ISI/SR-78-12
April 1978

ARPA ORDER NO. 2223

12 SC

Multi-Microprocessor Emulation

Annual Report for 1977

Charles Hayden
Peter W. Alfvin
Stephen D. Crocker

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

This research was supported by Rome Air Development Center under Contract DAHC15 72 C 0308.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ISI/SR-78-12
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: Final
4. TITLE (and Subtitle): Multi-Microprocessor Emulation: Annual Report for 1977
5. TYPE OF REPORT & PERIOD COVERED: Annual report for fiscal year 1977 and 197T
6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Charles Hayden, Peter W. Alfvin, Stephen D. Crocker
8. CONTRACT OR GRANT NUMBER(s): DAHC15-72-C-0308, ARPA Order-2223
9. PERFORMING ORGANIZATION NAME AND ADDRESS: USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55971413, 14
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISCA), Griffiss AFB, New York 13441
12. REPORT DATE: April 1978
13. NUMBER OF PAGES: 37
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Rome Air Development Center (ISCA), Griffiss AFB, New York 13441
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public sale and release; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Armand Vito (ISCA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

emulation, microprocessor, multi-microprocessor, SAEF, PRIM, QPRIM

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407 952

20. Abstract

The goal of the Multi-Microprocessor Emulation (MMPE) project is to develop modeling and emulation techniques for assemblies of microprocessors. An extension to an existing computer description language (ISPS) is proposed for representing the architecture of multi-microprocessor systems, and the results of some preliminary studies on the design of an emulation facility are described. This effort will eventually lead to a high-speed emulation facility based on the QPRIM system. The emulation facility is one component of the SAEF under development at RADC.

Introduction and Goals

Microprocessors and microprocessor networks are becoming more and more important as parts of new computer system designs. But these components, with their limited software, peripherals, and control panel functions, are more difficult to program and debug than more conventional computers. For economic reasons it may never be desirable to equip some systems with facilities to allow for interactive debugging of programs. Yet the problems encountered in the development of programs for these systems are often greater than for conventional systems. They often involve real-time tasks and the interfacing of unique hardware, and they are often programmed in assembly or machine language.

The goal of the Multi-Microprocessor Emulation project (MMPE) is to alleviate these problems by creating an environment in which system building and programming can be carried out conveniently. The system to be used is modelled using an appropriate description language. Then its behavior is emulated on a general purpose computer. The environment may be much richer and more controlled, enabling more convenient debugging and performance measurement. This can be a considerable aid in developing and testing software for systems (such as microprocessors) which have little program development support hardware.

There are a number of parts to the MMPE project aimed at creating such an environment. First, and most important, is an appropriate computer system description language. Then a translator must be developed to enable the described system to be simulated. Some means must be provided to build systems of hardware components that are separately described. Then the emulation environment, including a emulation supervisor, debugger, and measurement tools, must be created. This report details the

progress made in the design and experimental implementation of the description language, configuration system, and emulation supervisor. The debugging and measurement tools are not discussed further in this phase of the project.

PRIM and Extensions

The Programming Research Instrument (PRIM) system is an emulation system currently in use. It allows interactive execution and debugging of code for emulated computer systems by a number of users. It uses a microprogrammed computer, the MLP-900, for executing emulated computer instructions. This system is tightly coupled to a PDP-10 to support interactive access, provide debugging and environment support. Currently the emulation must be coded in MLP-900 microcode for each particular machine to be emulated. Furthermore, assorted support software, such as debugger tables, must be constructed for each computer system on the PDP-10.

The MMPE project extends the PRIM system in three ways. First, the system to be emulated is not restricted to be a single CPU, as required by the present PRIM system. A number of identical, similar, or different processors are allowed, with any degree of interconnection. I/O processors and devices may be described as concurrent processors, and the whole system may be emulated. In contrast, I/O processors in the present PRIM system are supported, at least in part, by code on the host PDP-10.

Second, the system to be emualted is described in a high level language, and this description is translated automatically to code to do the emulation. Such a high level is much easier to produce and understand than the microcode that does the emulation. It may even be the definition of how the computer in question behaves, and it can be used to produce a debugger that can refer to machine registers and operations symbolically, in the same terms as the description.

