can now extend instruction L100 by adding line 120 as follows:

```
*LIST /GEZ/PRJ.T3

L100   IF S(a) -> A(.abs) / .004 -> B(>=)
          THEN S(a) := S(a) -> A(-) / S(x) -> B(*)
                       / S(x) -> B(*)
                       / 2 / S(n) -> B(*)
                       / 2 / S(n) -> B(*) / 1 -> B(+)
                       -> B(*)
                       -> B(/)
                       = V1,
               S(s) := S(s) / V1 -> B(+)
       FI;
```

We skip line 130, because it is really not part of the hardware it is used only to 'see' what the answer is. Line 140 is independent of all other lines in L100, and thus, it can be executed in parallel. We will also put line 150 in as another parallel statement. Since 150 was a jump to line 100, it is now converted to a jump on itself, signaling that the program counter is not to be incremented.

```
*LIST /GEZ/PRJ.INP

$ADDMODULE
S<A(8191:0).BIT(15:0)WORD,
S<B, S<C, S<D, S>E, S>F, S>G, S>H;
$PROGRAM
BEGIN
L60PP  S(n) := 1,
       S(x) := S(c) / S(t) -> B(*),
       S(s) := S(c) / S(t) -> B(*),
       S(a) := S(c) / S(t) -> B(*);
L100   IF S(a) -> A(.abs) / .004 -> B(>=)
          THEN S(a) := S(a) -> A(-) / S(x) -> B(*)
```

```
                                / S(x) -> B(*)
                                / 2 / S(n) -> B(*)
                                / 2 / S(n) -> B(*) / 1 -> B(+)
                                -> B(*)
                                -> B(/)
                                = V1,
                     S(s) := S(s) / V1 -> B(+),
                     S(n) := S(n) / 1 -> B(+),
                     GOTO L100
              FI;
L200    X;
END
```

DUMMY OPERANDS

In the above listing we added an instruction, L200, as an exit for the loop. Since our BASIC program did not contain anything we wanted to include in MIMOLA, we make this instruction a NOP. In order to fulfill MIMOLA syntax, we use the dummy operand "X", which tells the allocator not to create any hardware. The X can be used anywhere that an operand is necessary for syntactic reasons. An example is the use of a B-operator for a monadic operation.

```
      S(a)/X -> B(.abs_a)
```

would be another way to execute the function abs(a).

The function .abs_a calculates the absolute value of the left operand of B. The right operand of B has no influence on the result, and thus, is a don't-care.

```
              -------------------------------------
              |  X is a NO-OP used for 'filling',  |
              |  to satisfy syntax rules.          |
              -------------------------------------
```

XI. THE EXAMPLE COMPLETED

In the PRJ.INP file shown above, you may have noticed a small change in the ADDMODULE section. Instead of saying "MOREPORT" we have explicitly listed eight ports of S, which gives us more control over the naming of ports.

We now have a complete behavioral description of an algorithm to calculate sin(c*t). We can imagine that in a real problem we would not build a machine for just this one function. We might add more algorithms to the description or embed this one as a macro or subroutine into a larger program. Hopefully, the example has shown us that we are able to manage the use of MIMOLA to describe algorithms.

Let us now use our program as an input to MIMB. The GEN file will immediately tell us if we have allocated enough resources:

```
*LIST /GEZ/PRJ.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN


L60PP.1
          S<A(n):=1,
          S<B(x):=S>E(c)/S>F(t)->B(*),
          S<C(s):=S>E(c)/S>F(t)->B(*),
          S<D(a):=S>E(c)/S>F(t)->B(*);

L100.1

          IF S>E(a)->A(.abs)/.004->B(>=)
            THEN
              S<A(a):=S>E(a)->A_A(-)/S>F(x)->B_A(*)/S>F(x)->B_B(*)
              /2/S>G(n)->B_C(*)/2/S>G(n)->B_C(*)/1->B_D(+)->B_E(*)
              ->B_F(/)=V1,
              S<B(s):=S>H(s)/V1->B_G(+),
              S<C(n):=S>G(n)/1->B_H(+),
              GOTO L100.1 FI
          ;

L200.1
          X;


END
```

Since none of the instructions have been split, we know that we had enough resources, namely memory ports.

Taking a close look at L100, we can see that the expression 2 * n, which appears twice, has been allocated to the same hardware in both cases:

```
          2/S>G(n) -> B_C(*)
```

Additionally, the system has allocated port S>E to both occurrences of "a" in the condition, and in the expression. If you look carefully you will find further examples of the allocation of the same hardware to common subexpressions. Thus, we would not have saved anything by extracting these common subexpressions manually.

We also count nine B-operators. What we expect, therefore, is that the SCR file with the structure is quite long. It is, so we have printed it in Appendix C as /GEZ/PRJ.SCR. Be sure to take a look at it; it should be self-explanatory by now. The main thing we see is a very expensive hardware configuration for a single function. What we want to know now is if this can really be justified.

XII. MEASURING PERFORMANCE

MIMB has to know two things in order to calculate performance:

- Logic delays
- Dynamic behavior of the algorithm

LOGIC DELAYS

Let us first enter the logic delays. In the last SCR file (Appendix C) we saw a list of functions of each operator in the ADDMODULE section These functions take different times to execute, and the default assumption is one unit, as in ".abs [1]".

Let us now assume that the units used are nanoseconds. Let us also assume execution times for the functions, e.g., 20 for the .abs function. To define the delay we simply write:

```
          .abs[20]
```

The ADDMODULE section now shows all function execution times of the A and B operators. In this example, we simply guessed at the times. If the technology is known, these can be determined more precisely.

```
*LIST /GEZ/PRL.T

$ADDMODULE
S<A(8191:0).BIT(15:0)WORD,
S<B, S<C, S<D, S>E, S>F, S>G, S>H,

A(.abs[20], -[15]).DUPLICATE,
B(+[25], -[30], *[100], /[150], <[30]).DUPLICATE;
```

Notice the attribute .DUPLICATE after the declaration of functional modules. This allows duplicates to be made, if necessary. Otherwise A and B would be the only operators.

The memory read and load delays are still missing. We simply add a line and use the standard functions .READ and .LOAD to define times. Delays for multiplexers cannot be declared, so we assume we have included these delays into the already defined ones.

```
-----------------------------------------
| .DUPLICATE allows hardware modules    |
| to be created as necessary.           |
-----------------------------------------
```

AN EXAMPLE

In order to better understand the time calculation, we now return to our simple example with only one instruction:

```
*LIST /GEZ/PRL.INP

$ADDMODULE
S<A(8191:0).BIT(15:0)WORD,
S<B, S<C, S<D, S>E, S>F, S>G, S>H,
S(.READ[50], .LOAD[75]),
A(.abs[20], -[15]).DUPLICATE,
B(+[25], -[30], *[100], /[150], <[30]).DUPLICATE;
$PROGRAM
BEGIN
L6OPP   S(n) := 1,
        S(x) := S(c) / S(t) -> B(*),
        S(s) := S(c) / S(t) -> B(*),
        S(a) := S(c) / S(t) -> B(*);
END
```

Let us try to calculate the resulting execution time. We assume that it takes no time to read fields in the microinstruction. Thus, the first statement "S(n) := 1" takes only 75 units (.LOAD) to store the 1 (which is in a microinstruction field) in S(n).

The next three statements are equal, from a timing standpoint. The two operands are read in parallel, requiring only 50 units to read S. 100 units have to be added for the multiplication, and 75 for loading S. This is a total of 225 units.

Since 225 is more than the 75 for the first statement, and since both are executed in parallel, the longest time is chosen to determine the necessary execution time of L6OPP.

In a design with more than one ESB, we calculate the time of each ESB in a similar fashion. The micro-instruction cycle time may vary, as the timing requirements of a particular ESB change. Thus, assume we have a two ESB design and one ESB takes 10 units to execute, and the other, 20. The total amount of time to execute these two ESBs would be 10 + 20 = 30 units. If we had a constant cycle time, our design would require 20 + 20 = 40 units.

The MSS always adds one time unit for the microinstruction handling, which we can ignore at this point. We can find the "ESTIMATED RUN TIME" in a file called OTB.

```
*LIST /GEZ/PRL.OTB

MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
***      1 ESBS READ    (          1 WEIGHTED)
***      0 ESBS CREATED (          0 WEIGHTED)
***      1 ESBS TOTAL   (          1 WEIGHTED)
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
  0 HELP-STORAGECELLS (OR REGISTERS) USED IN PROGRAM
ESTIMATED RUN TIME:        226
```

This file contains much more information, which we will partially explain later.

```
-------------------------------------------------
|  Input and Output files we have looked at:    |
|    INP - the MIMOLA input                     |
|    GEN - the bound statements (output)        |
|    SCR - hardware and connections (output)    |
|    OTB - statistics about program and hardware |
|            (output)                           |
-------------------------------------------------
```

MODELING THE DYNAMIC BEHAVIOR

In order to determine how long it takes to execute the total program, we have to enter information about the dynamic behavior of the program. Remember that the MIMOLA program has not yet been executed or simulated; we have only executed the equivalent BASIC program. So, let us go back and look at the number of iterations necessary to achieve the result. We carried the variable n and incremented it in the loop for this reason. The results are listed in the following table.

c*t	n
0	0
0.5	2
1	3
1.5	3
2	4

c*t	n
3	6
4	7
5	8
6	10
10	15

Assuming that the argument is between 0 and 2pi = ~ 6.28 (one full cycle of sin), a mean value of five iterations seems reasonable. We can then say that for every time sin is calculated, L60PP is executed once and L100 five times.

THE FACTOR COMMAND

We specify this relationship by inserting the following line before L100.

```
          $FACTOR = 5
```

This FACTOR will be used for all succeeding instructions until another $FACTOR command is found. So we enter a

```
          $FACTOR = 1
```

in front of L200, which is outside of the loop. Now we have declared the dynamic behavior of the algorithm.

There are other ways to determine the FACTORs. One is to calculate them by analytical or statistical means. Often the number of iterations of loops is known to the programmer. A second way is to use the MIMOLA simulator, which is described elsewhere /11/. This simulator collects dynamic behavior figures.

```
*LIST /GEZ/PRN.INP

$ADDMODULE
S<A(8191:0).BIT(15:0)WORD,
S<B, S<C, S<D, S>E, S>F, S>G, S>H,
S(.READ[50], .LOAD[75]),
A(.abs[20], -[15]).DUPLICATE,
B(+[25], -[30], *[100], /[150], >[30]).DUPLICATE;
$PROGRAM
BEGIN
L60PP  S(n) := 1,
       S(x) := S(c) / S(t) -> B(*),
       S(s) := S(c) / S(t) -> B(*),
       S(a) := S(c) / S(t) -> B(*);
$FACTOR=5
L100   IF S(a) -> A(.abs) / .004 -> B(>)
```

```
          THEN
           S(a) := S(a) -> A(-) / S(x) -> B(*)
                       / S(x) -> B(*)
                       / 2 / S(n) -> B(*)
                       / 2 / S(n) -> B(*) / 1 -> B(+)
                       -> B(*)
                       -> B(/)
                       = V1,
               S(s) := S(s) / V1 -> B(+),
               S(n) := S(n) / 1 -> B(+),
               GOTO L100
          FI;
$FACTOR=1
L200   X;
END
```

With this input file, the first few lines of the OTB file look like this:

```
*LIST /GEZ/PRN.OTB

MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
***     3 ESBS READ    (         7 WEIGHTED)
***     0 ESBS CREATED (         0 WEIGHTED)
***     3 ESBS TOTAL   (         7 WEIGHTED)
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
  0 HELP-STORAGECELLS (OR REGISTERS) USED IN PROGRAM
ESTIMATED RUN TIME:       2858
```

The dynamic runtime of our program is 2858 ns for each calculation of sin(x). This is our performance figure. This is as fast as we can get with the declared function execution times. However, another look at the structure (file /GEZ/PRJ.SCR in Appendix C) will convince us that the hardware cost is much too high.

XIII. REDUCING THE COST

We have lowered the cost before by reducing the number of hardware resources. But instead of doing it blindly, we now have a means to find the resources that are not used very often. If we get rid of those resources, we probably will not affect the performance as much as if we delete highly used resources.

