CMU-CS-79-150

The Design and Implementation of a PMS Level Hardware Interconnection Language

Brad W. Hosler
Department of Computer Science
Carnegie-Mellon University
24 October 1979

The work reported in this document was supported in part by the MCF project office, US Army, under contract DAAK80-79-C-0767, and in part by the Department of Electrical Engineering, Carnegie-Mellon University.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the MCF project office, the US Army, or the US Government.

Table of Contents

List of Figures

1. Introduction

The PMS notation was first introduced by Bell&Newell [Bell, 1971] as a means to formally describe the structure of a digital system. Their notation is mainly a graphic one showing the components of a system and how they are connected. A graphic representation of a systems structure is a very easy thing for a person to understand, unfortunately, it is difficult for a machine to interpret and it lacks information for use by a simulator and application programs. Specifically, it lacks information about the behavior of the system components.

PMS is the top level in the hierarchy of digital systems descriptions. At the level below PMS, the behavioral level, Bell&Newell introduced a notation called ISP. At CMU, ISP has been developed into a formal language [Barbacci,1977] that has been used as a design tool which covers a wider area of application than any other hardware description language. Some of the applications it has been used for are fault analysis, architecture evaluation, architecture certification, simulation, design automation and automatic software generation. A model of the ISPS system (with the PMS system) is shown in Figure1-1. The main characteristic is the use of a formally defined intermediate representation for the parse tree. This intermediate format (called Global Data Base or GDB) can be easily used by a multitude of applications, written in any language, and running on any machine.

There are several reasons for the development of a formal PMS interconnection language. The primary reason is, because there is no way to specify connections and other structural information in the ISPS language, the description of a system in ISPS can get very large and unwieldy. Using the PMS language, systems can be described using smaller PMS and ISPS components which are easier to program and debug. PMS will also promote modularity in the description.

Because of the wide use of ISPS and its ability to describe components in a digital system, the PMS system will produce a GDB file using the PMS description and the GDB files of the components in that description. By using the same internal format (GDB) it is possible to develop applications for formal machine descriptions that require both structural and behavioral information. The PMS notation thus complements ISPS and allows a designer to iterate over the design of a system, adding or eliminating structural and behavioral information. Thus, one can conceive of a design process in which the designer starts with only a few large components described in ISPS which are then decomposed into smaller components until at the end, the only behavioral descriptions (ISPS) are those corresponding to the "black boxes" (IC chips, boards, modules, etc.) which are available off the shelf.

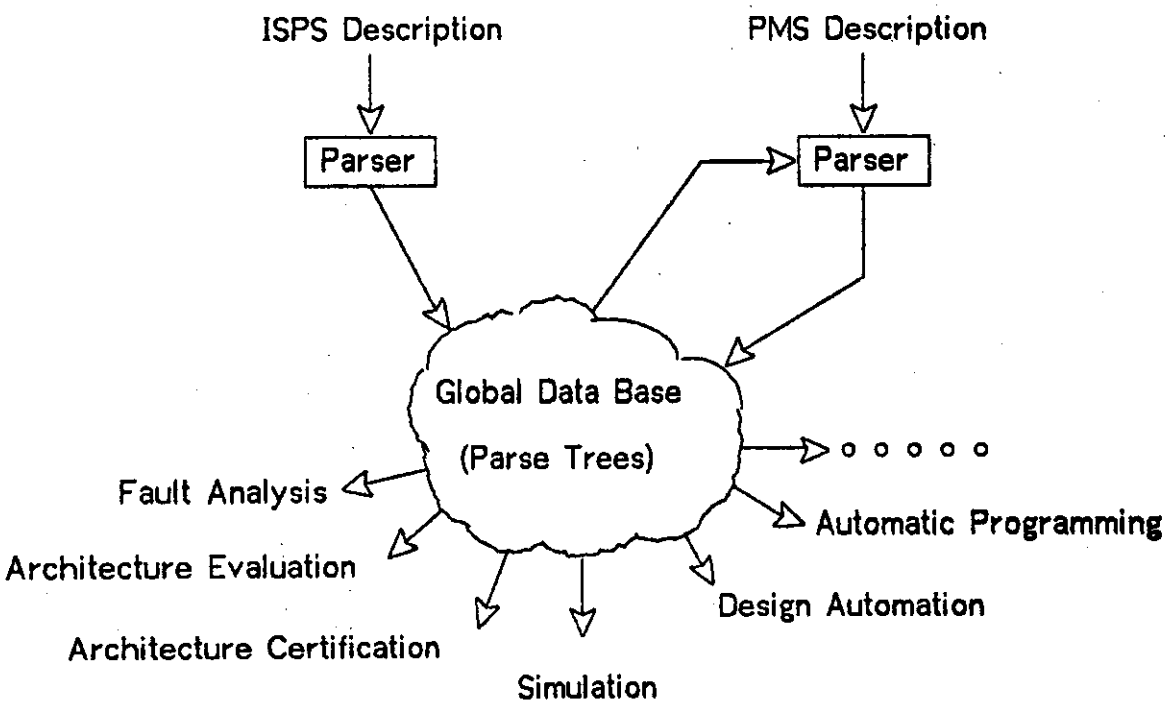


Figure 1-1: The PMS and ISPS System

Other implementations of PMS level languages [Knudsen, 1973] do not have established intermediate formats through which application programs can analyze the description. Typically, these implementations have developed their own description languages (generally complex) and their own systems analysis facility. In this respect the system developed here is unique.

2. The PMS Language

The PMS language describes a picture of a digital system. At the PMS level a computer could be represented by the picture shown in figure 2-1. The picture makes apparent that the whole computer is composed of three subcomponents connected together in a certain way. To do this with a language that is not graphic, the language must specify three things:

1. The name and interfaces of the new module being described.
2. Instantiations of components as the sub-components of the system.
3. Connections between sub-component interfaces and between sub-component and module interfaces.

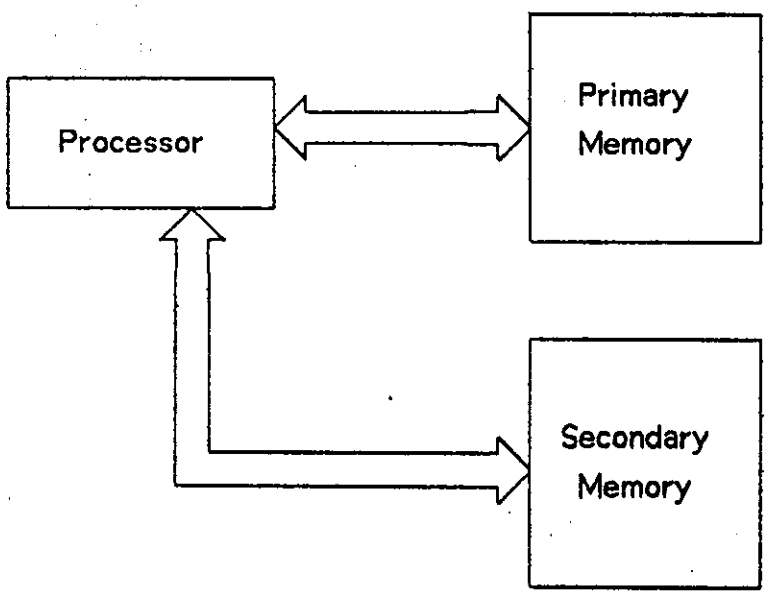


Figure 2-1: A Computer at the PMS Level

2.1 The Syntax

The syntax for the language was chosen to be simialr to the constructs in ISPS so that a person familiar with ISPS[1] will find the PMS language easy to learn and understand. Before describing the syntax a few definitions often used terms is necessary.

[1] A requirement if anything useful is to be done using the PMS system.

A module is a description of a type of component. A module consists of a header and a body.

The body is code in either ISPS or PMS. ISPS bodies describe the behavior of a module, PMS bodies describe how modules are connected together to make up another module. Modules described in ISPS are primitive (ie. not decomposable) while PMS modules are complex (ie. are described as interconnections of simpler components).

The header consists of a module name and a list of hooks[2] (e.g. registers, lines) through which the body may interact with the "outside".

A hook is of the form of an ISPS EHEAD (e.g. X<0:15>) followed by a list of attribute-value pairs (e.g. {SPEED:45;TECHNOLOGY:MOS}). The attribute-value pairs are mnemonics to specify physical attributes of the hook (described later).