The third extension to PRIM involves the collection of measurements of the simulated architecture. No means are now provided to collect such data, but it would be much

easier in the proposed system because the emulation itself is produced by a translator and the measurement tools could be inserted automatically where appropriate.

Accomplishments

A number of actual working programs have been produced to test the ideas behind the design of the MMPE system. These embody the ideas developed on the proper high level description language, the mechanism for assembling a system from components, and the form of run time support. These programs can be viewed as a concrete form of a design and as a prototype system. They have been used to simulate a particular computer to the level where it could successfully load and execute the manufacturer's diagnostics.

The programs that have been developed translate from a high level language, ISPS, suitably extended to describe multiple concurrent processors, to a PDP-10 based emulation. They do not produce microcode because, for the purposes of demonstration, PDP-10 code is easier to deal with. The PDP-10 code presents the same fundamental issues, but they are more conveniently solved than if we were dealing directly with microcode. This emulation does not currently operate under PRIM, and does not have sophisticated debugging facilities. This can be added to the present system without a great deal of trouble, but would require some modifications to the PRIM system.

A translation program produces a PDP-10 relocatable emulation module from a parsed ISPS program. The parsing of this program is done by a program resident at and maintained by Carnegie-Mellon University (CMU). The translator itself resides at ISI and is written in Interlisp. Its speed of translation is comparable with CMU's parser. The code it produces is actually in the language BLISS, which is translated to PDP-10 code by a standard compiler. This makes translation somewhat easier and produces more efficient emulations, since the BLISS compiler performs many optimizations.

The second program, a linker, has several functions. It takes separately described modules, translated individually by the parser and translator, and combines them into a system. It can be used to aggregate related modules to make composite modules. It can combine modules into a complete emulation system. And it computes storage allocation and prepares a symbol table for use by the debugger.

The third set of programs produced are the runtime support functions. These are, of necessity, oriented toward the PDP-10 implementation. But they embody all the functions that will be required in any other emulation system. This set includes an event driven scheduler, synchronization and timekeeping functions, and a set of predefined ISPS functions. These functions, together with debugging and measurement support functions, would comprise the run time support of any eventual emulation system.

All the programs that have been produced are experimental. They have provided a concrete embodiment of the design decisions and have served their purpose. The next phase of the project may involve scrapping these programs and creating new ones. But it is felt that the translator, in particular, could be easily modified so as to produce code for any reasonable microcoded machine. Major parts of the translator depend only on the ISPS language and not at all on the form of the generated code. The linker and support programs, on the other hand, would probably be useful only for their algorithms. None of these programs required a major programming effort, and all of them could be more easily produced the second time around.

Description Language

The description language employed in the specification of the computer system is the most crucial factor affecting the success of the MMPE project. The language, first of all, must permit and encourage clear and precise description of the action of a computer at the relevant level of detail. For the purposes of computer system emulation, this level of detail is the machine language instruction level. Most importantly, then, the language should make the description of instruction processing clear. In addition to this, facilities are needed to tie separate instruction processors together into a system. Thus, some mechanisms for modular description, specification of concurrency, data sharing between modules, synchronization, and timing must be included.

The computer description language ISPS, developed at CMU, was chosen as the description language for this project. It is well developed and in widespread use for describing computer instruction execution. Since these are the most important requirements, and ISPS fulfills them well, ISPS was augmented to meet the further requirements. ISPS is described in a series of documents [1]. The reader should have some familiarity with standard ISPS as described in these documents before continuing. Other languages, similar in philosophy to ISPS, could have been used instead (e.g. SMITE [?]). The major advantage of ISPS in this stage of the project was that a parser was already written and available at CMU, saving us some work. The use of ISPS should be reevaluated before implementing a final system.

MODULARITY

The largest deficiency of ISPS is its lack of provision for separately translated modules. All descriptions are self contained and refer to nothing external. The goal of this project is to be able to describe parts of a computer system separately, such as a

CPU, an I/O device, etc., and then at a later time to be able to combine them into an entire system without modifying the description of the modules themselves. This goal requires that any external connections that a module may have are not bound until it is assembled into a system and that it may be specialized and replicated so that several identical or nearly identical units may be specified with complete generality.