EXPLAINING THE OTB FILE

We find the information to do this in the OTB file. Because of its size, we can only look at parts of it here.

```
------------------------------------------------------------------------
*** GROUP : B
 B         18 FREE DUPLICATES                 , $:   5.0 +  4.5 = $  9.5
   U/C:    9.0226                                FREQ:       6 =    85.71%
   FUNCTIONS :  -[30], *[100], /[150], >[30], +[25],
   PORT USE:  OUTPUT 0 :     6X  1 :     1X
              INPUT  0 :     1X  2 :    6X
    >                        .AUTO(.READ      ) FREQ:        6 =    85.71%
           5 FUNCTIONS(EXCL .LOAD&.READ)
      DATA      15:  0                FREQ:       1 = 14.29 %
      FUNCTION  2:  0    ADR  D-BITS     MODULE&PORT   S-BITS   FREQ
                          0     2:  0    I>           FB        6 = 85.7%
   < a                      .AUTO(NONE       ) FREQ:  6 =   85.71%
      DATA      15:  0    ADR  D-BITS     MODULE&PORT   S-BITS   FREQ
                          0    15:  0    A>             15:  0  5 = 71.4%
                          1    15:  0    S>E            15:  0  1 = 14.3%
   < b                      .AUTO(NONE       ) FREQ:  6 =   85.71%
      DATA      15:  0    ADR  D-BITS     MODULE&PORT   S-BITS   FREQ
                          0    15:  0    I>           REAL      5 = 71.4%
                          1    15:  0    S>F            15:  0  1 = 14.3%
------------------------------------------------------------------------
```

We get this block of information for every module. We will now look at the module B, which is one of the dyadic operators in group B.

Let us first look at the frequency of use of module B. We can see that

```
          FREQ:  6 = 85.71%
```

This information is repeated for each of the three ports of B: the output >, and the two inputs <a and <b. The frequency is the same for each, because we used all of the ports of this functional unit all of the time. This is usually true for A and B type operators, but may not be true for multi-port memories.

The percentage of 85.71% is calculated by dividing the number of uses of B (6) by the total number of executed instructions. This total number is 7, because we execute L60PP and L200 once, and L100 five times. We see this information in the header of OTB as X:

```
     n ESBS TOTAL (X WEIGHTED)
```

*LIST /GEZ/PRN.OTB

```
MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
***     3 ESBS READ     (        7 WEIGHTED)
***     0 ESBS CREATED  (        0 WEIGHTED)
***     3 ESBS TOTAL    (        7 WEIGHTED)
```

The symbol n represents the number of ESBs before the FACTORs are applied, so n is the number of microinstructions that have to be stored. X is the factored or dynamic number of microinstructions executed.

The line ESBs READ provides the same information about the input (the INP file). The ESBs CREATED line is for the additional instruction generated by the Compiler. A zero in the CREATED line means that enough hardware resources were given and the Compiler did not have to be called.

Now let us look back at the GROUP B output of OTB. The line 'FUNCTIONS:' lists the set of functions that can be executed by this module, together with the assigned delays.

For Port > (the output) the interesting line is FUNCTION. This describes the function select input of B, and says that it is three bits wide (2:0), as is necessary for encoding the five functions of B. The next line shows that I>FB is a source for the function control input. This is, as we already know, the field FB of the microinstruction vector.

Input port <a has only a DATA connection that is 16 bits wide (15:0). We find two sources in the following lines, with the ADR (multiplexer address) 0 and 1: A> and S>E. The interesting information is how often these sources, and thus, the data path to the sources are used. We find these data at the end of the lines, 71.4% and 14.3% in this case.

Two inputs to the same port require the creation of a multiplexer. We see this if we look at the proper section of the SCR file:

```
$ADDCONNECT
A>.FCT.BIT(0:0)    <-   I>.FA,
A<.DAT.BIT(15:0)    <-   S>E.BIT(15:0),
A_A>.FCT.BIT(0:0)   <-   I>.FA_A,
A_A<.DAT.BIT(15:0)   <-   S>E.BIT(15:0),
B>.FCT.BIT(2:0)   <-   I>.FB,
B<a.DAT.BIT(15:0)   <-   "?UDBa>.BIT(15:0),?"
    "?UDBa<a.DAT.BIT(15:0)    <-  ?"S>E.BIT(15:0)/
    "?UDBa<b.DAT.BIT(15:0)    <-  ?"A>.BIT(15:0)
    "?,UDBa>.MPX.BIT(0:0)    <-   I>.MDBa?",
B<b.DAT.BIT(15:0)   <-   "?UDBb>.BIT(15:0),?"
    "?UDBb<a.DAT.BIT(15:0)    <-  ?"S>F.BIT(15:0)/
    "?UDBb<b.DAT.BIT(15:0)    <-  ?"I>.REAL
    "?,UDBb>.MPX.BIT(0:0)    <-   I>.MDBb?",
```

The most useful information in the OTB file is the frequencies, which give us an understanding of the dynamic utilization of resources. In our example there are no dramatic differences, because the sample is too small. In real designs we see a much larger variance and are able to find infrequently utilized or expensive devices, which would be candidates for deletion and the assignment of their functions to other modules.

SOME HARDWARE RESTRICTIONS

Let us try an experiment. We chose to restrict the hardware to two B-operators and three ports for S. This can be done with the following declaration:

```
*LIST /GEZ/PRO.T

$ADDMODULE
S<A(8191:0).BIT(15:0)WORD,
S>B, S>C,
S(.READ[50], .LOAD[75]),
A(.abs[20], -[15]).DUPLICATE,
B(+[25], -[30], *[100], /[150], >[30]).DUPLICATE(1);
$PROGRAM
  ...
```

Now we are anxious to look at the performance:

```
*LIST /GEZ/PRO.OTB

MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
***     3 ESBS READ    (         7 WEIGHTED)
***     9 ESBS CREATED (        33 WEIGHTED)
***    12 ESBS TOTAL   (        40 WEIGHTED)
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
  2 HELP-STORAGECELLS (OR REGISTERS) USED IN PROGRAM
ESTIMATED RUN TIME:      7381
```

The execution time is now 7381. Compared with 2858 from the design without hardware restrictions, this is slower by a factor of 2.6. We also see that we have to store 12 instructions instead of three, but this fact is partly compensated by a reduction of the microinstruction vector size from 218 to 137. We can find the width of the I vectors by looking in the SCR file:

```
*LIST /GEZ/PRO.T2
```

```
I>.BIT(135:134)MDB_Ab,
I>.BIT(136:136)MDRHLP_101,
RHLP_101<.DAT.BIT(15:0)
    "?,UDRHLP_101<a.DAT.BIT(15:0),
    UDRHLP_101<b.DAT.BIT(15:0),
    UDRHLP_101>.MPX.BIT(0:0),
    UDRHLP_101>.BIT(15:0)?",
RHLP_101>.BIT(15:0),
RHLP_102<.DAT.BIT(15:0),
RHLP_102>.BIT(15:0),
```

We also see that two registers had to be added for intermediate results.

The number of dynamically executed instructions (ESBs WEIGHTED) increases from seven to 40. This is a factor of 5.7, or twice as much as the increase in run time, and can be explained with the decrease in execution time per instruction. In the following listing of the GEN file we can see that the instructions have become much less complex, which is what we expected.

```
*LIST /GEZ/PRO.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN


L60PP.1
          S<A(n):=1;

L60PP.2
          S<A(x):=S>B(c)/S>C(t)->B(*);

L60PP.3
          S<A(s):=S>B(c)/S>C(t)->B(*);

L60PP.4
          S<A(a):=S>B(c)/S>C(t)->B(*);
          $FACTOR = 5
L100.1

          IF S>B(a)->A(.abs)/.004->B(>)
            THEN
              X
            ELSE
              GOTO L200.1 FI
          ;
```

```
L100.2
          RHLP_101:=S>B(a)->A(-)/S>C(x)->B(*)/S>C(x)->B_A(*);

L100.3
          RHLP_102:=2/S>C(n)->B(*)/1->B_A(+);

L100.4
          RHLP_102:=2/S>C(n)->B(*)/RHLP_102->B_A(*);

L100.5
          S<A(a):=RHLP_101/RHLP_102->B(/)=V1,
          RHLP_101:=V1;

L100.6
          S<A(s):=S>B(s)/RHLP_101->B(+);

L100.7
          S<A(n):=S>B(n)/1->B(+),
          GOTO L100.1;
          $FACTOR = 1
L200.1
          X;


END
```

EXERCISE 7
Verify that the GEN file still is a correct program for calculating sin (c*t). Write a BASIC program that is a one-to-one translation of it, and add the Input/Output features from the original BASIC program. Run on your machine and compare the results.

XIV. CONTROL

HOW THE CONTROLLER HARDWARE IS GENERATED

Until now we have ignored one important question: How does the controller for this hardware work? We have seen several interfaces to it already: the labels, the IF THEN, the GOTO and the instruction vector I, but no clear explanation of what it does.

To understand the controller, let us assume a simple implementation of it. Let the microprogram be stored in a microprogram memory SM, with an output port >A. In the simplest case we could replace all occurrences of I by SM>A. The current address of SM>A would be the label.

```
        S<A(x):=S>B(c)/S>C(t)->B(*),
        RP:=RP->A(.increment);
L60PP.3
        S<A(s):=S>B(c)/S>C(t)->B(*),
        RP:=RP->A(.increment);
L60PP.4
        S<A(a):=S>B(c)/S>C(t)->B(*),
        RP:=RP->A(.increment);
        $FACTOR = 5
L100.1
        IF S>B(a)->A(.abs)/.004->B(>)
          THEN
            RP:=RP->A(.increment)
          ELSE
            RP:=L200.1
            FI
        ;
L100.2
        RHLP_101:=S>B(a)->A(-)/S>C(x)->B(*)/S>C(x)->B_A(*),
        RP:=RP->A_A(.increment);
L100.3
        RHLP_102:=2/S>C(n)->B(*)/1->B_A(+),
        RP:=RP->A(.increment);
L100.4
        RHLP_102:=2/S>C(n)->B(*)/RHLP_102->B_A(*),
        RP:=RP->A(.increment);
L100.5
        S<A(a):=RHLP_101/RHLP_102->B(/)=V1,
        RHLP_101:=V1,
        RP:=RP->A(.increment);
L100.6
        S<A(s):=S>B(s)/RHLP_101->B(+),
        RP:=RP->A(.increment);
L100.7
        S<A(n):=S>B(n)/1->B(+),
        RP:=L100.1
        ;
        $FACTOR = 1
L200.1
        X,
        RP:=RP->A(.increment);
END
```

SECOND PASS THROUGH MIMB

If you look closely at the INP file shown (prp.inp), you see that it is already fully bound. That means that all functions have hardware resources bound (allocated) to it. The reason for this is that in order to enter all the RP := RP -> A (.increment) we replaced all ";"s with this expression followed by an ";" with one command of the editor, and then deleted the expressions where RP was already assigned. This did not result in a nice printing format, so we ran it through MIMB once to get a formatted GEN file, which we used here. The ADDMODULE section was added after formatting.

In the header of INP we notice a command

```
                $OPTION 7
```

This directs MIMB to create enable or control type connections to storage devices. They are marked by the attribute

```
                .CON
```

In order to declare the width of the CON vector, we added declarations in the $ADDMODULE section. For example:

```
          S<.CON.BIT(0:0)
```

tells us that the control (enable) input for input ports (<) of S is one bit wide.

We also declared RP with 10 bits for 1K microprogram memory, and RHLP with two duplicates: RLHP_101 and RHLP_102. If we do not declare enough duplicates we could get a warning in the OTB file:

```
  46  L100.2.1
  47                RHLP_101:=
                            ^
+++++ ERROR AT L100  41 MODULE NOT DECLARED  PARAMETER: RHLP_101
```

In the next section we will talk more about error messages.

ANOTHER LOOK AT THE SCR FILE

With all of these details, the SCR file illustrating the structure is now getting quite long, so we put it in Appendix C, file PRP.SCR. This is also a good reason for talking about the control section so late in the design. It does not add much to the cost or performance. Therefore, in initial phases of rough cost performance estimates, we simply neglect the details, but insert them as we approach the final solution. Thus, we save designer time for all the detailed declarations and also computer time to deal with the details.

Let us now look at the end of SCR.

```
*LIST /GEZ/PRP.SCR

RHLP_101<.CON.BIT(0:0)   <-   I>.CRHLP_101,
RHLP_102<.DAT.BIT(15:0)   <-   B_A>.BIT(15:0),
RHLP_102<.CON.BIT(0:0)   <-   I>.CRHLP_102,
RP<.DAT.BIT(9:0)   <-   "?UDRP>.BIT(9:0),?"
    "?UDRP<a.DAT.BIT(9:0)    <-  ?"A>.BIT(9:0)/
    "?UDRP<b.DAT.BIT(9:0)     <-  ?"A_A>.BIT(9:0)/
    "?UDRP<c.DAT.BIT(9:0)     <-  ?"I>.DRP
    "?,UDRP>.MPX.BIT(1:0)     <-   I>.MDRP?",
RP<.CON.BIT(0:0)   <-   I>.CRP,
S<A.DAT.BIT(15:0)   <-   "?UDSA>.BIT(15:0),?"
    "?UDSA<a.DAT.BIT(15:0)    <-  ?"I>.DSA/
    "?UDSA<b.DAT.BIT(15:0)    <-  ?"B>.BIT(15:0)
    "?,UDSA>.MPX.BIT(0:0)     <-   I>.MDSA?",
S<A.ADR.BIT(12:0)   <-   I>.ASA,
S<A.CON.BIT(0:0)   <-   I>.CSA,
S>B.ADR.BIT(12:0)   <-   I>.ASB,
S>C.ADR.BIT(12:0)   <-   I>.ASC;
```

All registers and the memory input port S<A now have their .CON input connected to fields of the microinstruction vector. Also, the data input of the program counter RP has a multiplexer with three sources.