2.1.1 Module Declarations

The module declaration statement must be the first line in a PMS description. This statement defines the name of the module being described as well as any hooks it might have. The syntax for declaring the new module name and hooks is:

```
MODULE <module.name>(<hook><qualifier>,<hook><qualifier>,...) :=
```

where <hook> has the form:

```
identifier < left.bit.name : right.bit.name >
```

or

```
identifier <>
```

and <qualifier> has the form:

```
{attribute:value,value,...;attribute:value,value,...}
```

Attributes are user specified identifiers; Values are either identifiers or constants. Qualifiers are application dependent specifications and are used to check the validity of connections. They are checked for syntactic correctness by the PMS compiler, but no attempt is made to ascertain their semantic correctness other than verifying that a connection

---

[2]The word hook will be used throughout this report as a technical term for an interface between the module body and the outside world.

is valid.

2.1.2 Instantiations

The instantiation statement declares an instantiation of a module as a sub-component within the PMS description. Every sub-component in a PMS description has to be instantiated separately even if there are multiple instances of the sub-component.

The syntax for the instantiation statement is one of the following:

```
<instantiation.name> := <GDB.file.name>
                     := <PMS.file.name>
                     := <PMS.file.name>,<attribute.file.name>
```

The file named <attribute.file.name> contains 'special case' attributes (described later) that are used by the system to validate connections made while processing the PMS file.

2.1.3 Connections

The connect statement defines connections made in the PMS description. There are two types of connections that can be made (ie. sub-component to sub-component or sub-component to module) but they are both handled by the same general connect statement. The general syntax for the statement is:

```
CONNECT <instance.name>(<hook>) := <instance.name>(<hook>)
```

Instance names can be replaced by the module name and if the hook being connected is a module hook, the module name can be omitted. There is no differentiation between the sides of a connect statement.

2.2 Example: PMS Module

An example will make the use of this language more clear. This example is very simple and does not illustrate any of the more advanced features of the system, but it does give a general feel for the kinds of things the system can do and the syntax of the language.

Imagine a simple system, consisting of a processor connected to a memory. At the PMS level this is shown in Figure 2-2. The ISPS descriptions of the processor and memory are shown below.

```
PROCESSOR(ADDRESS<0:15>,DATA<0:7>,R,W<0:2>) :=
    BEGIN
        (Description of processor behavior)
    END

MEMORY(DATA<7:0>,ADDRESS<15:0>,R,W<2:0>) :=
    BEGIN
        (Description of memory behavior)
    END
```

The PMS description of the desired module will involve the instantiation of a processor and memory and then the connection of the address,data, and control lines. The PMS description of the system is given below. Note that the instantiation statements use the parsed versions (ie. GDB files) of the ISPS descriptions given above. Instantiation statements always use either GDB files or PMS files.

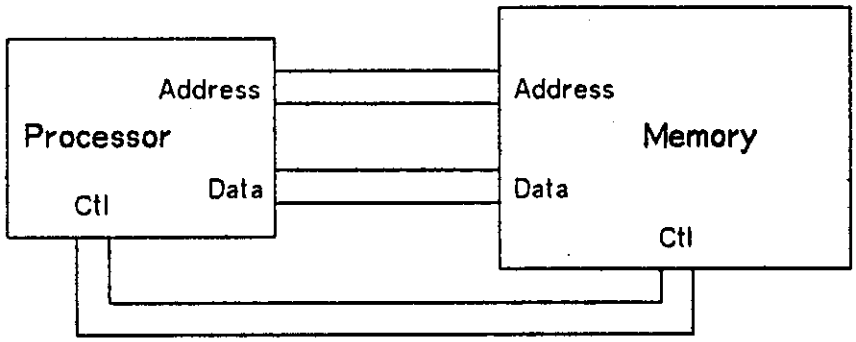


Figure 2-2: PMS Picture of a Processor and Memory

```
MODULE SYS :=

    PROC := PROCESSOR.GDB                                  instantiate processor
    MEM := MEMORY.GDB                                      instantiate memory

    CONNECT PROC(ADDRESS<0:15>) := MEM(ADDRESS<15:0>)      connect address lines
    CONNECT MEM(DATA<7:0>) := PROC(DATA<0:7>)              connect data lines
    CONNECT MEM(R,W<2:0>) := PROC(R,W<0:2>)                connect control lines
```

3. The PMS Compiler

The PMS compiler uses as input a PMS description and produces a GDB parse tree. The compiler does all error checking and building of the parse tree on the fly. As the parse tree is built it is immediately shipped to the output file.

3.1 Data Structures

As the compiler goes through the PMS file it generates a symbol table that has a tree structure. Every statement in the PMS file generates a tree that contains all the necessary information for error and attribute checking.

The main data structure in the generated tree is an integer matrix of arbitrary length (currently 500) and eight words wide with a parallel string array (length 500). There are five types of nodes allowed in this matrix (See fig. 3-1). The first word in each node determines the type of that node.

In a name-pair node, the second and third words contain the decimal values of numbers found in a word or bit FS-Set (word and bit name boundaries). If there is a pair of numbers, the first number is in word 2 of the node and the second is in word 3. If the FS-Set contains a single number then both word 2 and word 3 are set to that number. If there is no number in the FS set (ie. a single un-named bit) both word 2 and word 3 are set to -1. The differentiation between cases is necessary for generating the proper GDB output.

The fourth and fifth words in a name-pair node each contain the radix of the respective number preceding them. Note that all the numbers are stored in the table in their decimal equivalent value. Thus if the compiler reads a hex value of 1F for a boundary, this information is stored in the table as 31 ($1F_{16} = 31_{10}$) with a radix of 16. All error checking is done with the decimal equivalent values, and if the information is shipped to the output file the number will be restored to its original radix.

For the remaining four nodes (module, instantiation, hook, and connect) the first four words in the node all serve the same purpose. The first word is to identify the type of the node. The second word contains an index into the table that points to a hook node. The third and fourth words contain an index into the table that points to a name-pair node. The third word is associated with a Word FS-Set and the fourth word is associated with a Bit FS-Set. A value of -999 indicates a null pointer.

In the connect node the fifth through eighth words are not used and are set to -999. Likewise, the sixth through eighth words are not used in name-pair, module or instantiation nodes and are set to -999 for consistency.

Module node

1	first hook node or -999	word dimension node or -999	bit dimension node or -999	first qualifier node or -999	-999	-999	-999

Instantiation node

2	first hook node or -999	word dimension node or -999	bit dimension node or -999	first qualifier node or -999	-999	-999	-999

Hook node

3	first hook node or -999	word dimension node or -999	bit dimension node or -999	first qualifier node or -999	next hook node or -999	declared (if 1)	connected (if 1)

Name-pair node

4	left word/bit name	right word/bit name	left word/bit radix	right word/bit radix	-999	-999	-999

Connect node

5	first hook node or -999	word dimension node or -999	bit dimension node or -999	-999	-999	-999	-999

Figure 3-1: Node Types

The fifth word of hook, module and instantiation nodes contain an index into a RECORD!POINTER array that has a pointer to the records containing the qualifier for that hook. The sixth word of a hook node contains an index into the table that points to a hook node that is a brother to this hook. The seventh word contains a 1 if that hook has been defined in the GDB output file and the eighth word contains a 1 if the hook has been connected.

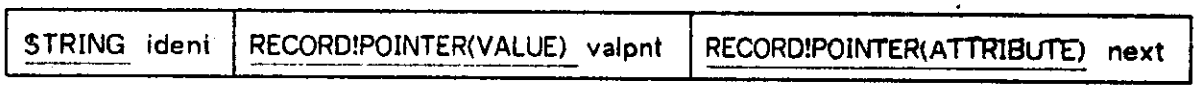
The parallel string array contains the identifiers associated with module, hook, instantiation and connect nodes.

Qualifiers are not contained in this array but they are still part of the generated tree structure. Qualifiers are handled using the record facilities in SAIL. The record structure for qualifiers is as follows.

There are two types of records in the qualifier structure. These are VALUE records and ATTRIBUTE records. VALUE records look like:

The first field contains an attribute value (stored as a string). The second field contains a pointer to the next VALUE record of that attribute.