The mechanism proposed to achieve modularity allows the outermost entity declaration to have parameters and provides a way to create a copy of a module and bind these parameters at that time. The skeleton of a module description in the extended language would appear:

```
module.name  (external.data.list):=

        begin

        section

        section

        **Interpretation.process**

        body of module

        end
```

The external.data.list is a list of variable names which will be bound by the creator of the module. These variables are shared between this module, its creator, and any other module the creator may give access to these variables. This module may, in turn, create others and pass to them some of these externals and/or its own local variables.

The body of the module is the statement that is executed after the module is created. This statement might loop forever, in the case of an instruction processor, or it might be null if this module exists only to create others and provide shared data. If the body of a module terminates, no more statements in the module will be executed, but its variables will exist and may be in use by other modules. No special action is taken upon termination of a module body, and other modules may detect this occurrence only through a module's actions (or inaction) upon variables they may share.

CREATION

Creation of modules is specified by the create statement in the extended language. The create statement is not executable but is more like a declaration. The named modules will be created when the module in which the create statement lies is created. Each create statement specifies that a new copy of the named module be created and connected in the specified way. The create statements occur in a section of their own, named **external.connections**. This section may contain only create statements. For example,

```
**external.connections**

local.name:=create module.name (params)
```

The local.name is a name given to the module being created. It is used by the debugger, etc., to refer to the particular instance of the created module, in case there are several instances of the same module. All local names used within one module must be distinct. The module.name is the name of the module being created. This is the name used as the

outermost entity name in the definition. The params is a list of variables which will be bound to the externals of the created module. The two lists must agree in length and dimensionality. The params list may contain constants, local variables, or variables external to the module. Constants are used to specialize created modules. This is similar to jumper or switch-selectable options on actual hardware modules -- something that may be selected and then left constant thereafter.

Variable parameters (variables passed at creation time) serve as communication paths between modules. The flexibility of passing down local or external variables means that a wide range of interconnection schemes may be described quite naturally.

SYNCHRONIZATION

All modules are understood to be operating concurrently and asynchronously with all other modules. The creation mechanism ensures that all modules in a system will be created before any of the module bodies are executed. The modules have several mechanisms at their disposal to synchronize their actions.

First, any variables that they share may be used freely to do explicit synchronization. A module may test a shared variable and take an action based upon the value. The language does not enforce any restriction on the use of shared variables. Thus timing-dependent actions are possible, just as they are in the typical hardware system.

The semaphore operations signal and wait are provided to facilitate explicit safe synchronization. These implement the general semaphore operations and are used in the present version of extended ISPS in the form of procedure calls

```
wait(sem)

signal(sem)
```

The semantics of the semaphore is the usual. It consists of a counter, which records the excess of signal operations over wait operations. If a signal is made, the counter is increased by one, and if a wait is made it is decreased by one. If the counter is zero, then a wait operation will cause the delay of the process attempting the wait until a signal arrives. Any number of processes can be waiting for a signal. They are served in the order in which they made their wait request in this implementation.

The semaphore variable should be a variable reserved for use as a semaphore, and should not be manipulated, tested, etc., in any other way. It should be declared without explicit bit structures, i.e.,

```
sem<>,                    ! a semaphore.
```

Semaphores used in this way enable safe and efficient synchronization. The wait and signal routines are implemented without busy waiting and are simulated very efficiently.

TIMING

Another approach to synchronization relies on explicit knowledge of timing. Processes may use synchronization methods based upon the knowledge of each others timing characteristics. Timing in extended ISPS is specified by a statement similar to a procedure call:

```
delay (time)
```

The argument time may be either a variable or a constant; in either case the value determines the delay experienced within the statement. Time is a global concept. It advances within each module as delay statements are encountered. A module which executes a semaphore wait statement passes time also: processing resumes at the same time that the corresponding signal was issued. These two statements are the only ones that may cause the passage of time. For consistency, all other statements are treated as if they do not cause the passage of time -- that is they are executed instantaneously. This provision blurs the distinction in ISPS between sequential (next) and concurrent (;) statement separators. For this reason they are treated identically as sequential separators in the extended ISPS.