What we do not see is how the condition in L100.1 controls the RP mux to either increment or jump to L200.1. The hardware structure does not show that this is achieved by another multiplexer for the address input UDRP>.MPX. If we look at the I-field declaration in SCR:

```
I>.BIT(96:94)MDBb,
I>.BIT(98:97)MDRP,
I>.BIT(100:99)MDRP@A,
I>.BIT(103:101)FB_A,
```

we see two multiplexer address fields: MDRP and MDRP@A. In the case of THEN, the first is selected to address the mux UDRP, the second in the case of ELSE (the @ indicates additional fields in cases where more than one is necessary for the same destination). The selection is controlled by the conditional value, in our example by the compare output of B(>).

Besides this one little piece of control logic, and the actual declaration of the microprogram memory, the controller is now complete.

The microprogram memory is difficult to declare in MIMB. The easiest way is to use an editor and replace all "I."s by "SM>A(R)." and add in the ADDCONNECT section the connection:

```
         SM>A.ADR <- RP
```

The OTB file does not significantly differ from the last one, and we include a complete copy of it in Appendix C.

XV. ERRORS

WHAT HAPPENS IF WE MAKE AN ERROR?

The beginner especially will make many errors, but even the experienced MIMOLA user will make some, because his designs become more complex as he progresses. Therefore, we will see how MIMOLA can help in this important problem.

We can distinguish between syntactic and semantic errors. Syntactic errors are deviations from the language definition of MIMOLA, and are always detected by the syntax analyzer of the MIMOLA system and reported to the user. This does not mean that the user can always tell what he did wrong. This depends on the quality of the reported error message.

Semantic errors can only be partially detected automatically. In many cases it is up to the designer to find these errors.

A semantic error that can be detected is, for example, a double assignment to a storage device in one instruction:

```
          R1 := 1,  R1 := 2;
```

Since both are executed in parallel, we do not know if the resultant value in R1 is 1 or 2. MIMOLA will report this as "conflict at destination".

Other semantic errors, such as an incorrect algorithm, can be found most of the time by simulation. The MSS simulator is explained in /11/.

Errors in hardware declaration are normally caught by careful examination of the SCR or OTB files. A close look at the GEN file and the allocated hardware is also very helpful.

This examination of the outputs is also important not only for finding errors, but also for discovering improper ways of describing an algorithm. Such "mistakes" sometimes cause unnecessary hardware to be generated. This is often not the user's fault, but a limitation of the MIMOLA System.

HOW TO MAKE AND FIND SYNTACTIC ERRORS

We probably do not need to help anyone make errors. However, for the purpose of learning about the responses, we will lead you through some examples.

If you tried some of the examples in the text and something went wrong, you should have been able to find the problem by comparing your work with the printed versions in this text. We will use the same technique now by inserting errors into the following examples.

```
*LIST /GEZ/PRQ.T

$ADDMODULE
S(8191:0).BIT(15:0)WORD;
$PROGRAM
BEGIN
L60  S(n) := 1;
END
```

```
*LIST /GEZ/PRQ.INP

$ADDMODULE
S(8191:0).BIT(15:0)WORD;
$PROGRAM
BEGIN
L60  S(n := 1;
END
```

By now our trained eye immediately detects the error in the second version. Will MIMB find it? Where does it show up?

If we are lucky and run MIMB interactively, the error will appear on the terminal and in the OTB file. If we run in the Batch mode, errors are reported only in the OTB file. This file provides much information about the error, so let's look at it.

```
*LIST /GEZ/PRQ.OTB

MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
   5   BEGIN
   6   L60  S(n :
              ^
+++++ ERROR AT L60, 4 NO MATCHING RULE OF SYNTAX PARAMETER: 113
+++++ ':' NOT EXPECTED
     SYMBOL                   DATA         PORT        HI    LO
   "left"(bottom of stack)                             -1    0
   <program head>                                      -1    0
   <label>                   L60                       -1    0
   S                         S                         -1    0
   (                                                   -1    0
-> "low letter or id strng"            n               -1    0
   :                                   :               -1    0
   "right"(top of stack)                               -1    0
REPAIRED STACK
     SYMBOL                   DATA         PORT        HI    LO
   "left"(bottom of stack)                             -1    0
```

```
-> <program head>                                         -1     0
   END                                        -1     0
   "right"(top of stack)                                  -1     0
***     0 ESBS READ    (          0 WEIGHTED)
***     0 ESBS CREATED (          0 WEIGHTED)
***     0 ESBS TOTAL   (          0 WEIGHTED)
        1 ERROR(S) DURING COMPILATION
```

The first line specifies the part of the MIMOLA Software System that was used (in this case MIMB - Part B) and version information. Since the MSS is still being improved, it is important to notice which version was used, because results can change slightly in format and contents.

The second line (CHARACTERSTRING ACCEPTED) tells us that the first section, here the ADDMODULE section, was accepted as syntactically correct.

The forth line is the line in which an error was detected. Line numbers and the preceding line are always shown for orientation in the program. The "^" points to the location in the string above which the error is suspected. In our case this is as close as it can get. In more complicated cases the "^" may point to where the error was detected, not where it occured.

The next line gives us the label of the faulty instruction and the reason for the error. In the case of syntactic errors, it is nearly always error number 4 or "no matching rule of syntax". The parameter number comes from the "chemistry" of syntax analysis. Normally we can skip this information, but for the curious: 113 stands for the nonterminal symbol <operand> in the BNF rules. Appendix F of the MIMOLA Report /3/ has the conversion table.

The message "':' not expected" tells us what we need to know. What was expected was either a continuation of the expression for the address (operand) or a ")".

In this case, even without the correct program for comparison, we would easily have found the error.

However, let us look at the additional output. This is still related to the same error and gives us the state of the syntax analysis stack at the moment of the detection of the error. Do not panic now, even if you are not a syntax analysis specialist. Some of the information can be easily understood.

```
*LIST /GEZ/PRQ.T1

     SYMBOL                   DATA          PORT       HI     LO
   "left"(bottom of stack)                             -1     0
   <program head>                                      -1     0
   <label>                    L60                      -1     0
   S                          S                        -1     0
   (                                                   -1     0
-> "low letter or id strng"              n             -1     0
   :                                     :             -1     0
   "right"(top of stack)                               -1     0
```

The stack stands on its head and its current ends are marked by the tokens: left, right. Therefore, the ":" was on top of the stack when the MSS detected that there was no way to apply a syntax rule. The pointer -> points to the entry before that, saying that something like an identifier had been found and that this was "n". The " " tells us that some preprocessing with the string has been done, for example, that a "name" or an "attribute identifier" was found.

Going two steps down the stack we now have

```
          S("Low letter or id string":
```

We also know from the message

```
          parameter:  113
```

that "Low letter or id string" was reduced to symbol 113 or <operand>. Now the only rules with "S" that apply in the PROGRAM section are:

```
          S ( <operand> )
          S"name" ( <operand> )
```

(If you want to check this, the rules are listed in Appendix B of the Report /3/.)

Since the syntax analyzer looks ahead one symbol (it is a LR(1) grammer), it sees that the ":" is not possible in this situation. Therefore, it reports the error.

Otherwise the stack looks fine. It starts with <program head> and <label>, which is a correct situation, and there are rules waiting to proceed.

Most of you will be confused by now and we can only tell you that this is normal. Look at the information this way: try to get as much out of it as you can to find the error. With more experience this will make it easier to find errors. Otherwise, you have to look more closely at your file and compare it to MIMOLA syntax.

REPAIRING THE STACK

One of the problems of syntax errors is that the analyzer cannot continue. The current stack situation would lead to error messages for any further symbol. Therefore, the program tries to repair the stack, and that is what it says in the OTB file. It does so by skipping around the error. In our case it skips instruction L60 and goes on to "END". This is now a correct end of the program and we get the usual output information. This is correct, in addition to the fact that erroneous instructions have been skipped.

If the MSS cannot recover, it stops reading the input and gives the message:

"STOP FOR ERRORS"

```
*LIST /GEZ/PRR.OTB

MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80
   0   $ADDMOGULE
        ^
+++++ ERROR AT 0   31 INVALID MONITOR KEYWORD  PARAMETER: ADDMOGULE
***** STOP FOR ERRORS,   ERNR= 31 ERCOUNT=   1
```

The message should be clear enough. The keyword was misspelled.

Sometimes the analyzer goes a long way before it detects the error, as in the following example.

```
*LIST /GEZ/PRS.INP

$ADDMODULE
S(8191:0).BIT(15:0)WORD.MOREPORT;
$PROGRAM
BEGIN
L60  S(n) := 1,
L70  S(x) := S(c) / S(t) -> B(*);
L80  S(S) := S(x);
L90  S(a) := S(x);
END
```

There are two errors, but when first looking at the error message we are confused (at least the author was when he first saw it).

```
*LIST /GEZ/PRS.OTB

MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
   6   L70  S(x) := S(c) / S(t) -> B(*);
   7   L80  S
```

```
+++++ ERROR AT L60,  4 NO MATCHING RULE OF SYNTAX PARAMETER: 44
+++++ ';' NOT EXPECTED ,',' (CODE=44) EXPECTED
     SYMBOL                      DATA         PORT         HI     LO
   "left"(bottom of stack)                                 -1     0
   <program head>                                          -1     0
   <label>                       L60                       -1     0
   <statement block>             S            B            -1     0
   ,                                                       -1     0
   <label>                       L70                       -1     0
   <statement block>             S            B            -1     0
-> ;                                                       -1     0
   L                             L80                       -1     0
   "name"                        L80                       -1     0
   S                             S                         -1     0
   "right"(top of stack)                                   -1     0
REPAIRED STACK
     SYMBOL                      DATA         PORT         HI     LO
   "left"(bottom of stack)                                 -1     0
-> <program head>                                          -1     0
   L                             L80                       -1     0
   "name"                        L80                       -1     0
   S                             S                         -1     0
   (                                                       -1     0
   "right"(top of stack)                                   -1     0
   6    L70  S(x) := S(c) / S(t) -> B(*);
   7    L80  S(S) :
               ^
+++++ ERROR AT L80,  4 NO MATCHING RULE OF SYNTAX PARAMETER: 40
+++++ ')' NOT EXPECTED ,'(' (CODE=40) EXPECTED
     SYMBOL                      DATA         PORT         HI     LO
   "left"(bottom of stack)                                 -1     0
   <program head>                                          -1     0
   <label>                       L80                       -1     0
   S                             S                         -1     0
   (                                                       -1     0
   S                             S                         -1     0
-> )                                                       -1     0
   :                                          :            -1     0
   "right"(top of stack)                                   -1     0
REPAIRED STACK
     SYMBOL                      DATA         PORT         HI     LO
   "left"(bottom of stack)                                 -1     0
-> <program head>                                          -1     0
   L                             L90                       -1     0
   "name"                        L90                       -1     0
   S                             S                         -1     0
   "right"(top of stack)                                   -1     0
+++++ ERROR AT L90, 54 UNDEFINED LABEL:                    PARAMETER: L70
***     1 ESBS READ     (          1 WEIGHTED)
***     0 ESBS CREATED  (          0 WEIGHTED)
***     1 ESBS TOTAL    (          1 WEIGHTED)
        3 ERROR(S) DURING COMPILATION
```

The "^" points at L80, but the message right below says "Error at L60". Which should we believe? Then it says "';' not expected, ',' expected".

Let us first look at L60 in the INP file. The error is that an ',' was used instead of an ';'. After it saw the ',' it tried to find more parallel executable statements. The construct <label> can be used in many different contexts. Since it can appear within a statement, no error was flagged after the ','. The error became obvious when the ';' was found at the end of instruction L70. The syntax analyzer, however, correctly assumed it was still in instruction L60 and reported the error correctly. We included this example so the reader will not be discouraged if he encounters a similar problem.

The system then recovers to find another error in L80. The second "S" is not an identifier, so it can only be a nested reference to another memory location in S. The expected character is a "(" for the address of this memory, and this is stated in the error message. The system thus proposes a solution. In our case, the system could not know that the "S" should be an "s", but it clearly points to the correct location. The system immediately recovers from this error and finally finds one correct instruction, L90. The report of the error at L90: "Undefined Label" results from the above confusion and is only a warning that can be skipped for now. This is a typical semantic error message.