ATTRIBUTE records look like:

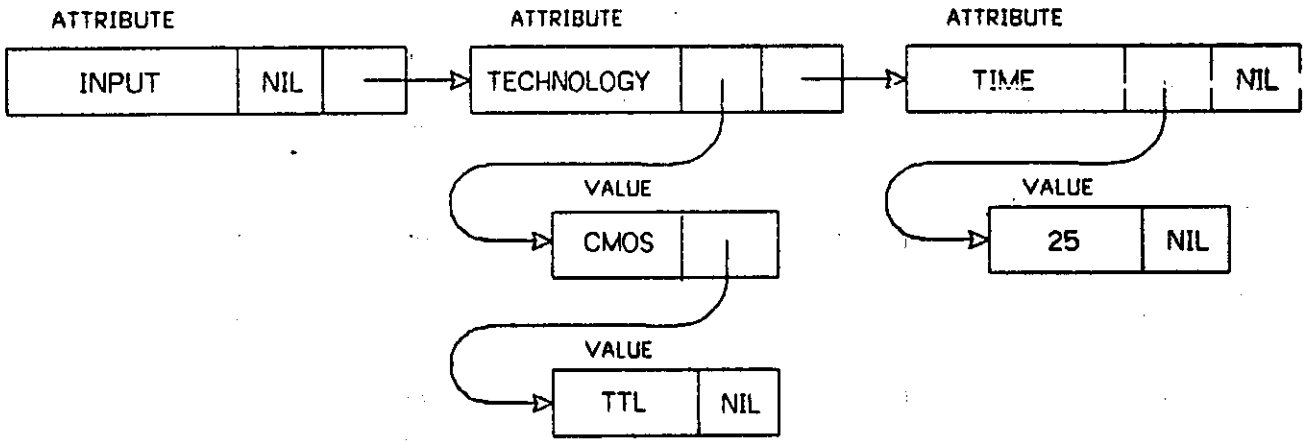


Here the identifier field contains the name of the attribute. The second field contains a pointer to the first VALUE record for that attribute. The third field contains a pointer to the next attribute of the hook.

The best way to show how this record structure looks for an actual qualifier is to show what happens. Suppose that a hook has the following qualifier.

```
(INPUT:TECHNOLOGY:CMOS,TTL;TIME:25)
```

This will generate a record structure that looks like:



This structure will be pointed to in field five of the symbol table for this hook.

Now that the data structures have been explained, a detailed description of what happens wth the three statements of the PMS language can be given.

3.2 Module Declarations

Unlike the other statements in a PMS description, the module declaration statement does not cause any immediate output to the GDB file. This is because attributes of the modules' hooks can be inherited from the components of the module. Therefore the module head portion of the GDB parse tree is output after the whole PMS description has been processed.

The information in the module statement is kept in the symbol table and the record structure described previously. The module declaration

```
MODULE MEM(ADDBUSS<0:"F>{INPUT},DBUS<0:#7>{TRISTATE},R,W<>{INPUT}) :=
```

will generate a tree in the symbol table like the one shown in figure 3-2. Any connections made to a module hook can be checked using the information contained in this tree.

3.3 Instantiation Declarations

Instantiation declarations contain the name of a component in the system and the name of the file which describes the component. The description can be either GDB or PMS, but if it is PMS then the compiler calls itself recursively to generate a GDB tree which is then used.

Once the GDB file has been specified, the file is opened and the information in the header

1	1	2	-999	-999	-999	-999	-999	-999	MEM
2	3	-999	-999	3	1	4	-999	-999	ADDBUS
3	4	0	15	10	16	-999	-999	-999	
4	3	-999	-999	5	2	6	-999	-999	DBUS
5	4	0	7	10	8	-999	-999	-999	
6	3	-999	-999	7	3	-999	-999	-999	R.W
7	4	-1	-1	10	10	-999	-999	-999	

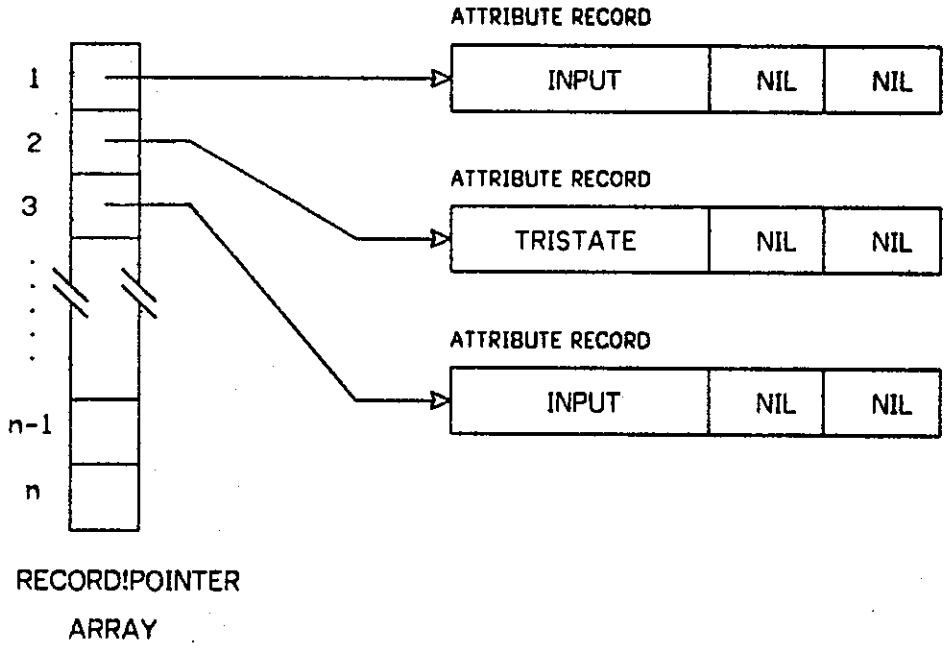


Figure 3-2: Module declaration tree

is processed and loaded in the symbol table. From this information an interface for the component is generated. This interface is what makes possible the connections and also

avoids conflicts in scope with other variables that may have the same name as one of the hooks.

To avoid naming conflicts between similar names appearing as module names, instantiation names, and hook names, the GDB files which are instantiated are slightly transformed before being inserted in the resulting GDB file.

The renaming of hooks is performed by enclosing the body of a GDB file inside a compiler defined section. This section contains the declaration of the module as it appeared in the original GDB file (but with the FC-Set eliminated) and one mapping declaration for each hook in the original FC-Set. Each hook appears on the left hand side of a mapping whose right hand side has the same structure but where the entity name has been replaced with a compiler generated entity name. For instance, assume the following module:

```
X(Y<>) := BEGIN Y = 0 END
```

This module could be instantiated as:

```
M :=  X.GDB                                  (assume the file name and
                                             module name are the same)
```

The renaming that takes place could be expressed as if the original ISP description had been written as:

```
M := BEGIN

    **INTERFACE**        !section header introduced by the PMS compiler
        Y<> := M.Y<>,                    ! mapping replacing the formal
                                        !parameter (FC-Set) declarations
        X  := BEGIN  Y =  0 END          ! X without  formal parameters
       END,
  M.Y<>                              !declaration of the interface carrier
```

The entity Y is available inside the body of X. The entity M.Y is declared outside the context of X. Notice that it is perfectly legal for X to have a local entity named M.Y but this does not present any problem given the Algol scope rules used in ISPS (ie. the M.Y used in the definition of Y is the outside one, not the inside one local to X).

By enclosing the whole section in a body with the name of the instantiation, the formal parameters (Y in the example) are also protected from naming conflicts with formal parameters of other instantiations. Since instance names are unique within a module, the pseudo-formal that the formal is mapped into (M.Y) is also unique.

The mechanism described above is in fact equivalent to an ISPS description whose top level declarations consist of instance declarations and pseudo-formal declarations. Connections between modules are performed through pseudo-formal mappings.

3.4 Connections

Connections are the core of the PMS description. They describe the actual structure of the system. Connections also involve almost all of the error checking done in the system. They have to be checked to see that the hooks being connected have been defined, that they haven't already been connected, that they are the same size, and that their attributes are compatible.

A connection statement in the PMS description can have two different conceptual meanings depending on the connection being made. If two instantiation hooks are being connected then the connection can be viewed as a physical link between two entities. However, a connection statement involving a module hook and an instantiation hook should not be viewed as a physical connection between entities, but rather as a renaming of that entity.