The units of measurement of time are defined entirely by the definition of the system. The emulation scheduler executes statements from the module with the earliest value of time. If this module executes a wait which blocks or a delay statement making it no longer the earliest, then statements from another module will be executed next. Statements from two modules with the same value of time may be executed concurrently or arbitrarily interleaved by the scheduler. If this is not desired, the body of a procedure can be made indivisible by use of the qualifier {indivisible}. If a delay or wait statement occurs within an indivisible procedure, the procedure is divided at that point. But still the statements will be executed as a whole between successive delay, wait, or signal statements.

Description Language Comments

This section describes some of the considerations behind the language extensions proposed in the previous section. These are classified in several categories: timing, synchronization, and interconnection. This section assumes knowledge of the constructs actually proposed and describes other possibilities in terms of the ones actually chosen.

TIMING STATEMENT

The delay statement was invented independently at ISI and at CMU, in essentially the same form. It is a fairly obvious way to express the fact that a path is to take a fixed amount of time, but it is nevertheless fraught with problems.

The first of the problems comes when one tries to resolve the questions of parallel paths. How should delay statements in parallel paths be handled? One could imagine that a parallel action takes as much time as the longest action within it, or some other convention. The problems with arbitrarily nested parallel actions that have delay statements become complex because the parallel branches cannot be simulated sequentially, but must really be simulated in parallel.

Something else happens when all time delays are spelled out specifically with delay statements--all statements between delay statements become instantaneous. The reality of spread-out delays is modeled by lumped delays. This is consistent with our bias toward a description from the machine language programmer's point of view, but it may not be sufficient. For instance, one can, if needed, increase the accuracy of the timing model by inserting increasingly smaller delays throughout the description. But once one reaches the statement level this cannot be continued, and all statements are thus carried out instantaneously.

A final observation should be made, and that is that programmers often consider detailed instruction timing irrelevant. It may depend on the computer model, the nature of the attached memory, the nature of other simultaneous occurrences, and so forth. Specifying it at all in the machine description allows emulations to have a concrete timing to model. But some cases the timing cannot be specified accurately, and the use of the delay statement leads only to solutions that have predetermined, repeatable timing in all cases.

SYNCHRONIZING STATEMENTS

The proposed method of synchronization of the semaphore operations wait and signal. A number of things should be said about the use of these operations.

First, signal and wait operations are not themselves hardware primitives. So why are they included? One reason is that the real hardware primitives which control concurrent operation all use a form of busy waiting. In essence all computer hardware is made up of concurrently executing gates, modules, etc., and the sequential seeming nature of a computer system is designed into it. Yet we normally understand the operation of a computer as a sequence of steps and our description and modules all make use of the fact that computers all operate by performing some kind of steps one after the other. This fundamental assumption is violated in a direct description of concurrent synchronization devices. There are two consequences of this. First, it is often far from clear what a synchronization device is actually doing when it is described in ISPS. This suggests that some higher level abstraction out of which clear synchronization devices could be built would be desirable. Secondly, direct emulation of non-sequential components can be inefficient since this typically involves nonproductive looping until conditions are met.

Higher-level primitives (such as semaphores), on the other hand, are quite efficient. It should be mentioned that other ISPS primitives, such as the procedure call, are not themselves hardware primitives. The control structure corresponding to real hardware is more likely to be the unstructured go to, but the advantages of describing a system in more structured terms are well known in that case.

Semaphores are not the only synchronization primitives available. The construct perhaps closest to the operation of hardware is a statement that causes a delay until a given boolean condition is true, in effect, a short form of a while loop, continually testing the boolean. This is characteristic of the operation of computer hardware, and it expresses clearly the conditions existing when the system is allowed to proceed. The major objection to this construct is, again, that emulation is inefficient and the design relatively unstructured. Race conditions are allowed to occur if several conditions are satisfied simultaneously, and they immediately act so as to make the conditions false again.

The next step in abstraction away from this is to support a named set of ***events*** and to provide event variables and ***await*** statements to manipulate them. The power of these statements is somewhat greater than that of the semaphore, and they are somewhat more complex and inefficient in implementation. Nevertheless, they represent a great advance because they are no longer representations of hardware operations and require a higher level understanding of the synchronization by the description writer. They could have served as well as semaphores for the purposes of this project.