If we look at the GEN file we see that L90 is the only instruction that made it correctly through the system. All others have been skipped.

```
*LIST /GEZ/PRS.GEN

"MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80"
     $PASS   1
     $PROGRAM


BEGIN


L90.1
            S<B(a):=S>A(x);


END
```

SEMANTIC ERRORS

Semantic errors change the meaning of the input. Since there are various error messages to point out what could be caught or looks like an error, the designer has to decide which ones apply to his situation. Appendix G in the Report /3/ explains the messages.

Some messages refer to limitations in the version of the Software System. The user can only try to understand what might have caused the error and work around the problem. With future revisions of the MSS, these limitations will be reduced.

XVI. CONCLUSION

We hope that this PRIMER has been helpful to the reader as an introduction to the MIMOLA Design Methodology, language and Software System. It was not intended to be complete or to be a reference manual. It only covers part of the available software - MIMB. At this point the reader should be able to make extended use of the capabilities by reading the listed documentation. Also, there is no substitute for hands-on experience, and we again encourage the use of the software.

We want to express our thanks at this point for the excellent editing and text processing that Beth Hurd did on the PRIMER.

/1/ Zimmermann, G. "Eine Methode zum Entwurf von Digitalrechnern mit der Programmiersprache MIMOLA," (A Method for the Design of Computers with the Programming Language MIMOLA), GI-6. Jahrestagung Stuttgart 1976.

/2/ Zimmermann, G. "Report on the Computer Architecture Design Language MIMOLA, Machine Independent Microprogramming Language," Bericht Nr. 4/77 des Institute fuer Informatik und Praktische Mathematik, Kiel 1977.

/3/ Zimmermann, G., and P. Marwedel. "MIMOLA Report Rev. 1 and the MIMOLA Software System User's Manual," Christian-Albrechts-Universitat, Kiel, Bericht Nr. 2/79, May 1979.

/4/ Zimmermann, G. "The MIMOLA Design System, A Computer Aided Processor Design Method," _16th Design Automation Conf. Proc.,_ San Diego 1979, pp. 53-58.

/5/ Zimmermann, G. "MDS - The MIMOLA Design Method," invited paper, _Journal of Digital Systems,_ Vol. 4, No. 3, 1980, pp. 337-369.

/6/ Zimmermann, G. "VLSI Design with the MIMOLA Design System," _Proc. IEE European Conf. on Electronic Design Automation,_ Brighton, UK, 1981, pp. 277-280.

/7/ Zimmermann, G. "Computer Aided Synthesis of Digital Systems," _Proc. 5th Intl. IFIP Conf. on Comp. Hardware Description Languages and their Application,_ Kaiserslautern, 1981.

/8/ Rahf, D., Zimmermann, G., and R. Cloutier. "MIMD User's Manual," (translated from Master's thesis entitled "Ein Codegenerator and Speicherzuordnungs- und Parallelisierungs-strategien fuer MIMOLA-Programme," University of Kiel, December 1981). CCSC Document, October 1982.

/9/ Cloutier, R., and R. Rudell. "MIMOLA to DL Translator - User's Manual," CCSC Document, October 1982.

/10/ Cloutier, R., and R. Rudell. "MIMOLA to DL Translator - Computer Operation Manual," CCSC Document, December 1982.

/11/ Albert, M. "Simulator User's Manual," CCSC Document, (to be published).

"WHAT IS MIMOLA?"

R. Cloutier and G. Zimmermann

Many people have heard about "MIMOLA", but few understand what it really is. Documentation is available, courses have been taught, and numerous presentations have been given, but still MIMOLA is nebulous. Why is it so difficult to understand? We believe it is because MIMOLA is new to most people and new concepts are difficult to comprehend. A new programming language is relatively easy to explain to experienced programmers, by relating it to features found in existing languages. A new hardware description language is also easy to understand if the constructs it describes are already familiar to the hardware designer. If however, the language provides features that do not directly relate to implementable hardware, these features may be difficult to imagine.

MIMOLA might be difficult to understand because the new features it provides are unexpected. However, MIMOLA is based on some common concepts that can be related to existing tools. For example, MIMOLA is a hardware description language (HDL). It also includes a simulator that can operate with the design description. Unfortunately, this is where the similarity with existing tools ends. MIMOLA may also be used to express algorithms and may be considered a programming language. This combination of an HDL and programming features is difficult to visualize. MIMOLA also includes a software package that can do hardware synthesis, design analysis, and microcode generation. Each of these may generally be comprehended by relating them to what one already knows, but the terms are sufficiently vague that they are of little value when describing the function of the tool. Because of this lack of clarity, it is difficult to describe what MIMOLA is needed for, but if this need was understood, perhaps the view of MIMOLA itself would become clear. With this in mind, we will first explain the problem that MIMOLA can be used to solve, before explaining MIMOLA itself.

NEED FOR DESIGN AUTOMATION

While digital system design is still an art, it does involve some tasks that are well understood. If these tasks were to be automated, the productivity of the designer would be vastly increased. Design automation may also be applied to the resolution of problems caused by new technologies and new requirements.

Examples of productivity improvements by DA include:

- o Pre-fabrication error detection by simulation, timing analysis or design rule checking

- o Schematic entry and editing of systems for documentation

- o Automatic printed circuit board layout

Examples of difficulties and new requirements imposed by technology:

- o VLSI layout complexity

- o GaAs signal synchronization

- o Distributed systems reliability requirements

The productivity examples are related in that they are all well understood and do not require complete designer attention. The second set of examples are related in that they must be resolved during the planning phase. MIMOLA was designed to aid planning and to increase productivity.

THE IMPORTANCE OF PLANNING

Three activities are strongly related in designing a new product: planning, implementation and analysis. Planning requires experience, creativity and design decisions, and is based on assumptions about the implementation. The implementation is the plan translated into reality, and should be more or less automatic, or at least predictable. Analysis is used to evaluate the plan and the implementation and to compare the results with the requirements.

It should then be obvious that planning determines the quality of the product and assures that the requirements are met and the plan is implementable. Nevertheless, most tools for integrated circuit design support only the implementation and analysis. Planning is left to the designer. Examples are automatic layout, design rule checkers and test generators.

Because of this lack of planning tools, the implementation stage is often started before planning is completed so that assumptions made during the planning stage can be checked against the actual results. This is a rather expensive method of checking assumptions. The expense of a reimplementation dictates that whatever assumptions were made first become the standard, and alternative choices are not considered. In such a top-down design path, optimal designs are a rarity. Every decision must be correct the first time, because it is difficult and expensive to go back and modify a design once it has been implemented. It is unreasonable to expect that a correct and optimal design for a large digital system could be achieved in a single design cycle; the complexity is too great.

FRONT-END DESIGN TOOLS

Tools for planning (or front-end design tools as they are sometimes called) may operate at several levels. The basic level would be planning data entry, editing and documentation aids. The addition of plan analysis and evaluation tools that examine the design and access the planning data would be the next level. Automatic model generation and then fully automatic synthesis of plans would be the ultimate goal, but it is currently far off. In fact, very few tools currently exist that even address the planning problem.

Chip planning is an excellent example of a front-end tool. It is also one of the central issues in VLSI design, because of the difficulty in arranging 100,000 elements optimally and economically on a chip surface. Chip planning is an approach that partitions the problem into many smaller designs (cells) of manageable complexity. The planning tasks then involve the organization of these cells. Currently this task is performed manually, but work is being done to aid in the manipulation of cells, and the estimation of cell size.

Now let us step back to the architectural and logic design levels. Decisions made here are even more important to the final product than those to be made during chip planning. An aid to planning at this level is the MIMOLA Software System (MSS). Although it is still used mostly for research, the MSS may also be used for architectural planning and evaluation.

THE MIMOLA SOFTWARE SYSTEM

Assume, for example, we want to design a very fast processor for image processing. We have a basic knowledge of the algorithms and know how much time it takes to process one picture. The goal is to develop the minimum number of VLSI chips needed to do the processing. This is certainly not a basic logic design problem, because any general purpose computer could be programmed to execute the algorithm. However, it would be far too slow and too expensive to have enough processors in parallel. Therefore, we are dealing with the optimization problem of finding a system with the least cost for the required performance. This problem can be solved with proper planning.

Now assume several designers get together and brainstorm about parallel and pipelined architectures. Processor structures are drawn on the back of envelopes, but how do they decide which design has the lowest cost and yet fulfills the requirements? Build prototypes? - Too expensive! Simulate? - A lot of work! Guess? - Possible, but only if the designers are very experienced. Use MIMOLA? - But what can it do for the designer?!

The MIMOLA Methodology structures the design process. The first step is to specify the behavior. In our example, the algorithms for image processing are the desired behavior, so we have to express them in the programming language MIMOLA. Now we make certain that the MIMOLA programs execute correctly. This is accomplished by using the MIMOLA simulator. Only the behavioral description is simulated, since nothing has been specified in the hardware area. For example, the behavioral description of an instruction set processor is the collection of behavioral descriptions of each instruction, not the algorithm that would be executed on the processor.

The next step is to compare different architectures. This requires that the algorithm be totally rewritten, yet still describe the same function. As an example, a sorting algorithm could be described as a bubble sort or as a merge sort; both would produce the same sorted results. However, their approach, speed, and required hardware could be very different. The MSS could be used to compare the architectures by calculating the dynamic performance and estimated maximum cost. The designer interaction is very important at this stage of the design. Both the creation of algorithms and the interpretation of the results require the designer; they cannot be automated.

Once a decision on the system architecture is made, the task of designing the processor is again supported by MIMOLA. MIMOLA is able to generate several processor structures that all have the desired behavior. Differences are in the amount of parallel hardware used, which is controlled by the designer. He can also propose and enter structures that reflect his own ideas and let the system analyze them. The output of real interest is the dynamic performance, the cost estimate and the the dynamic utilization of resources. The utilization figures are very important for deciding whether to add or remove hardware resources. The MIMOLA system optimizes the utilization to achieve a good cost - performance tradeoff. Thus, the designer is able to reach an acceptable processor structure.

After a structure has been defined, more checking is needed. First we make sure that the generated microprogram for this structure is still behaviorally correct, again using the MIMOLA simulator. Then we check that the processor structure can execute the microprograms correctly. For this purpose, there is a MIMOLA to DL translator that generates a complete description of the structure in Honeywell's Hardware Description Language, DL. The MIMOLA microcode generator creates a list of zeros and ones representing the microcode. This code, together with the DL description, is fed into the DL simulator to simulate the design.

Simultaneously, we enter this design into the chip planner. This provides more reliable figures regarding the necessary chip area than earlier estimates. In fact, we should already be using the chip planner for selected trial structures during the design, to make certain that we can fulfill the requirements. Chip planning also provides us with wire length calculations to improve our delay estimates. Thus, we get closer to the real performance.

Chip planning may show us that we still haven't reached our goal and must go back a few steps. The advantage of the MIMOLA system is that these iterations do not take much time. In the future, the designer will be able to do all of these activities at his workstation and let the computer do the work for him. This gives the designer more time for creating new architectures and hardware configurations, and the system helps him to evaluate his ideas. Documentation will be automatic, because the MIMOLA and DL descriptions can be used for this purpose.

CONCLUSION

It is our hope that the reader now has a better understanding of what MIMOLA is and what front-end tools can accomplish for the designer. Much work still needs to be done, but designers should be able to take advantage of what is already available. Therefore, we strongly encourage the use of the MIMOLA system and will give every possible support. For more information contact Rich Cloutier or Gerhard Zimmermann (HVN 554-4077 and 554-4067, respectively).