A connection statement defines a mapping between hooks. There are no restrictions in the order in which the connections are specified. Moreover, the left and right hand sides of a connection can be swapped.

The way connections are handled in the GDB output file is similar to the way the interface is made in instantiation declaractions. The two hooks in the connections statement are mapped onto one another so that the two names become synonymous. For instance, the connections statement

```
CONNECT COMP1(D<>) := COMP2(B<>)
```

will cause the entities COMP1.D and COMP2.B to refer to the same thing. This connection coupled with the interface in the component descriptions produces the desired mapping so that the overall connection is made. The equivalent ISPS declaration would look like:

```
COMP1.D<> := COMP2.B<>,
```

When connecting component hooks, the hooks are renamed to componentname.hookname. However, when connecting a module hook, the module hookname is not changed to modulename.hookname. Thus the statement

```
CONNECT EXAMP(A<>) := COMP(B<>)
```

with EXAMP the name of the module is equivalent to:

```
COMP.B<> := A<>
```

This is so the complete GDB tree of the system can be used as a primitive in another PMS description.

When a connection statement is encountered in the PMS file, the information about the left hook is stored in the symbol table. Then the rest of the table is searched to see if the hook exists. If it is found then the hook is marked connected and since it appears on the left hand side of the connect statement it is also marked defined. If the hook has already been connected then a warning in given. If it has already been defined then an error message is given because this will produce a double definition in the GDB file.

The right hook in the connection statement is handled in the same way as the left one except that it is not marked as defined. An exception to this rule is if the lefthand hook is a module hook. Then, effectively, the positions of the two hooks are interchanged, so that the right hand hook will be marked defined. Once the right hand hook has been processed, the attributes of the hooks are checked for compatability, the sizes of the hooks are checked for equality, and then the connection is output to the GDB file.

3.4.1 Attribute Checking

A hook consists of a carrier and a qualifier. The carrier is of the form of an ISPS EHEAD. The qualifier is a list of attribute-value pairs. The attributes specify properties of this hook. Since the set of attributes is open ended, attributes (and their values) are not necessarily understood by the system.

Essentially, qualifiers limit the functionality of a hook. If a hook is specified with no qualifier, then it is a 'super' hook and can be connected to any other hook of the same size. A hook with a qualifier is limited by the values specified.

Attribute checking follows different rules depending on the connection being made. There are two cases for connecting hooks in a PMS description. Case 1 is when hooks from instantiations are connected, and case 2 is when a module hook is being connected to an instantiation hook.

3.4.2 Connecting two instantiation hooks

In this case, if an attribute is shared by both hooks, then the values of that attribute in one hook must be a subset of the values in the other hook. This convention was chosen as

being the most proper criteria for checking a connection. If the two sets of values are completely disjoint the connection is suspect. If the hooks have exactly the same values, then the connection is deemed legitimate. The question arises in what to do for the other conditions. When the two sets of values have some values in common but there is a value in each set that is not found in the other (ie. one set is not a subset of the other) then because there are these inconsistencies this connection is also suspect.

If one of the hooks has a certain attribute and the other hook does not then this will be a legal connection. The second hook has no restrictions with regard to that attribute so there is no incompatability.

3.4.3 Connecting an instantiation hook to a module hook

When connecting a module hook to an instantiation hook the rules are a little different.

In case 1 (above) the attribute value pairs are used only for error checking in making the connection. After the checking has been done, the qualifier is done with and is not included in the GDB output file. However, in case 2 this information is necessary in the GDB output file because the new module could be used as an instantiation in another module.

If a module hook and an instantiation hook have an attribute in common, then the module values have to be a subset of the instantiation values. This is so the person describing the module can specify which values of an attribute are inherited (ie. acquired from the instantiation hook) and also so that the module cannot have attribute values that have not been explicitly stated in an internal instantiation.

If an instantiation hook has an attribute that the module hook does not have, then the module hook will inherit that attribute and its values. This is so a PMS module can be described with the hooks having no qualifiers, but in the final GDB file the hooks will have inherited the qualifiers from the instantiations.

3.4.4 Attributes Known by the System

It was decided that some attributes should not follow the rules given above. For example, the attributes INPUT and OUTPUT should not. If two instantiation hooks are to be connected and one is specified OUTPUT the other should be specified INPUT. On the other hand, if a module hook and an instantiation are to be connected, they must have the same attribute, either INPUT or OUTPUT. The system does not know about these 'special case' attribute pairs unless they are specified by the user. This is done by putting these special attribute pairs in a file, one pair per line, with the two attributes in the pair separated by a space.

The system will prompt the user for the name of the attribute file. These 'special case' attribute pairs will then be handled the same way as INPUT and OUTPUT were in the example above.

Along with attributes specified in the attribute file, the compiler is also 'smart' about the attribute BUS. If a hook is specified with the attribute BUS the compiler will not object if more than one connection is made to that hook (see example on page 25). A user should beware that any hook that appears in more than one connect statement, can only appear on the right hand side of a connect statement once. The compiler will give a warning if this rule is broken because it will result in a double-definition in the gdb file.

3.4.5 How Attribute Checking Is Done

When connecting two instantiation hooks together attribute checking is done for each attribute individually. Each attribute of the first hook is first checked to see if it is a 'special case' attribute. If it is then the second hook's attributes are searched to see if the proper match can be found. If the proper match cannot be found then a warning is given. If the attribute is not 'special case' then the second hook's attributes are searched for a match. If there is a match then the values for that attribute are tested for the subset criteria given above. If the matched attributes values are not subsets a warning is given.

When all of the first hook's attributes have been checked then the second hooks attributes are checked for 'special case' attributes. If one is found then the first hooks attributes are searched for the proper match. A warning is given if it is not found. When all the second hooks attributes have been checked for 'special case' then attribute checking for this type of connect statement is done.

When connecting a module hook to an instantiation hook the module hooks attributes are examined one at a time. An attribute is first checked to see if it is a 'special case' attribute. If it is then the instantiation hooks attributes are checked for the proper match. If no match is found a warning is generated.

If it is not a special case attribute then the instantiation hooks attributes are searched for a match. If there is a match then the values are compared to make sure that the module's values are a subset of the instantiations values. Processing then moves on to the next attribute.

When all the attributes of the module hook have been processed, then the attributes of the instantiation hook are checked one at a time to see if the module hook has the same attribute. If the module hook does not then the attribute (and its values) are added to the module

hook's attributes. This is how attributes are inherited by the module. When all additions have been made then attribute checking for this type of connect statement is done.

3.5 Cleanup

When all the statements in the PMS file have been taken care of then several things have to be done before the output GDB file is complete. First, all of the hooks in the connection statements that weren't defined will have to be now. For example, the connection statement

```
CONNECT COMP1(D<>) := COMP2(B<>)
```

generated the output (ISPS equivalent):

```
COMP1.D<> := COMP2.B<>,
```

This leaves the entity COMP2.B undefined in the GDB file. To define it the GDB equivalent of

```
COMP2.B<>,
```

is output. All of the hooks left undefined by connection statements are declared in this way.

Any hooks that were left unconnected in the PMS description are also undefined. The symbol table is searched for any unconnected hooks, a warning is given to the user and then the hook is declared in the output file in the same way as undefined hooks resulting from connection statements.

Another thing that is added to the GDB file is an entity that will activate all of the components of the system. This is the main entity of the system and is given the same name as the module followed by a ".". It is needed so that if the description of the system is used as a primitive in another system, all of its components will be activated. Thus if a module has three components the equivalent ISPS of the main entity would be:

```
MAIN modulename & "." :=
    BEGIN
        comp1();
        comp2();
        comp3();
    END
```

Once all the undefined hooks have been declared and the main entity has been output, then the output file is closed. The GDB file is now complete except for the module header. Since the header has to go at the head of the GDB file, the file is reopened for both input and

output. Then the module header (with any inherited hooks) is output to the file and the rest of the file is copied after it enclosed in a section called PMS. With that done the GDB file is complete and the program is ended.

4. Using the System

The PMS language developed here is easy to grasp and understand but it requires a knowledge of ISPS. This is because the primitive components in a PMS description have to be written in ISPS and they have to have a certain format.