Another construct, one more primitive than the semaphore, is Wirth's signal operation, as used in the Modula language. It resembles a semaphore with no memory. That is, if no process is waiting, then a signal has no effect. If processes are waiting, then a signal starts one of them. Signals could have been used as the basis for synchronization

in the extended language.

Finally we come to semaphores themselves. They are perhaps the simplest synchronization tool out of which a variety of other synchronization routines may be built. There are simpler primitives, but it is often difficult to build the many kinds of synchronization situations from them. Rather than provide a variety of hardware-oriented special primitives, these can be built and distributed with the systems that use them. The provision of indivisible routines aids in the constructions of these.

The objection that semaphores are too powerful a tool, because they cannot be implemented in hardware is not to be slighted. However, a global understanding of the intent of a system is required both for understanding and efficient emulation, and the notation should force this to be conveyed rather than hidden. It is felt that semaphores accomplish this better than the simpler tools like conditional wait.

Finally it should be noted that hardware systems, unlike software systems, can be and often are designed in such a way that time dependent operation is possible. While this is often undesirable for software, it may be essential for hardware. Thus, much of the modern work on safe concurrent programming does not apply, because our solutions rarely fit this mold. The use of semaphores certainly does not constrain any of the configurations we might like to describe. Furthermore, there are no inhibitions on the arbitrary use of shared variables, and the user may program any form of sharing or synchronization he desires using these variables.

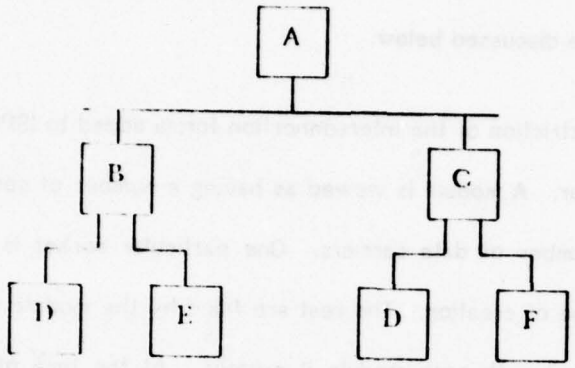
INTERCONNECTION FORMS

The one create statement embodies a number of decisions. These decisions and some of the alternatives are discussed below.

The most obvious restriction of the interconnection forms added to ISPS is that each module has only one creator. A module is viewed as having a number of sockets, each of which is made up of a number of data carriers. One particular socket is filled by the module's creator at the time of creation. The rest are filled by the module itself creating others and filling one socket with each module it creates. At the time of creation the creator may connect the data carriers of the socket to constants, local registers, and to data carriers of its creator. But, there is no direct way for a module to know who its creator is or how the creator has connected their shared socket.

This scheme supports hierarchical network structures in a natural way. Systems that are not hierarchical can be represented in this way, but not necessarily naturally. The major restriction preventing this is that data connections may not be specified to existing modules: data connections always imply the creation of new modules. And a module must always get its external data from its creator. Several networks will be given as examples to clarify these restrictions.

A hierarchical system is diagrammed:

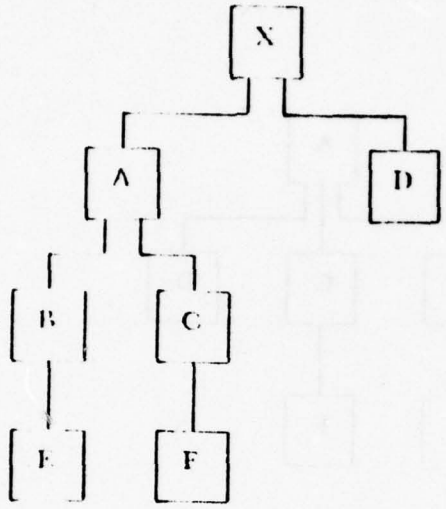


This is expressed naturally in extended ISPS. Module A creates B and C, which create in turn D and E and D and F. Note that they create separate instances of D.