ANSWERS TO EXAMPLES

Example 1

```
 ^ZDA>UDD>DA>ALBERT>MAN>PRIM.B

   list

   10  print 'This Program calculates sin(c*t)'
   20  print ' c = ';
   30  input c
   40  print ' t = ';
   50  input t
   60  n = 1
   70  x = c * t
   80  s = x
   90  a = x
   100 if abs(a) < .004 then goto 200
   110 a = -a * x * x / (2 * n * ( 2 * n + 1 ) )
   120 s = s + a
   130 print 'n = ';n,'    a = ';a, '    s = ';s
   140 n = n + 1
   150 goto 100
   200 print 'sin( ';x;' ) = ';s
   210 print ' type: 0 for quit, 1 for new c and 2 for new t'
   220 input i
   230 if i = 1 then goto 20
   240 if i = 2 then goto 40
   250 end

   run

 BASIC- 3.0-08/02/0700  1982/12/08 0932:26               PAGE 1
 ^ZDA>UDD>DA>ALBERT>MAN>PRIM.B


 This Program calculates sin(c*t) @ c = ? 1 @ t = ? 1
 n =  1              a = -.166667                s =  .833333
 n =  2              a =  .833333E-2             s =  .841667
 n =  3              a = -.198413E-3             s =  .841468
 sin(  1  ) =  .841468
  type: 0 for quit, 1 for new c and 2 for new t @? 2 @ t = ? 2
 n =  1              a = -1.33333                s =  .666667
 n =  2              a =  .266667                s =  .933334
 n =  3              a = -.0253968               s =  .907937
 n =  4              a =  .141093E-2             s =  .909348
 sin(  2  ) =  .909348
```

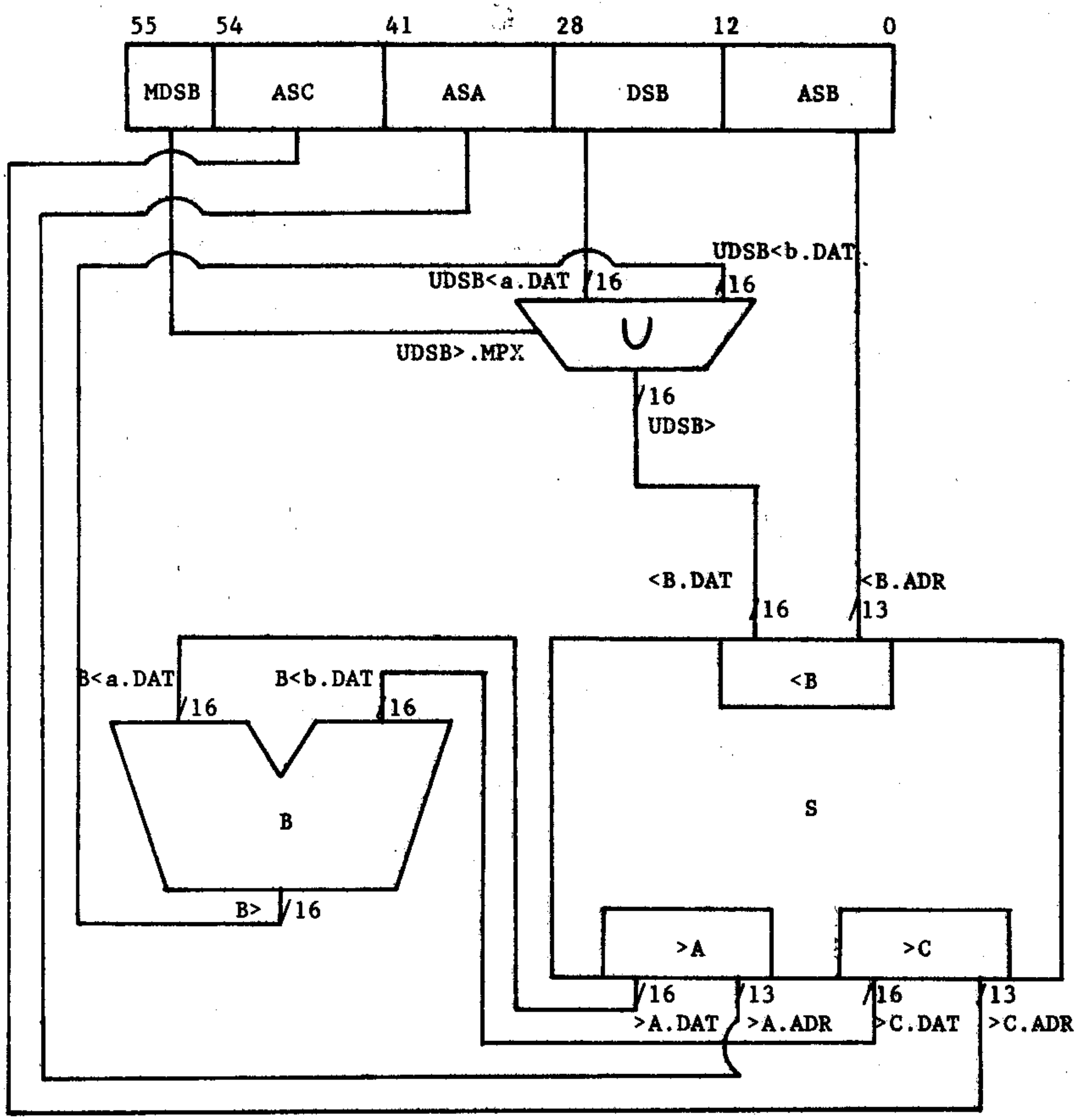
```
  type: 0 for quit, 1 for new c and 2 for new t @? 2 @ t = ? 4
 n =  1               a = -10.6667                  s = -6.66667
 n =  2               a =  8.53333                  s =  1.86667
 n =  3               a = -3.2508                   s = -1.38413
 n =  4               a =  .722399                  s = -.661729
 n =  5               a = -.105076                  s = -.766805
 n =  6               a =  .010777                  s = -.756028
 n =  7               a = -.821108E-3               s = -.756849
 sin(  4  ) = -.756849
  type: 0 for quit, 1 for new c and 2 for new t @? 2 @ t = ? 8
 n =  1               a = -85.3333                  s = -77.3333
 n =  2               a =  273.067                  s =  195.734
 n =  3               a = -416.102                  s = -220.368
 n =  4               a =  369.868                  s =  149.5
 n =  5               a = -215.196                  s = -65.6962
 n =  6               a =  88.2856                  s =  22.5894
 n =  7               a = -26.9061                  s = -4.31673
 n =  8               a =  6.33085                  s =  2.01412
 n =  9               a = -1.18472                  s = ~r
.8294
 n =  10              a =  .180529                  s =  1.00993
 n =  11              a = -.0228337                 s =  .987095
 n =  12              a =  .243559E-2               s =  .989531
 sin(  8  ) =  .989531
  type: 0 for quit, 1 for new c and 2 for new t @? 0
 END AT 250
```

Example 2

These are some possible answers, other correct solutions exist.

```
  S(a)/S(b) -> B(+)
  2/S(c)/5 -> B1(+)/5 -> B2(*) -> B3(+)
  3/4 -> Ba(+)/1/2 -> Bb(+) -> Bc(*)/3/S(a)/S(b)
                             -> B1(+) -> B2(*) -> B3(+)
```

Example 3


55
54
41
28
12
0
MDSB
ASC
ASA
DSB
ASB
UDSB<b.DAT
UDSB<a.DAT
16
16
U
UDSB>.MPX
16
UDSB>
<B.DAT
16
<B.ADR
13
<B
S
B<a.DAT
16
B<b.DAT
16
B
B>
16
>A
>C
16
13
16
13
>A.DAT
>A.ADR
>C.DAT
>C.ADR

Example 4

```
in_line
```

Example 5

```
a+b        --->       S(a)/S(b) -> B(+)
(a+b)*c        --->       S(a)/S(b) -> B(+)/S(c) -> B(*)
(a+b)*c+d        --->       S(a)/S(b) -> B(+)/S(c) -> B(*)/S(d) -> B(+)
```

Example 6

```
-a    --->    S(a) -> A(-)
-a+b    --->    S(a) -> A(-)/S(b) -> B(+)
abs(a+b)    --->    S(a)/S(b) -> B(+) -> A(.ABS)
-abs(a+b)    --->    S(a)/S(b) -> B(+) -> A(.ABS) -> A(-)
(-abs(a+b))*c    --->    S(a)/S(b) -> B(+) -> A(.ABS)
                                    -> A(-)/S(c) ->B(*)
```