In order for an ISPS described component to be connected to another component, the hooks have to be included in the FC-Set of the ISPS-declaration. For example, if a component called MEM has three hooks, ADDBUS, DBUS, and R.W, then the first line of the ISPS description should look like:

```
MEM(ADDBUS<0:15>,DBUS<0:7>,R.W<>) := BEGIN . . . END
```

The entities ADDBUS, DBUS, and R.W are the only interface between that component and the outside world.

4.1 The PMS Description

A PMS description consists of a module declaration, instantiations of components, and the connection of hooks. The module declaration has the syntax described earlier and must be the first element of a PMS description.

Instantiations of components may be declared anytime after the module declaration. These declarations can occur only one per line and the declaration must be all on one line. Because the instantiation declaration does not include the names of the hooks of the component, the user has to know what hooks a component has in order to make the proper connections.

Connection statements must also occur one per line and cannot be more than one line long. They may occur anywhere in the description but the hooks being connected have to have already been declared for the connection to make sense.

4.2 Running the Compiler

The compiler for the PMS system can be started by typing:

```
run pms
```

The compiler will prompt the user for the necessary input. Let's use the example given earlier to illustrate the use of the system. In the earlier example we had two components, a processor and a memory. These components were used to build the system shown in Figure 2-2.The PMS description below generates this system.

```
MODULE SYS :=

    PROC := PROCESSOR.GDB
    MEM := MEMORY.GDB

    CONNECT PROC(ADDRESS<0:15>) := MEM(ADDRESS<15:0>)
    CONNECT MEM(DATA<7:0>) := PROC(DATA<0:7>)
    CONNECT MEM(R,W<2:0>) := PROC(R,W<0:2>)
```

The dialogue with the computer to get the description parsed into GDB is shown below.

```
.run pms
Input name of PMS file:comp
Name of attribute file [RET if none]:
Errors will be logged in COMP.ERR

End of SAIL execution
```

4.3 Error Messages

When the compiler finds an error it will give a message that is hopefully self-explanatory. Most error messages will not cause the compiler to abort, but the output GDB file will probably be incorrect. Messages generated because of an incompatability with the attributes do not cause any error in the GDB file. All of the error messages given by the compiler while compiling a PMS file will be logged in an error file with the same name as the PMS file but with the extension ERR.

5. Conclusions

The language developed in this paper is not meant to be a stand alone description language. It is more of an upward extension of ISPS intended to make the description of large systems easier and more flexible. Its development is mainly because the ISPS environment has grown to such an extent that a description language at the level above ISPS is needed and can be used.

The PMS system should prove very useful in describing large modules because a user can partition the module into smaller components so that the ISPS descriptions are at a manageable size. Also a database of components could be developed so that the design of a system would merely involve writing a PMS description of the interconnection of these predefined components.

In the future, an addition that might make the system easier to use would be to make it more interactive. Because the language has no block structure, each statement of a PMS description could be typed directly to the system with any errors or warnings communicated immediately. Another interesting feature would be to add a graphics section that would generate a picture of the described module on a Graphics Display Processor.

Another addition that might make the system a little more elegant would be to have the user declare one of the instantiated modules as the MAIN component. This would give the user control over which components are activated and what order they are activated. As it is now, components that just add structure to the system (ie. no behavioral characteristics) are activated along with components that do have behavioral characteristics. While this activation does not hurt anything it is unnecessary and has no physical meaning.

6. References

[Barbacci, 1977] M. R. Barbacci, G. E. Barnes, R. C. Cattell and D. P. Sieworek, "The ISPS Computer Description Language", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1977.

[Barbacci, 1978] M. R. Barbacci, A. Nagle, "An ISPS Simulator", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1978.

[Bell, 1971] C. G. Bell and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill Book Company, New York, 1971.

[Knudsen, 1973] M. J. Knudsen, "PMSL, An Interactive Language for System-Level Description and Analysis of Computer Structures", PhD. Thesis, Department of Computer Science, Carnegie-Mellon University, 1973.

I. Appendix A: Examples

I.1 A Multiprocessor Example

Assume that a description of a system of two microprocessors sharing a memory is desired. Further assume that ISPS descriptions of a microprocessor and a two-ported memory exist. Figure I-1 shows the ISPS essentials of the descriptions of a microprocessor and a memory.

```
MICRO(ADDRESS<11:0>, DATA<7:0>, DIN<>, DOUT<>) :=
        BEGIN

        (Body describing behavior of microprocessor)

        END


MEMORY(ADD1<1:12>, ADD2<1:12>, DATA1<1:8>, DATA2<1:8>,
    DIN1<>, DIN2<>, DOUT1<>, DOUT2<>) :=
        BEGIN

           (Body describing behavior of memory)

        END
```

Figure I-1: ISPS of Microprocessor and Memory

Note that the microprocessor has address and data lines as well as flags to let the memory know which way data is flowing. The memory has two sets of address and data lines which make up the two ports of the memory. The multiprocessor system will consist of two of the described microprocessors both sharing the described memory.

The PMS description of the multiprocessor is shown in Figure I-2. Note that the described module has no hooks and therefore cannot be used as a primitive in another PMS description.

```
MODULE multi:=

        p1 := micro.gdb
        p2 := micro.gdb
        m := mem.gdb

        Connect p1(address<11:0>) := M(add1<1:12>)
        Connect p1(data<7:0>) := M(data1<1:8>)
        Connect P1(din<>) := M(din1<>)
        Connect P1(dout<>) := M(dout1<>)
        Connect p2(address<11:0>) := M(add2<1:12>)
        Connect p2(data<7:0>) := M(data2<1:8>)
        Connect P2(din<>) := M(din2<>)
        Connect P2(dout<>) := M(dout2<>)
```

Figure 1-2: PMS of a multiprocessor

The ISPS equivalent of the GDB output generated by the PMS compiler is shown below.

```
MULTI :=

  BEGIN
    ** PMS **

    P1 :=
      BEGIN
        ** INTERFACE **
        ADDRESS<11:0> := P1.ADDRESS<11.0>,
        DATA<7:0> := P1.DATA<7:0>,
        DIN<> := P1.DIN<>,
        DOUT<> := P1.DOUT<>,

        MAIN MICRO :=
            BEGIN
                (Body describing behavior of microprocessor)
            END
      END,

    P2 :=
      BEGIN
        ** INTERFACE **
        ADDRESS<11:0> := P2.ADDRESS<11.0>,
        DATA<7:0> := P2.DATA<7:0>,
        DIN<> := P2.DIN<>,
        DOUT<> := P2.DOUT<>,

        MAIN MICRO :=
            BEGIN
                (Body describing behavior of microprocessor)
            END
      END,

    M :=
      BEGIN
        ** INTERFACE **
        ADD1<1:12> := M.ADD1<1:12>,
        ADD2<1:12> := M.ADD2<1:12>,
        DATA1<1:8> := M.DATA1<1:8>,
        DATA2<1:8> := M.DATA2<1:8>,
```

```
        DIN1<> := M.DIN1<>,
        DIN2<> := M.DIN2<>,
        DOUT1<> := M.DOUT1<>,
        DOUT2<> := M.DOUT2<>,

        MAIN MEM :=
            BEGIN
                (Body describing the memory)
            END
      END,

    P1.ADDRESS<11:0> := M.ADD1<1:12>,
    P2.ADDRESS<11:0> := M.ADD2<1:12>,
    P1.DATA<7:0> := M.DATA1<1:8>,
    P2.DATA<7:0> := M.DATA2<1:8>,
    P1.DIN<> := M.DIN1<>,
    P2.DIN<> := M.DIN2<>,
    P1.DOUT<> := M.DOUT1<>,
    P2.DOUT<> := M.DOUT2<>,

    M.ADD1<1:12>,
    M.ADD2<1:12>,
    M.DATA1<1:8>,
    M.DATA2<1:8>,
    M.DIN1<>,
    M.DIN2<>,
    M.DOUT1<>,
    M.DOUT2<>,

    MAIN MICRO,:=
      BEGIN
        P1();
        P2();
        M()
      END

END
```

I.2 A 4K Ram

Suppose we have the ISPS description of a 1K RAM and we would like to build a 4K RAM from them. The first thing we would need is a controller that could take a 12 bit address and then choose the proper 1K RAM and feed it a 10 bit address. Figure I-3 shows a diagram of the memory module. The ISP's of such a controller and a 1K RAM are shown below, as is the PMS descripton of the entire module.