Example 7

```
UDD>DA>ALBERT>MAN>PRIMER.ANS7

list

10   print 'This Program calculates sin(c*t)'
20   print ' c = ';
30   input c
40   print ' t = ';
50   input t
61   n = 1
62   x = c * t
63   s = x
64   a = x
110 if abs(a) < .004 then goto 200
120 r1 = -a * x * x
130 r2 = 2 * n +1
140 r2 = 2 * n * r2
150 a = r1 / r2
155 r1 = a
160 s = s + r1
165 print 'n = ';n,'     a = ';a, '     s = ';s
170 n = n + 1
175 goto 110
200 print 'sin( ';x;' ) = ';s
210 print ' type: 0 for quit, 1 for new c and 2 for new t'
220 input i
230 if i = 1 then goto 20
240 if i = 2 then goto 40
250 end
```

```
run

This Program calculates sin(c*t)
@ c = ? 1
@ t = ? 2
n =  1              a = -1.33333                   s =  .666667
n =  2              a =  .266667                   s =  .933334
n =  3              a = -.0253968                  s =  .907937
n =  4              a =  .141093E-2                s =  .909348
sin(  2  ) =  .909348
 type: 0 for quit, 1 for new c and 2 for new t
@? 2
@ t = ? 4
n =  1              a = -10.6667                   s = -6.66667
n =  2              a =  8.53333                   s =  1.86667
n =  3              a = -3.2508                    s = -1.38413
n =  4              a =  .722399                   s = -.661729
n =  5              a = -.105076                   s = -.766805
n =  6              a =  .010777                   s = -.756028
n =  7              a = -.821108E-3                s = -.756849
sin(  4  ) = -.756849
 type: 0 for quit, 1 for new c and 2 for new t
@? 2
@ t = ? 8
n =  1              a = -85.3333                   s = -77.3333
n =  2              a =  273.067                   s =  195.734
n =  3              a = -416.102                   s = -220.368
n =  4              a =  369.868                   s =  149.5
n =  5              a = -215.196                   s = -65.6962
n =  6              a =  88.2856                   s =  22.5894
n =  7              a = -26.9061                   s = -4.31673
n =  8              a =  6.33085                   s =  2.01412
n =  9              a = -1.18472                   s =  .8294
n =  10             a =  .180529                   s =  1.00993
n =  11             a = -.0228337                  s =  .987095
n =  12             a =  .243559E-2                s =  .989531
sin(  8  ) =  .989531
 type: 0 for quit, 1 for new c and 2 for new t
? 0
```

```
*LIST /GEZ/PRJ.SCR

$ADDMODULE
A>(.abs[1,0]),
A>.BIT(15:0),
A<.DAT.BIT(15:0),
A_A>(-[1,0]),
A_A>.BIT(15:0),
A_A<.DAT.BIT(15:0),
B>(>=[1,0],*[1,1]),
B>.BIT(15:0),
B>.FCT.BIT(0:0),
B<a.DAT.BIT(15:0)
    "?,UDBa<a.DAT.BIT(15:0),
    UDBa<b.DAT.BIT(15:0),
    UDBa>.MPX.BIT(0:0),
    UDBa>.BIT(15:0)?",
B<b.DAT.BIT(15:0)
    "?,UDBb<a.DAT.BIT(15:0),
    UDBb<b.DAT.BIT(15:0),
    UDBb>.MPX.BIT(0:0),
    UDBb>.BIT(15:0)?",
B_A>(*[1,0]),
B_A>.BIT(15:0),
B_A<a.DAT.BIT(15:0),
B_A<b.DAT.BIT(15:0),
B_B>(*[1,0]),
B_B>.BIT(15:0),
B_B<a.DAT.BIT(15:0),
B_B<b.DAT.BIT(15:0),
B_C>(*[1,0]),
B_C>.BIT(15:0),
B_C<a.DAT.BIT(15:0),
B_C<b.DAT.BIT(15:0),
B_D>(+[1,0]),
B_D>.BIT(15:0),
B_D<a.DAT.BIT(15:0),
B_D<b.DAT.BIT(15:0),
B_E>(*[1,0]),
B_E>.BIT(15:0),
B_E<a.DAT.BIT(15:0),
B_E<b.DAT.BIT(15:0),
B_F>(/[1,0]),
B_F>.BIT(15:0),
B_F<a.DAT.BIT(15:0),
B_F<b.DAT.BIT(15:0),
B_G>(+[1,0]),
B_G>.BIT(15:0),
B_G<a.DAT.BIT(15:0),
B_G<b.DAT.BIT(15:0),
B_H>(+[1,0]),
B_H>.BIT(15:0),
B_H<a.DAT.BIT(15:0),
B_H<b.DAT.BIT(15:0),
I>.BIT(15:0)REAL,
I>.BIT(28:16)ASA,
I>.BIT(41:29)ASB,
I>.BIT(54:42)ASC,
I>.BIT(67:55)ASD,
I>.BIT(83:68)DSA,
I>.BIT(96:84)ASE,
I>.BIT(109:97)ASF,
I>.BIT(122:110)ASG,
I>.BIT(135:123)ASH,
I>.BIT(136:136)FB,
I>.BIT(152:137)DB_Ca,
I>.BIT(168:153)DB_Db,
I>.BIT(184:169)DB_Hb,
I>.BIT(185:185)MDBa,
I>.BIT(186:186)MDBb,
I>.BIT(187:187)MDSA,
I>.BIT(188:188)MDSB,
I>.BIT(189:189)MDSC,
 S>E(8191:0),
 S<A.DAT.BIT(15:0)
     "?,UDSA<a.DAT.BIT(15:0),
     UDSA<b.DAT.BIT(15:0),
     UDSA>.MPX.BIT(0:0),
     UDSA>.BIT(15:0)?",
 S<A.ADR.BIT(12:0),
 S<B.DAT.BIT(15:0)
     "?,UDSB<a.DAT.BIT(15:0),
     UDSB<b.DAT.BIT(15:0),
     UDSB>.MPX.BIT(0:0),
     UDSB>.BIT(15:0)?",
 S<B.ADR.BIT(12:0),
 S<C.DAT.BIT(15:0)
     "?,UDSC<a.DAT.BIT(15:0),
     UDSC<b.DAT.BIT(15:0),
     UDSC>.MPX.BIT(0:0),
     UDSC>.BIT(15:0)?",
 S<C.ADR.BIT(12:0),
 S<D.DAT.BIT(15:0),
 S<D.ADR.BIT(12:0),
 S>E.BIT(15:0),
 S>E.ADR.BIT(12:0),
 S>F.BIT(15:0),
 S>F.ADR.BIT(12:0),
 S>G.BIT(15:0),
 S>G.ADR.BIT(12:0),
 S>H.BIT(15:0),
S>H.ADR.BIT(12:0);
$ADDCONNECT
A<.DAT.BIT(15:0)   <-   S>E.BIT(15:0),
A_A<.DAT.BIT(15:0)   <-   S>E.BIT(15:0),
B>.FCT.BIT(0:0)   <-   I>.FB,
B<a.DAT.BIT(15:0)   <-   "?UDBa>.BIT(15:0),?"
    "?UDBa<a.DAT.BIT(15:0)    <-  ?"S>E.BIT(15:0)/
    "?UDBa<b.DAT.BIT(15:0)    <-  ?"A>.BIT(15:0)
    "?,UDBa>.MPX.BIT(0:0)    <-   I>.MDBa?",
B<b.DAT.BIT(15:0)   <-   "?UDBb>.BIT(15:0),?"
    "?UDBb<a.DAT.BIT(15:0)    <-  ?"S>F.BIT(15:0)/
    "?UDBb<b.DAT.BIT(15:0)    <-  ?"I>.REAL
    "?,UDBb>.MPX.BIT(0:0)    <-   I>.MDBb?",
B_A<a.DAT.BIT(15:0)    <-    A_A>.BIT(15:0),
B_A<b.DAT.BIT(15:0)    <-    S>F.BIT(15:0),
B_B<a.DAT.BIT(15:0)    <-    B_A>.BIT(15:0),
B_B<b.DAT.BIT(15:0)    <-    S>F.BIT(15:0),
B_C<a.DAT.BIT(15:0)    <-    I>.DB_Ca,
B_C<b.DAT.BIT(15:0)    <-    S>G.BIT(15:0),
B_D<a.DAT.BIT(15:0)    <-    B_C>.BIT(15:0),
B_D<b.DAT.BIT(15:0)    <-    I>.DB_Db,
B_E<a.DAT.BIT(15:0)    <-    B_C>.BIT(15:0),
B_E<b.DAT.BIT(15:0)    <-    B_D>.BIT(15:0),
B_F<a.DAT.BIT(15:0)    <-    B_B>.BIT(15:0),
B_F<b.DAT.BIT(15:0)    <-    B_E>.BIT(15:0),
B_G<a.DAT.BIT(15:0)    <-    S>H.BIT(15:0),
B_G<b.DAT.BIT(15:0)    <-    B_F>.BIT(15:0),
B_H<a.DAT.BIT(15:0)    <-    S>G.BIT(15:0),
B_H<b.DAT.BIT(15:0)    <-    I>.DB_Hb,
S<A.DAT.BIT(15:0)   <-   "?UDSA>.BIT(15:0),?"
    "?UDSA<a.DAT.BIT(15:0)    <-  ?"I>.DSA/
    "?UDSA<b.DAT.BIT(15:0)    <-  ?"B_F>.BIT(15:0)
    "?,UDSA>.MPX.BIT(0:0)    <-   I>.MDSA?",
S<A.ADR.BIT(12:0)   <-   I>.ASA,
S<B.DAT.BIT(15:0)   <-   "?UDSB>.BIT(15:0),?"
    "?UDSB<a.DAT.BIT(15:0)    <-  ?"B>.BIT(15:0)/
    "?UDSB<b.DAT.BIT(15:0)    <-  ?"B_G>.BIT(15:0)
    "?,UDSB>.MPX.BIT(0:0)    <-   I>.MDSB?",
S<B.ADR.BIT(12:0)   <-   I>.ASB,
S<C.DAT.BIT(15:0)   <-   "?UDSC>.BIT(15:0),?"
    "?UDSC<a.DAT.BIT(15:0)    <-  ?"B>.BIT(15:0)/
    "?UDSC<b.DAT.BIT(15:0)    <-  ?"B_H>.BIT(15:0)
    "?,UDSC>.MPX.BIT(0:0)    <-   I>.MDSC?",
S<C.ADR.BIT(12:0)   <-   I>.ASC,
S<D.DAT.BIT(15:0)   <-   B>.BIT(15:0),
S<D.ADR.BIT(12:0)   <-   I>.ASD,
S>E.ADR.BIT(12:0)   <-   I>.ASE,
S>F.ADR.BIT(12:0)   <-   I>.ASF,
S>G.ADR.BIT(12:0)   <-   I>.ASG,
S>H.ADR.BIT(12:0)   <-   I>.ASH;
```

```
*LIST /GEZ/PRP.SCR

$ADDMODULE
A>(.abs[20,0],-[15,1],.increment[20,2]),
A>.BIT(15:0),
A>.FCT.BIT(1:0),
A<.DAT.BIT(15:0)
     "?,UDA<a.DAT.BIT(15:0),
     UDA<b.DAT.BIT(15:0),
     UDA>.MPX.BIT(0:0),
     UDA>.BIT(15:0)?",
A_A>(.abs[20,0],-[15,1],.increment[20,2]),
A_A>.BIT(15:0),
A_A>.FCT.BIT(1:0),
A_A<.DAT.BIT(15:0),
B>(-[30,0],*[100,1],/[150,2],>[30,3],+[25,4]),
B>.BIT(15:0),
B>.FCT.BIT(2:0),
B<a.DAT.BIT(15:0)
     "?,UDBa<a.DAT.BIT(15:0),
     UDBa<b.DAT.BIT(15:0),
     UDBa<c.DAT.BIT(15:0),
     UDBa<d.DAT.BIT(15:0),
     UDBa>.MPX.BIT(1:0),
     UDBa>.BIT(15:0)?",
B<b.DAT.BIT(15:0)
     "?,UDBb<a.DAT.BIT(15:0),
     UDBb<b.DAT.BIT(15:0),
     UDBb<c.DAT.BIT(15:0),
     UDBb<d.DAT.BIT(15:0),
     UDBb<e.DAT.BIT(15:0),
     UDBb>.MPX.BIT(2:0),
     UDBb>.BIT(15:0)?",
B_A>(-[30,0],*[100,1],/[150,2],>[30,3],+[25,4]),
B_A>.BIT(15:0),
B_A>.FCT.BIT(2:0),
B_A<a.DAT.BIT(15:0),
B_A<b.DAT.BIT(15:0)
     "?,UDB_Ab<a.DAT.BIT(15:0),
     UDB_Ab<b.DAT.BIT(15:0),
     UDB_Ab<c.DAT.BIT(15:0),
     UDB_Ab>.MPX.BIT(1:0),
     UDB_Ab>.BIT(15:0)?",
I>.BIT(15:0)REAL,
I>.BIT(28:16)ASA,
I>.BIT(44:29)DSA,
I>.BIT(45:45)CSA,
I>.BIT(47:46)FA,
I>.BIT(48:48)CRP,
I>.BIT(61:49)ASB,
I>.BIT(74:62)ASC,
I>.BIT(77:75)FB,
I>.BIT(78:78)MDSA,
I>.BIT(80:79)FA_A,
I>.BIT(90:81)DRP,
I>.BIT(91:91)MDA,
I>.BIT(93:92)MDBa,
I>.BIT(96:94)MDBb,
I>.BIT(98:97)MDRP,
I>.BIT(100:99)MDRP@A,
I>.BIT(103:101)FB_A,
I>.BIT(104:104)CRHLP_101,
I>.BIT(120:105)DBa,
I>.BIT(136:121)DB_Ab,
I>.BIT(137:137)CRHLP_102,
I>.BIT(139:138)MDB_Ab,
I>.BIT(140:140)MDRHLP_101,
I>.BIT(156:141)DBb,
RHLP_101>.BIT(15:0),
RHLP_101<.DAT.BIT(15:0)
    "?,UDRHLP_101<a.DAT.BIT(15:0),
    UDRHLP_101<b.DAT.BIT(15:0),
    UDRHLP_101>.MPX.BIT(0:0),
    UDRHLP_101>.BIT(15:0)?",
RHLP_101<.CON.BIT(0:0),
RHLP_102>.BIT(15:0),
RHLP_102<.DAT.BIT(15:0),
RHLP_102<.CON.BIT(0:0),
RP<.DAT.BIT(9:0)
    "?,UDRP<a.DAT.BIT(9:0),
    UDRP<b.DAT.BIT(9:0),
    UDRP<c.DAT.BIT(9:0),
    UDRP>.MPX.BIT(1:0),
    UDRP>.BIT(9:0)?",
RP<.CON.BIT(0:0),
RP>.BIT(15:0),
S>B(.LOAD[75,0],.READ[50,1]),
S>B(8191:0),
S<A.DAT.BIT(15:0)
    "?,UDSA<a.DAT.BIT(15:0),
    UDSA<b.DAT.BIT(15:0),
    UDSA>.MPX.BIT(0:0),
    UDSA>.BIT(15:0)?",
S<A.ADR.BIT(12:0),
S<A.CON.BIT(0:0),
S>B.BIT(15:0),
S>B.ADR.BIT(12:0),
S>C.BIT(15:0),
S>C.ADR.BIT(12:0);
$ADDCONNECT
A>.FCT.BIT(1:0)   <-   I>.FA,
A<.DAT.BIT(15:0)   <-   "?UDA>.BIT(15:0),?"
    "?UDA<a.DAT.BIT(15:0)    <-  ?"RP>.BIT(15:0)/
    "?UDA<b.DAT.BIT(15:0)    <-  ?"S>B.BIT(15:0)
    "?,UDA>.MPX.BIT(0:0)    <-   I>.MDA?",
A_A>.FCT.BIT(1:0)   <-   I>.FA_A,
A_A<.DAT.BIT(15:0)   <-   RP>.BIT(15:0),
B>.FCT.BIT(2:0)   <-   I>.FB,
B<a.DAT.BIT(15:0)   <-   "?UDBa>.BIT(15:0),?"
    "?UDBa<a.DAT.BIT(15:0)    <-  ?"S>B.BIT(15:0)/
    "?UDBa<b.DAT.BIT(15:0)    <-  ?"A>.BIT(15:0)/
    "?UDBa<c.DAT.BIT(15:0)    <-  ?"I>.DBa/
    "?UDBa<d.DAT.BIT(15:0)    <-  ?"RHLP_101>.BIT(15:0)
    "?,UDBa>.MPX.BIT(1:0)    <-   I>.MDBa?",
B<b.DAT.BIT(15:0)   <-   "?UDBb>.BIT(15:0),?"
    "?UDBb<a.DAT.BIT(15:0)    <-  ?"S>C.BIT(15:0)/
    "?UDBb<b.DAT.BIT(15:0)    <-  ?"I>.REAL/
    "?UDBb<c.DAT.BIT(15:0)    <-  ?"RHLP_102>.BIT(15:0)/
    "?UDBb<d.DAT.BIT(15:0)    <-  ?"RHLP_101>.BIT(15:0)/
    "?UDBb<e.DAT.BIT(15:0)    <-  ?"I>.DBb
    "?,UDBb>.MPX.BIT(2:0)    <-   I>.MDBb?",
B_A>.FCT.BIT(2:0)   <-   I>.FB_A,
B_A<a.DAT.BIT(15:0)   <-   B>.BIT(15:0),
B_A<b.DAT.BIT(15:0)   <-   "?UDB_Ab>.BIT(15:0),?"
    "?UDB_Ab<a.DAT.BIT(15:0)    <-  ?"S>C.BIT(15:0)/
    "?UDB_Ab<b.DAT.BIT(15:0)    <-  ?"I>.DB_Ab/
    "?UDB_Ab<c.DAT.BIT(15:0)    <-  ?"RHLP_102>.BIT(15:0)
    "?,UDB_Ab>.MPX.BIT(1:0)    <-   I>.MDB_Ab?",
RHLP_101<.DAT.BIT(15:0)   <-   "?UDRHLP_101>.BIT(15:0),?"
    "?UDRHLP_101<a.DAT.BIT(15:0)    <-  ?"B_A>.BIT(15:0)/
    "?UDRHLP_101<b.DAT.BIT(15:0)    <-  ?"B>.BIT(15:0)
    "?,UDRHLP_101>.MPX.BIT(0:0)    <-   I>.MDRHLP_101?",
RHLP_101<.CON.BIT(0:0)   <-   I>.CRHLP_101,
RHLP_102<.DAT.BIT(15:0)   <-   B_A>.BIT(15:0),
RHLP_102<.CON.BIT(0:0)   <-   I>.CRHLP_102,
RP<.DAT.BIT(9:0)   <-   "?UDRP>.BIT(9:0),?"
    "?UDRP<a.DAT.BIT(9:0)    <-  ?"A>.BIT(9:0)/
    "?UDRP<b.DAT.BIT(9:0)    <-  ?"A_A>.BIT(9:0)/
    "?UDRP<c.DAT.BIT(9:0)    <-  ?"I>.DRP
    "?,UDRP>.MPX.BIT(1:0)    <-   I>.MDRP?",
RP<.CON.BIT(0:0)   <-   I>.CRP,
S<A.DAT.BIT(15:0)   <-   "?UDSA>.BIT(15:0),?"
    "?UDSA<a.DAT.BIT(15:0)    <-  ?"I>.DSA/
    "?UDSA<b.DAT.BIT(15:0)    <-  ?"B>.BIT(15:0)
    "?,UDSA>.MPX.BIT(0:0)    <-   I>.MDSA?",
S<A.ADR.BIT(12:0)   <-   I>.ASA,
S<A.CON.BIT(0:0)   <-   I>.CSA,
S>B.ADR.BIT(12:0)   <-   I>.ASB,
S>C.ADR.BIT(12:0)   <-   I>.ASC;
```

```
*LIST /GEZ/PRP.OTB

MIMOLA PART B (GCOS VERSION CM) VERSION 3.0 OF 02/27/80
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
***   12 ESBS READ    (       40 WEIGHTED)
***    0 ESBS CREATED (        0 WEIGHTED)
***   12 ESBS TOTAL   (       40 WEIGHTED)
<<<<**** CHARACTERSTRING ACCEPTED ****>>>>
 0 HELP-STORAGECELLS (OR REGISTERS) USED IN PROGRAM
ESTIMATED RUN TIME:       7406
*** GROUP : A
A               25 FREE DUPLICATES                  , $:   3.0 +  2.5 = $  5.5
   U/C:    15.9091                                    FREQ:        35 =   87.50%
   FUNCTIONS :  .abs[20], -[15], .increment[20],
   PORT USE:  OUTPUT 0 :    5X  1 :   35X
              INPUT  0 :    5X  1 :   35X
    >                               .AUTO(.READ     ) FREQ:        35 =   87.50%
           3 FUNCTIONS(EXCL .LOAD&.READ)
      DATA           15:  0                FREQ:      35 = 87.50 %
      FUNCTION        1:  0   ADR  D-BITS     MODULE&PORT   S-BITS  FREQ
                                0     1:  0     I>            FA              35 = 87.5%
   <                                .AUTO(NONE      ) FREQ:        35 =   87.50%
      DATA           15:  0   ADR  D-BITS     MODULE&PORT   S-BITS  FREQ
                                0    15:  0     S>B           15:  0          10 = 25.0%