ISPS of 4K RAM Controller

```
cont4k(add<0:11>(input),memadd<0:9>(bus;output),ctl<0:1>(bus;tristate),
data<0:7>(bus;tristate),on1<>,on2<>,on3<>,on0<>) :=
    begin

** Memory.Controller **
```

```
interp :=
    begin
        memadd = add<2:11>next
        decode add<0:1> =>
            begin
                0 := begin
                     on0 = 1
                     end,
                1 := begin
                     on1 = 1
                     end,
                2 := begin
                     on2 = 1
                     end,
                3 := begin
                     on3 = 1
                     end
            end
    end,
Main  Run.cycle :=
    begin
        WAIT(ctl) next
        interp() next
        wait(on0  on1  on2  on3 EQL 0) next
        restart run.cycle
    end
end
```

ISPS of 1K RAM

```
ram1k(add<0:9>(input),data<0:7>(tristate),ctl<0:1>(tristate),on<>) :=
    begin
        ** Memory **

        Mp[0:1023]<0:7>,

        ** Memory.Cycle **

     Mcycle :=
        begin
            decode ctl =>
                   begin
                    1\Read := data = Mp[add],
                    2\Write := Mp[add] = data,
                    [0,3] := No.op()
                   end next
            ctl = 0
        end

        ** Cycle **

Main  Run :=
    begin
        WAIT(on) next
        Mcycle() next
        on = 0 next
        restart run
    end
end
```

PMS Description of 4K RAM

```
module ram4k(add<0:11>,data<0:7>,ctl<0:1>) :=

        controller := cont4k.gdb
        mem0 := ram1k.gdb
        mem1 := ram1k.gdb
        mem2 := ram1k.gdb
        mem3 := ram1k.gdb

connect add<0:11> := controller(add<0:11>)
connect ctl<0:1> := controller(ctl<0:1>)
connect data<0:7> := controller(data<0:7>)
connect mem0(data<0:7>) := controller(data<0:7>)
connect mem1(data<0:7>) := controller(data<0:7>)
connect mem2(data<0:7>) := controller(data<0:7>)
connect mem3(data<0:7>) := controller(data<0:7>)
connect mem0(ctl<0:1>) := controller(ctl<0:1>)
connect mem1(ctl<0:1>) := controller(ctl<0:1>)
connect mem2(ctl<0:1>) := controller(ctl<0:1>)
connect mem3(ctl<0:1>) := controller(ctl<0:1>)
connect mem0(add<0:9>) := controller(memadd<0:9>)
connect mem1(add<0:9>) := controller(memadd<0:9>)
connect mem2(add<0:9>) := controller(memadd<0:9>)
connect mem3(add<0:9>) := controller(memadd<0:9>)
connect controller(on0<>) := mem0(on<>)
connect controller(on1<>) := mem1(on<>)
connect controller(on2<>) := mem2(on<>)
connect controller(on3<>) := mem3(on<>)
```

The duty of the controller is to decode the top two bits of the address, activate the proper memory and wait until the memory transfer is complete before starting again. The RAMs are always waiting to be activated by the controller. When one is, it checks the control lines to see which operation (read/write) is desired and when the transfer is complete, it resets the control lines. The PMS description of the 4K RAM is shown above. This description will produce a GDB that maps all the 1K RAMs data lines onto the modules data lines, maps all the control lines onto the modules control lines, and maps the memories address lines onto the controller's memory address lines.

The description as it is now, will generate a valid GDB tree when run through the PMS compiler, but the compiler will issue warning messages because of the multiple connections to several of the entities (e.g. CONTROLLER(DATA), CONTROLLER(CTL)). Because these lines have multiple connections to them, we will give them the attribute BUS, so the compiler will not issue its warnings. Also we will give the hooks other attributes depending on whether the lines are read, write, or both. The new headers for the descriptions are shown below.

```
RAM1K(ADD<0:9>(INPUT),DATA<0:7>(TRISTATE),CTL<0:1>(TRISTATE),ON<>) :=

CONT4K(ADD<0:11>(INPUT),MEMADD<0:9>(BUS;OUTPUT),CTL<0:1>(BUS;TRISTATE),
                DATA<0:7>(BUS;TRISTATE),ON1<>,ON2<>,ON3<>,ON0<>) :=
```

When the PMS compiler is run using the revised ISPS descriptions, the module hooks will

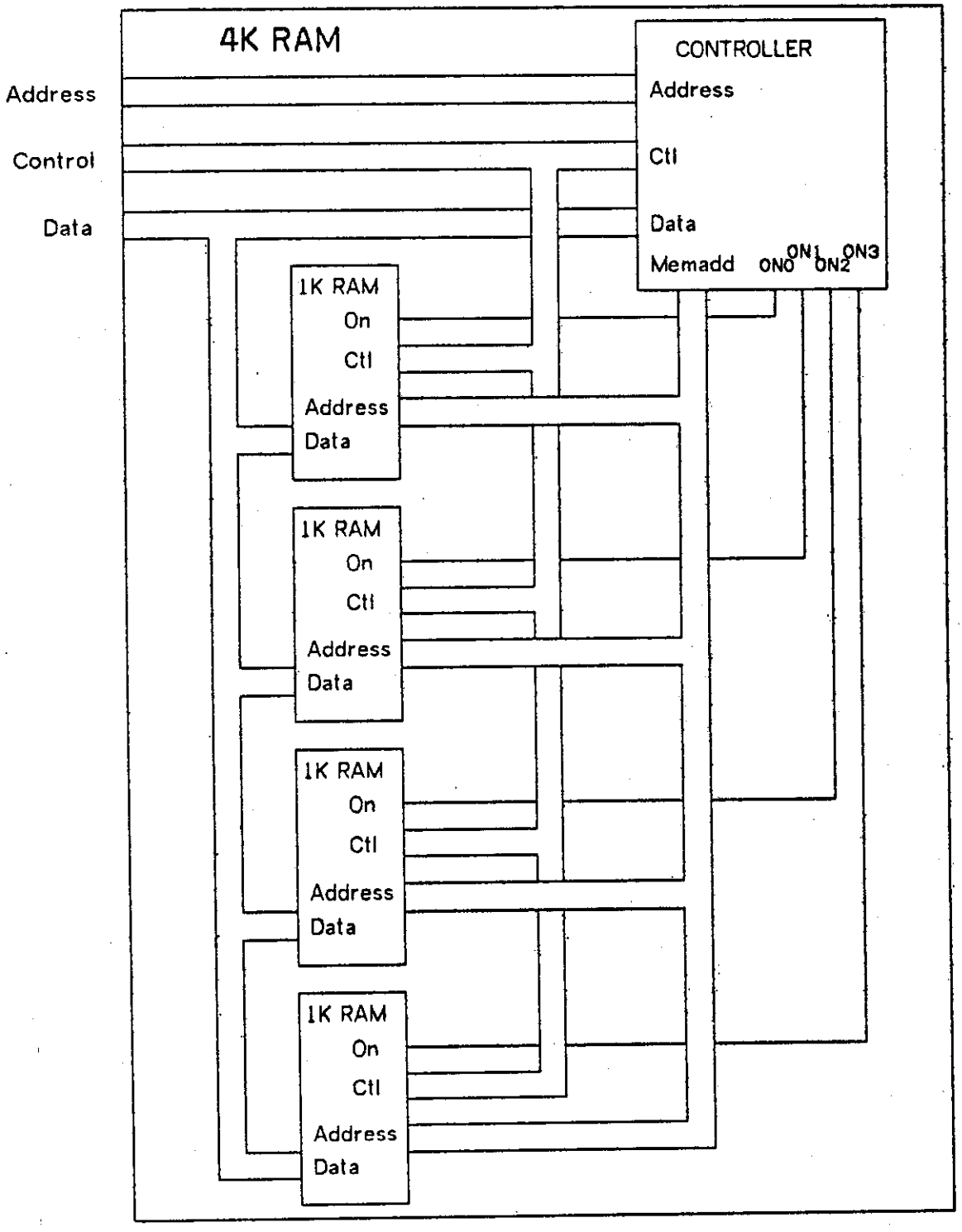


Figure I-3: 4K RAM Module

inherit the attributes from the hooks they were connected to. The equivalent ISPS of the output module header is shown below.

```
RAM4K(ADD<0:11>(INPUT),DATA<0:7>(BUS;TRISTATE),
                CTL<0:1>(BUS;TRISTATE)) :=
```

I.3 A Simulator Example

In this example, a very crude processor (based on the Manchester University Mark-1 Computer) will be connected to the memory described in the previous section. Then the whole module will be simulated on the ISPS simulator [Barbacci, 1976]. The ISPS description of the processor is shown below.

```
! ISPS description of a very limited microprocessor based
! on the Manchester University Mark-1 Computer

MICRO(ADDRESS<0:11>(OUTPUT),DATA<0:7>(TRISTATE),CTL<0:1>(TRISTATE)) :=
    BEGIN

** PROCESSOR.STATE **

IR\INSTRUCTION.REGISTER<0:15>,
    ADD<0:11> := IR<4:15>,
    F<0:3> := IR<0:3>,
PC\PROGRAM.COUNTER<0:11>,
ACC\ACCUMULATOR<0:15>,

** INSTRUCTION.EXECUTION **

MAIN  I.CYCLE :=
    BEGIN
        IR = MR(PC) next
        DECODE F =>
            BEGIN
                0\JMP  := PC = MR(ADD),
                1\JRP: = PC = PC + MR(ADD),
                2\LDN  := ACC = -MR(ADD),
                3\STO  := MW(),
                4:5\SUB  := ACC = ACC - MR(ADD),
                6\CMP  := IF ACC LSS 0 => PC = PC + 2,
                7\STP  := LEAVE I.CYCLE,
                8:15   :=NO.OP()
            END NEXT
        PC = PC +2 NEXT
        RESTART I.CYCLE
    END,

    MR\Memory.Read.Cycle(Arg<0:11>)<0:15>:=
        begin
            address = arg next
            ctl = 1 next
            WAIT(ctl EQL 0) next
            mr<0:7> = data next
            address = arg + 1 next
            ctl = 1 next
            WAIT(ctl EQL 0) next
            mr<8:15> = data
        end,

        MW\Memory.Write:=
```

```
            begin
                ADDRESS = ADD;
                DATA = ACC<0:7> NEXT
                CTL = 2 NEXT
                WAIT(CTL EQL 0) NEXT
                ADDRESS = ADD + 1;
                DATA = ACC<8:15> NEXT
                CTL = 2 NEXT
                WAIT (CTL EQL 0)
              END,
end
```

The processor will use the memory described in the previous section as its primary memory. The PMS description of the entire module is shown below. Note that the instantiated memory module is itself a PMS description. When the PMS compiler processes this statement it will call itself recursively, generate a GDB file of the memory module, and then when it returns use that GDB file in the entire module.

```
module microcomputer :=

        comp := micro.gdb
        mem := ram4k.pms,ram4k.att

        connect mem(data<0:7>) := comp(data<0:7>)
        connect mem(add<0:11>) := comp(address<0:11>)
        connect comp(ctl<0:1>) := mem(ctl<0:1>)
```

A reproduction of a session at the terminal shows how the PMS compiler is run and then shows the described module being simulated on the ISPS simulator. The compiler is given an attribute file so that it can do more detailed checking of the validity of connections being made., The attribute file is shown below.

```
input,output
tristate,tristate
```

The special case attribute pair TRISTATE,TRISTATE will have the compiler check that any hook with the attribute TRISTATE is connected to another hook that has that attribute. The INPUT,OUTPUT attribute pair will work as described earlier.

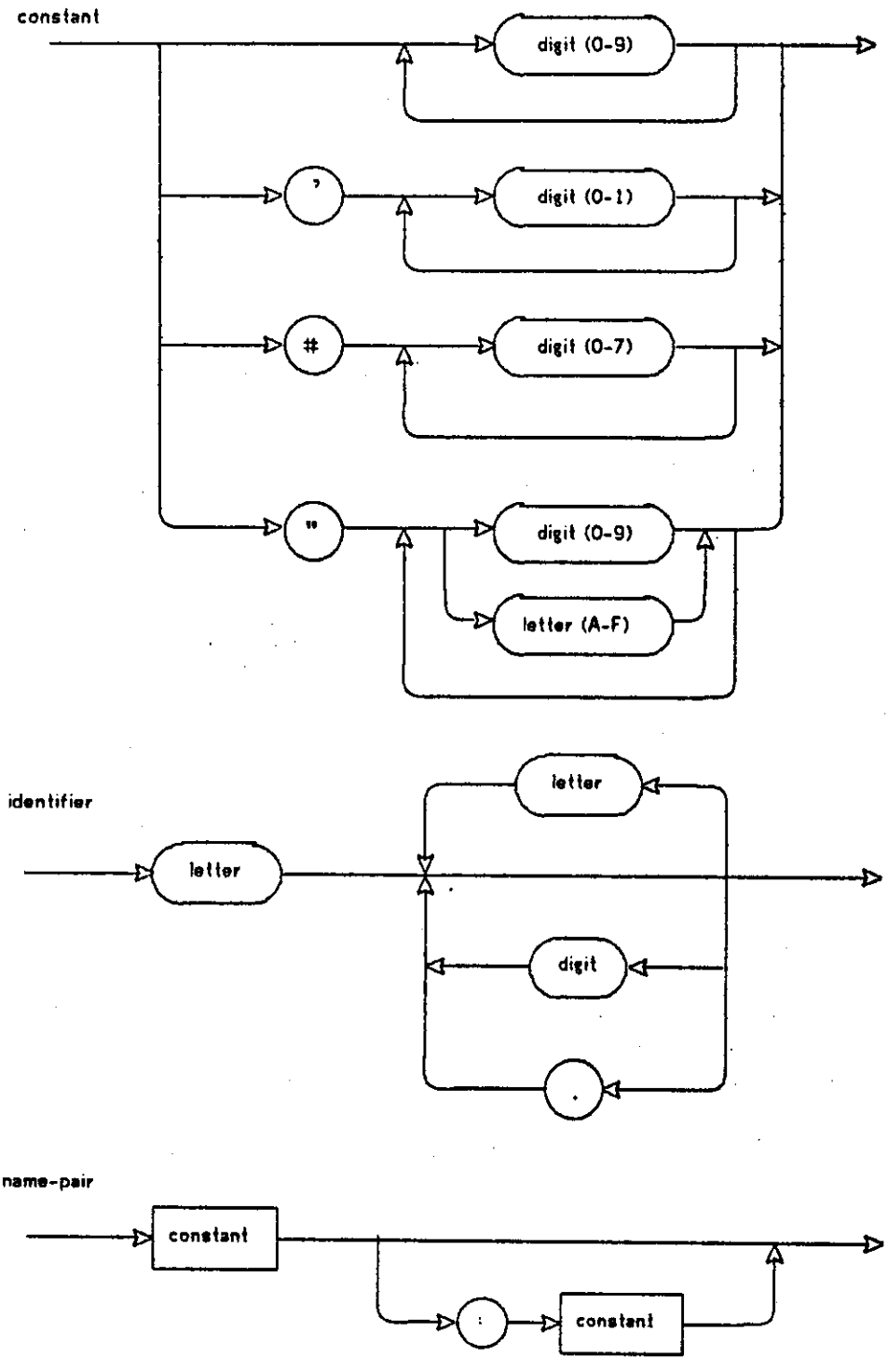
The reproduction of the session follows. The simple program loaded into the simulator will have the processor add four numbers while keeping a running subtotal.

```
.run pms
PMS File:comp
Attribute File [CRLF if none]:comp.att
Errors Will be Logged in COMP.ERR

End of SAIL execution

.ru gdbrtm
GDB to RTM Translator V4B(0)-7
Error Messages Will be Logged On DSK:<filename>.ERR
GDB File:comp
PMS:P;PMS Compiler VA-5;DSK:COMP.PMS[X710BH50];23 Oct 1979;09:06:16;
```