                                1    15:  0     RP>           15:  0          25 = 62.5%
--------------------------------------------------------------------------------------
A_A              0 FREE DUPLICATES                  , $:   3.0 +  1.0 = $  4.0
   U/C:    6.2500                                     FREQ:        10 =   25.00%
   FUNCTIONS :  .abs[20], -[15], .increment[20],
   PORT USE:  OUTPUT 0 :   30X  1 :   10X
              INPUT  0 :   30X  1 :   10X
    >                               .AUTO(.READ     ) FREQ:        10 =   25.00%
           3 FUNCTIONS(EXCL .LOAD&.READ)
      DATA           15:  0                FREQ:      10 = 25.00 %
      FUNCTION        1:  0   ADR  D-BITS     MODULE&PORT   S-BITS  FREQ
                                0     1:  0     I>            FA_A            10 = 25.0%
   <                                .AUTO(NONE      ) FREQ:        10 =   25.00%
      DATA           15:  0   ADR  D-BITS     MODULE&PORT   S-BITS  FREQ
                                0    15:  0     RP>           15:  0          10 = 25.0%
--------------------------------------------------------------------------------------
*** GROUP : B
B                0 FREE DUPLICATES                  , $:   5.0 +  9.5 = $ 14.5
   U/C:    6.5517                                     FREQ:        38 =   95.00%
   FUNCTIONS :  -[30], *[100], /[150], >[30], +[25],
   PORT USE:  OUTPUT 0 :    7X  1 :   33X
              INPUT  0 :    2X  2 :   38X
    >                               .AUTO(.READ     ) FREQ:        38 =   95.00%
           5 FUNCTIONS(EXCL .LOAD&.READ)
      DATA           15:  0                FREQ:      33 = 82.50 %
      FUNCTION        2:  0   ADR  D-BITS     MODULE&PORT   S-BITS  FREQ
                                0     2:  0     I>            FB              38 = 95.0%
   < a                              .AUTO(NONE      ) FREQ:        38 =   95.00%
      DATA           15:  0   ADR  D-BITS     MODULE&PORT   S-BITS  FREQ
```

```
                             0    15:  0     RHLP_101>    15:  0        5 = 12.5%
                             1    15:  0     I>          DBa           10 = 25.0%
                             2    15:  0     A>           15:  0       10 = 25.0%
                             3    15:  0     S>B          15:  0       13 = 32.5%
   < b                       .AUTO(NONE       ) FREQ:     38 =    95.00%
      DATA          15:  0   ADR  D-BITS    MODULE&PORT   S-BITS   FREQ
                             0    15:  0     I>          DBb            5 = 12.5%
                             1    15:  0     RHLP_101>    15:  0        5 = 12.5%
                             2    15:  0     RHLP_102>    15:  0        5 = 12.5%
                             3    15:  0     I>          REAL           5 = 12.5%
                             4    15:  0     S>C          15:  0       18 = 45.0%
--------------------------------------------------------------------------------
B_A              0 FREE DUPLICATES              , $:    5.0 +  4.0 = $  9.0
   U/C:    4.1667                                FREQ:      15 =    37.50%
   FUNCTIONS :  -[30], *[100], /[150], >[30], +[25],
   PORT USE:  OUTPUT 0 :    25X   1 :     15X
              INPUT  0 :    25X   2 :     15X
    >                        .AUTO(.READ      ) FREQ:       15 =    37.50%
           5 FUNCTIONS(EXCL .LOAD&.READ)
      DATA          15:  0             FREQ:      15 = 37.50 %
      FUNCTION       2:  0   ADR  D-BITS    MODULE&PORT   S-BITS   FREQ
                             0     2:  0     I>          FB_A          15 = 37.5%
   < a                       .AUTO(NONE       ) FREQ:      15 =    37.50%
      DATA          15:  0   ADR  D-BITS    MODULE&PORT   S-BITS   FREQ
                             0    15:  0     B>           15:  0       15 = 37.5%
   < b                       .AUTO(NONE       ) FREQ:      15 =    37.50%
      DATA          15:  0   ADR  D-BITS    MODULE&PORT   S-BITS   FREQ
                             0    15:  0     RHLP_102>    15:  0        5 = 12.5%
                             1    15:  0     I>          DB_Ab          5 = 12.5%
                             2    15:  0     S>C          15:  0        5 = 12.5%
--------------------------------------------------------------------------------
*** GROUP : F
*** GROUP : I
I               WIDTH=157 * ESB=     1884, $:    7.5 +  0.   = $  7.5
   U/C:    13.2696                               FREQ:      40 =   100.00%
   PORT USE:  OUTPUT 1 :    40X
              INPUT  0 :    40X
    >                        .AUTO(.READ      ) FREQ:       40 =   100.00%
      DATA          REAL     WIDTH= 16                FREQ:      5 = 12.50 %
      DATA          ASA      WIDTH= 13                FREQ:     19 = 47.50 %
      DATA          DSA      WIDTH= 16                FREQ:      1 =  2.50 %
      DATA          CSA      WIDTH=  1 DISABLE        FREQ:     19 = 47.50 %
      DATA          FA       WIDTH=  2 .increment     FREQ:     35 = 87.50 %
      DATA          CRP      WIDTH=  1 enable         FREQ:     40 = 100.0 %
      DATA          ASB      WIDTH= 13                FREQ:     23 = 57.50 %
      DATA          ASC      WIDTH= 13                FREQ:     18 = 45.00 %
      DATA          FB       WIDTH=  3                FREQ:     38 = 95.00 %
      DATA          MDSA     WIDTH=  1                FREQ:     19 = 47.50 %
      DATA          FA_A     WIDTH=  2                FREQ:     10 = 25.00 %
      DATA          DRP      WIDTH= 10                FREQ:     10 = 25.00 %
      DATA          MDA      WIDTH=  1       001      FREQ:     35 = 87.50 %
      DATA          MDBa     WIDTH=  2                FREQ:     38 = 95.00 %
      DATA          MDBb     WIDTH=  3                FREQ:     38 = 95.00 %
      DATA        MDRP        WIDTH=  2       002    FREQ:       40 = 100.0 %
      DATA        MDRP@A      WIDTH=  2              FREQ:        9 = 22.50 %
      DATA        FB_A        WIDTH=  3              FREQ:       15 = 37.50 %
      DATA        CRHLP_1 1   WIDTH=  1 DISABLE      FREQ:       10 = 25.00 %
      DATA        DBa         WIDTH= 16              FREQ:       10 = 25.00 %
      DATA        DB_Ab       WIDTH= 16              FREQ:        5 = 12.50 %
      DATA        CRHLP_102   WIDTH=  1 DISABLE      FREQ:       10 = 25.00 %
      DATA        MDB_Ab      WIDTH=  2              FREQ:       15 = 37.50 %
      DATA        MDRHLP_101  WIDTH=  1              FREQ:       10 = 25.00 %
      DATA        DBb         WIDTH= 16              FREQ:        5 = 12.50 %
--------------------------------------------------------------------------------
*** GROUP : R
RHLP_101                                       , $:    1.0 +  2.5 = $  3.5
    U/C:    10.7143                                  FREQ:       15 =   37.50%
    PORT USE:  OUTPUT 0 :    30X   1 :    10X
               INPUT  0 :    30X   1 :    10X
    <           CONTROLBITS:1 .AUTO(.LOAD      ) FREQ:      10 =   25.00%
       DATA          15:  0    ADR  D-BITS    MODULE&PORT   S-BITS  FREQ
                                0     15:  0    B>            15:  0        5 = 12.5%
                                1     15:  0    B_A>          15:  0        5 = 12.5%
     >                          .AUTO(.READ      ) FREQ:      10 =   25.00%
       DATA          15:  0                FREQ:     10 = 25.00 %
--------------------------------------------------------------------------------
RHLP_102                                       , $:    1.0 +  1.0 = $  2.0
    U/C:    18.7500                                  FREQ:       15 =   37.50%
    PORT USE:  OUTPUT 0 :    30X   1 :    10X
               INPUT  0 :    30X   1 :    10X
    <           CONTROLBITS:1 .AUTO(.LOAD      ) FREQ:      10 =   25.00%
       DATA          15:  0    ADR  D-BITS    MODULE&PORT   S-BITS  FREQ
                                0     15:  0    B_A>          15:  0       10 = 25.0%
     >                          .AUTO(.READ      ) FREQ:      10 =   25.00%
       DATA          15:  0                FREQ:     10 = 25.00 %
--------------------------------------------------------------------------------
RP                                             , $:    1.0 +  3.5 = $  4.5
    U/C:    22.2222                                  FREQ:       40 =  100.00%
    PORT USE:  OUTPUT 0 :     5X   1 :    35X
               INPUT  1 :    40X
    <           CONTROLBITS:2 .AUTO(.LOAD      ) FREQ:      40 =  100.00%
       DATA           9:  0    ADR  D-BITS    MODULE&PORT   S-BITS  FREQ
                                0      9:  0    I>          DRP            10 = 25.0%
                                1      9:  0    A_A>           9:  0       10 = 25.0%
                                2      9:  0    A>             9:  0       25 = 62.5%
     >                          .AUTO(.READ      ) FREQ:      35 =   87.50%
       DATA          15:  0                FREQ:     35 = 87.50 %
--------------------------------------------------------------------------------
*** GROUP : S
S               ( 8191:     0) -3 FREE PORTS, $:   27.0 +  4.0 = $ 31.0
    U/C:     3.1452                                  FREQ:       39 =   97.50%
    FUNCTIONS :  .LOAD[75], .READ[50],
    PORT USE:  OUTPUT 0 :     7X   1 :    25X  2 :    8X
               INPUT  0 :    21X   1 :    19X
    < A         CONTROLBITS:1 .AUTO(.LOAD      ) FREQ:      19 =   47.50%
       DATA          15:  0    ADR  D-BITS    MODULE&PORT   S-BITS  FREQ
```

```
                               0    15:  0     B>            15:  0        18 = 45.0%
                               1    15:  0     I>        DSA                1 =  2.5%
       ADDRESS      12:  0    ADR  D-BITS    MODULE&PORT   S-BITS  FREQ
                               0    12:  0     I>        ASA               19 = 47.5%
     >B                       .AUTO(.READ      ) FREQ:       23 =   57.50%
       DATA         15:  0            FREQ:      23 = 57.50 %
       ADDRESS      12:  0    ADR  D-BITS    MODULE&PORT   S-BITS  FREQ
                               0    12:  0     I>        ASB               23 = 57.5%
     >C                       .AUTO(.READ      ) FREQ:       18 =   45.00%
       DATA         15:  0            FREQ:      18 = 45.00 %
       ADDRESS      12:  0    ADR  D-BITS    MODULE&PORT   S-BITS  FREQ
                               0    12:  0     I>        ASC               18 = 45.0%
-----------------------------------------------------------------------------------------
*** GROUP : X
AV. # OF FIXED OR EQUIV BITS :  0.000e+00,   STD. DEVIATION :  0.000e+00
AV. # OF NOT EQUIVAL. FIELDS :  1.265e+01,   STD. DEVIATION :  2.209e+00
\
DISTRIBUTION OF FIXED OR EQUIVALENCED MICRO BIT USES
----------------------------------------------------
  0 TO   3 BITS :100%  ********************************************************
\c*******************************************
  4 TO   7 BITS :  0%
  8 TO  11 BITS :  0%
 12 TO  15 BITS :  0%
 16 TO  19 BITS :  0%
 20 TO  23 BITS :  0%
 24 TO  27 BITS :  0%
 28 TO  31 BITS :  0%
 32 TO  35 BITS :  0%
 36 TO  39 BITS :  0%
 40 TO  43 BITS :  0%
 44 TO  47 BITS :  0%
 48 TO  51 BITS :  0%
 52 TO  55 BITS :  0%
 56 TO  59 BITS :  0%
 60 TO  63 BITS :  0%
 64 TO  67 BITS :  0%
 68 TO  71 BITS :  0%
 72 TO  75 BITS :  0%
 76 TO  79 BITS :  0%
 80 TO  83 BITS :  0%
 84 TO  87 BITS :  0%
 88 TO  91 BITS :  0%
 92 TO  95 BITS :  0%
 96 TO  99 BITS :  0%
100 TO 103 BITS :  0%
DISTRIBUTION OF NOT EQUIVALENCED MICRO FIELD USES
----------------------------------------------------
   0FIELDS:  0%
   1FIELDS:  0%
   2FIELDS:  0%
   3FIELDS:  0%
   4FIELDS:  0%
   5FIELDS:  2%  ++
  6FIELDS:  0%
  7FIELDS:  2%  ++
  8FIELDS:  7%  +++++++
  9FIELDS:  0%
 10FIELDS:  0%
 11FIELDS:  0%
 12FIELDS:  0%
 13FIELDS: 62%  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 14FIELDS: 12%  ++++++++++++
 15FIELDS: 12%  ++++++++++++
 16FIELDS:  0%
 17FIELDS:  0%
 18FIELDS:  0%
 19FIELDS:  0%
 20FIELDS:  0%
 21FIELDS:  0%
 22FIELDS:  0%
 23FIELDS:  0%
 24FIELDS:  0%
 25FIELDS:  0%
\
# OF ENABLE BITS :     5
# OF FCT-SEL BITS:    10
# OF MPX-ADR BITS:    16
# OF MPXER INPUTS:    21
# OF CONNECTIONS :    31
MODULE COST/ BIT :     46.0$, MPX COST :    28.0$ MICRO MEMORY:     7.5
SUM =      74.0 * WIDTH +      7.5
GRAND TOTAL OF    1191.54$, TIME * COST =  8.825e+06
AV. TREE HEIGHT :  185.15
```