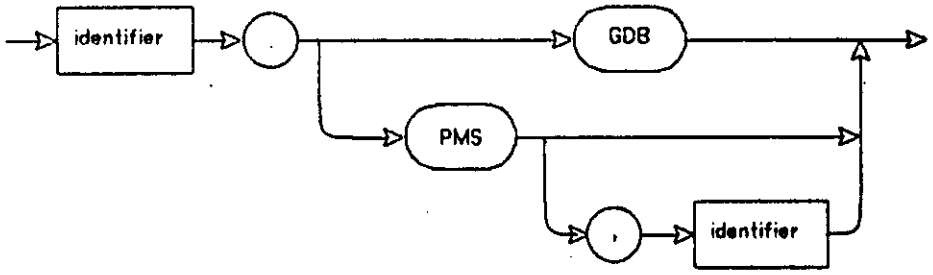
```
MACRO:   RTM
Files deleted:
034RTM.TMP       90

.r link
*comp
*@ispsim
*/ssave test
*/go

EXIT

.run test
ISP SIMULATOR V9.1
PMS: P; PMS Compiler VA-5; DSK: COMP.PMS[X710BH50]; 23 Oct 1979; 09:06:16;

Sequential Simulation? [YES]: n
Type HELP for Help
Type ^C^C to Interrupt Simulation Loops
Latest News: 22 Oct 79
>>echo
>read micro.cmd
>>radix hex
>>s mem0%ram1k%mp[0]=24,10       ! LDN  410
>>s mem0%ram1k%mp[2]=44,12       ! SUB  412
>>s mem0%ram1k%mp[4]=3C,10       ! STO  C10
>>s mem0%ram1k%mp[6]=44,14       ! SUB  414
>>s mem0%ram1k%mp[8]=3C,12       ! STO  C12
>>s mem0%ram1k%mp[A]=44,16       ! SUB  416
>>s mem0%ram1k%mp[C]=3C,14       ! STO  C14
>>s mem0%ram1k%mp[E]=2C,10       ! LDN  C10
>>s mem0%ram1k%mp[10]=3C,10      ! STO  C10
>>s mem0%ram1k%mp[12]=2C,12      ! LDN  C12
>>s mem0%ram1k%mp[14]=3C,12      ! STO  C12
>>s mem0%ram1k%mp[16]=2C,14      ! LDN  C14
>>s mem0%ram1k%mp[18]=3C,14      ! STO  C14
>>s mem0%ram1k%mp[1A]=72,00      ! STOP
>>
>>
>>radix decimal
>>! Load mem1 with the numbers to be added
>>s mem1%ram1k%mp["10]=0,27
>>s mem1%ram1k%mp["12]=0,53
>>s mem1%ram1k%mp["14]=0,14
>>s mem1%ram1k%mp["16]=0,6
>>
>>! Load PC with location of start of program
>>PC=0
>>! Set a break so that simulation will stop when the processor stops
>>abreak comp
>>26 Lines Read
>v mem1%ram1k%mp["10:"17]

MEM1%RAM1K%MP[16]=0
MEM1%RAM1K%MP[17]=27
MEM1%RAM1K%MP[18]=0
MEM1%RAM1K%MP[19]=53
MEM1%RAM1K%MP[20]=0
MEM1%RAM1K%MP[21]=14
MEM1%RAM1K%MP[22]=0
MEM1%RAM1K%MP[23]=6
>start

BREAK Tail of COMP
```

```
 END
*v mem3%ram1k%mp["10:"15]

MEM3%RAM1K%MP[16]=0
MEM3%RAM1K%MP[17]=80          27 + 53 = 80
MEM3%RAM1K%MP[18]=0
MEM3%RAM1K%MP[19]=94          27 + 53 + 14 = 94
MEM3%RAM1K%MP[20]=0
MEM3%RAM1K%MP[21]=100         27 + 53 + 14 + 6 = 100
*exit

Simulation Completed
Run Time(Milliseconds)=1857
RTM OPS EXECUTED=8202
>>exit

EXIT
[2.65 9.21 247 1]
```

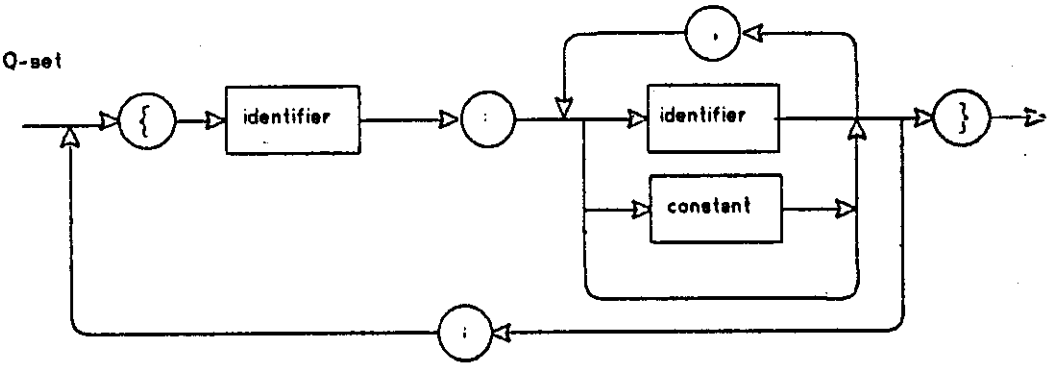
II. Appendix B: Syntax



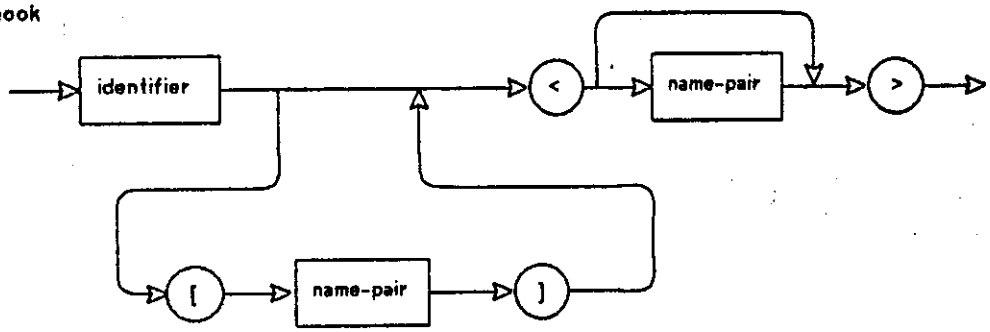
file-name



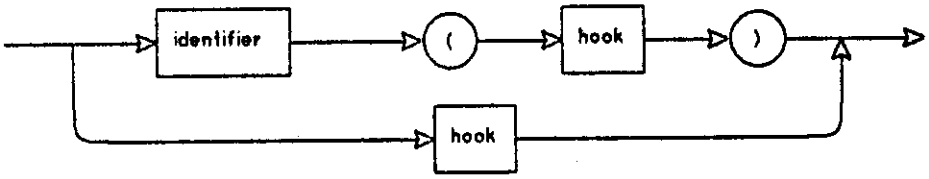
Q-set



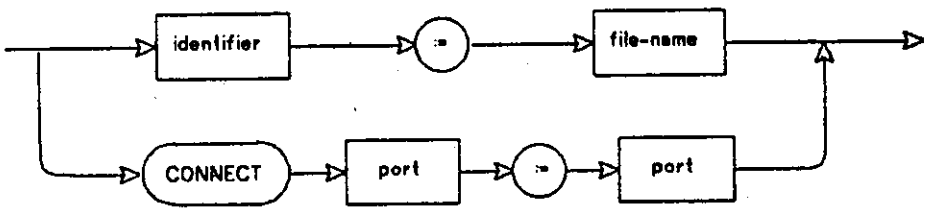
hook



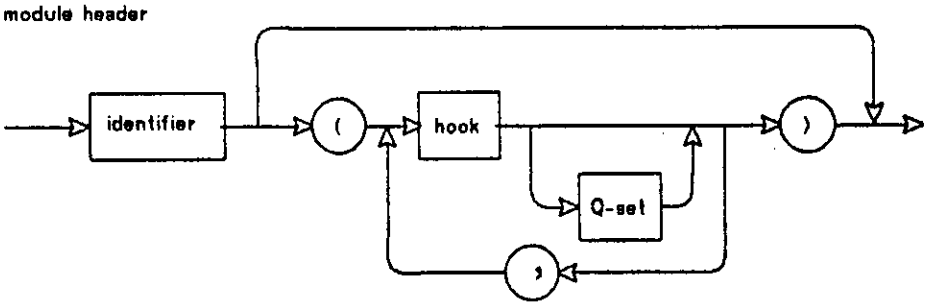
port



statement



module header



PMS description