This Appendix contains a list of the command files that may be used to execute the available MSS software. Currently these files exist only on CNO's GCOS D.

1. Initialization

Before any of the command files listed here may be used, a catalog must be created. This catalog creation must only be done once. The command H1196B/CREATE invokes a command file that first asks for the catalog name, creates a catalog with that name, and then creates a file called SNUMB within the catalog. It is suggested that the catalog name be the initials of the user or the name of the project.

Example:

```
H1196/CREATE
CATALOG? ccc
```

This example creates a catalog called CCC under the current user's account. It also places empty files called SNUMB and JCL in the catalog.

2. Running MIMB - MIMOLA Compiler and Allocator

MIMB is the MSS tool that was used for all examples in the MIMOLA Primer.

Example:

```
H1196B/BRUN
CATALOG?  ccc
ROOT NAME?  rrr
MIMB VER?  CN
```

This command file creates a JCL file for the batch run of the CN version of MIMB. The JCL assumes that the naming convention listed in this appendix is used and that the file rrr.INP exists in the catalog ccc. The files rrr.OTB, rrr.COD, rrr.GEN and rrr.SCR are created if they do not exist. Any previous contents of these files are then automatically erased. The batch job is automatically submitted and its progress may be monitored with the JMON command.

3. Running MIMD

MIMD requires that the information contained in the SCR and GEN files, which are output from MIMB, be combined into a single file. This process is automatically performed by the command file H1196B/DRUN.

Example:

```
H1196B/DRUN
CATALOG?  ccc
ROOT NAME?  rrr
```

The file rrr.IND is created from the files rrr.SCR and rrr.GEN, which are found in the catalog specified by ccc. Minor changes are made to this combined file and when complete, it is written out to the catalog ccc. The files rrrAIF and rrr.OUT are created, if they don't already exist. Any previous contents of these files are then automatically erased. The batch job is submitted and its progress may be monitored with the JMON command.

4. Running MIDL

If the naming conventions listed in Section 8 of this appendix are followed, the command file H1196B/MIDLRUN may be used to submit a batch run of MIDL.

Example:

```
H1196B/MIDLRUN
CATALOG?  ccc
ROOT NAME?  rrr
```

MIDL creates a JCL file that assumes the following files exist in the named catalog: rrr.SCR, NAMCON and USRFIL. The files rrrSUD and rrr.OUT are created, if they don't already exist. Any previous contents of these files are then automatically erased. The batch job is submitted and its progress may be monitored with the JMON command.

5. Clean-up

To simplify the releasing of space, the command file H1196B/CLEAN may be used. This file releases all files that use the standard qualifiers for a particular catalog and root name, except for the initial input file with the qualifier .INP.

Example:

```
H1196B/CLEAN
CATALOG?  ccc
ROOT NAME?  rrr
```

This example deletes all of the MSS1 files associated with the root rrr under the catalog ccc. This command also places an entry in SNUMB to indicate the action that has been taken.

6. Other Software

/FRUN	Critical path analysis
/MRUN	TREEMOLA to MIMOLA Conversion
/NRUN	MIMOLA to TREEMOLA Conversion
/PPRUN	Compile and execute Pascal program
/PRUN	TREEMOLA to Pascal Conversion
/TRUN	TREEMOLA Tree modification

See attached Diagram for the assumed filename qualifiers. All of these command run files requires "CATALOG?" and "ROOT NAME?".

7. Notes

1) The root name is the design file name without any qualifiers. The root name may be at most four characters, due to GCOS3 filename conventions.

2) A list of all batch SNUMBs submitted for a user is maintained in the catalog ccc in the file SNUMB.

3) The most recent JCL file created is saved in the catalog ccc in the file JCL (i.e., the JCL may be rerun if only an input file is changed).

4) As DL does not accept periods in filenames, the qualifiers for files accessed by DL do not contain periods.

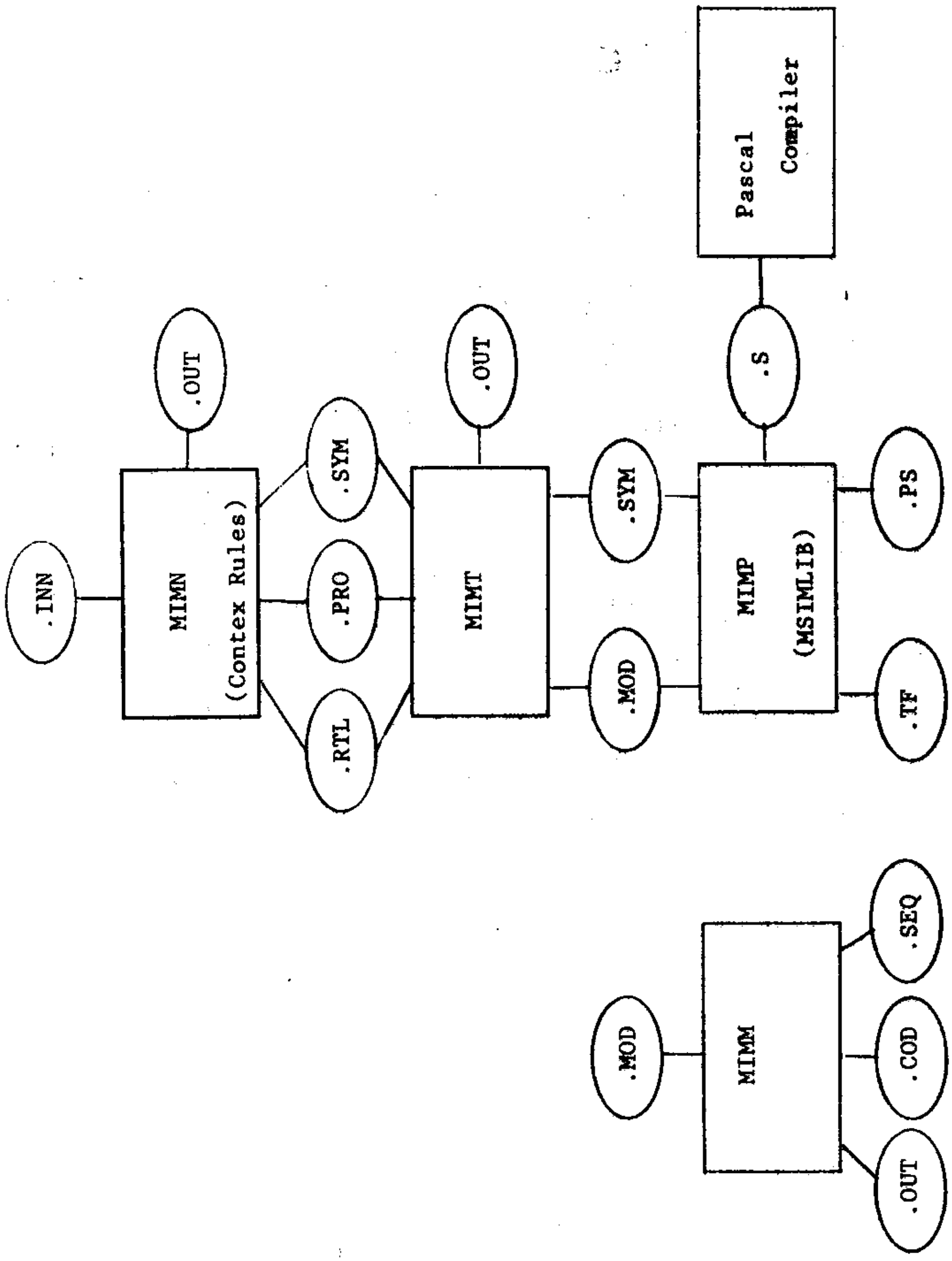
8. MSS Standard Qualifiers

The following list contains the correct standard qualifiers used by the MSS.

.AA	MIMP output file
.AB	MIMP output file
.AC	MIMP output file
.AD	MIMP output file
AIF	Assembler Input File - binary microcode in format for DL assembler
.COD	MIMB COD File - Symbolic Microcode
DL	Compiled DL model
.GEN	MIMB GEN file - new version of MIMOLA program
.IND	MIMD INP file - created automatically from MIMB SCR and MIMB GEN
.INN	MIMN INP file - created manually from MIMB GEN
.INP	MIMB INP file - MIMOLA program
MAC	DL Assembled Microcode
.MOD	MSS2 MODBRAC - RTL program in TREEMOLA as modified by MIMT
.OTB	Output of MIMB
.OTD	Output of MIMD
.OUT	General purpose output file for all MSS programs
.PRO	MSS2 PROGRASS - ??
.PS	MSS2 PSCR - Scratch file used by MIMP
.RTL	MSS2 RTLBRAC - RTL program in TREEMOLA bracket notation
.S	MSS2 SIMPAS - Pascal program generated by MIMP
.SCR	MIMB SCR File - new version of MIMOLA hardware declaration
.SEQ	MSS2 SEQMIM - Program in MIMOLA format
SUD	System Under Design - DL description output of MIDL
.SYM	MSS2 SYMTAB - Symbol table
.TIM	MSS2 TIMLST - Timing list generated by MIMF

Other Names

NAMCON	MIMOLA to DL Name Conversion Table
USRFIL	MIMOLA to DL User Template File



Additional MSS Tools and Intermediate File Names