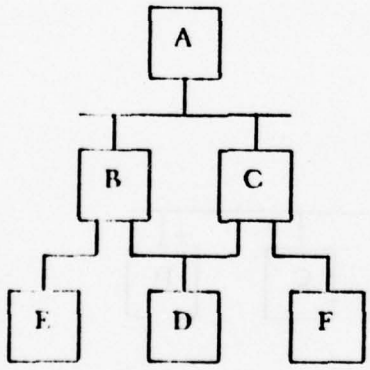
```
Module A                Module B                Module C

A:=                     B(bus):=                C(bus):=

begin                   begin                   begin

.                       .                       .

.                       .                       .

create B (bus)          create D(-)             create D(-)

create C (bus)          create E(-)             create F(-)

.                       .                       .

.                       .                       .

end                     end                     end
```

This configuration has been proposed as an interconnection strategy for microprocessor networks, and it also appears as a portion of the I/O description of the Nova minicomputer.

If the same instance of module D had been required, the configuration would be

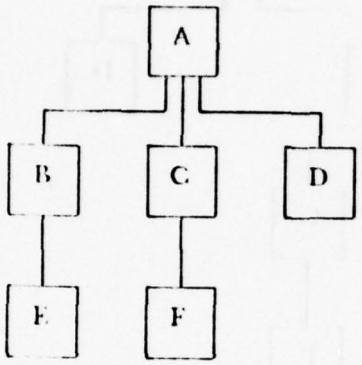


This is not hierarchical and is therefore not described as naturally. The diagram of the modules required is:

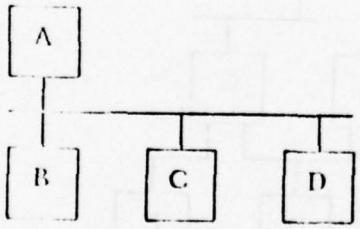


The module X has no actions associated with it, but serves to hold shared data. In the previous diagram, module D may access local data from A,B, or C. All of this shared

data is gathered into X and made external to the other modules. Note that X could be eliminated by putting D below A, on the same level as B and C.



A bus system may be diagrammed

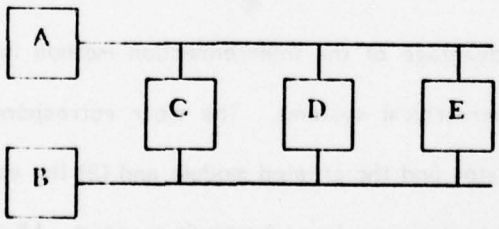


The module A creates the others:

```
A:=
begin
  .
  .
  .
create B(bus)
create C(bus)
create D(bus)
  .
  .
  .
end
```

The module A might be just a data module, representing a list of the configuration description of the computer system.

A dual bus presents more problems. It is diagrammed:



To describe this, both busses must be described in one module. This might look like

```
X:=
begin
        .
        .
create A(bus1)
create B(bus2)
create C(bus1, bus2)
create D(bus1, bus2)
create E(bus1, bus2)
        .
        .
end
```

These examples point up the description strategy that must be followed: all shared data in a module must be provided by the creator and can originate only in one of its ancestors.

The shortcomings of this approach in terms of unnatural representation of some nonhierarchical forms have been discussed. The next section describes the advantages of this system and some alternative methods of specifying interconnections.

The main advantage of the interconnection method in use is its simplicity and its naturalness for hierarchical systems. The clear correspondence between (1) the data objects in the creator and the created module and (2) the correspondence with what one expects based upon programming experience is good. Minimal use is made of external module names, and these names need not be passed around or stored in variables. The structure of the entire computer system is fixed at linkage time, so run-time creation is not allowed. This is realistic in terms of actual hardware and is an aid to analysis.

The present linkage system does allow packaging of several modules into one composite module with the same characteristics as a single module. That is, it is created and has shared data specified by one module, and it creates a number of other modules. These composite modules may be thought of as single modules, a concept useful in cases where one device is made up of a number of concurrent processors. These may be described as a number of modules and then combined by the linker to yield one composite that may be included in a larger system as a unit.

This system also avoids some of the problems associated having systems with multiple instances of identical modules. Each instance of a module is given a unique name at the time it is created. This name identifies it to the debugger, etc. The name is not used elsewhere in the description, however, simplifying its treatment. No distinction needs to be made between referring to an existing module and creating a new one, because there is no way to refer to one without creating it. The interface description explicitly names all the shared data, and a description of its behavior should be sufficient to determine all possible interaction effects.

One unsolved problem concerns creation of similar modules. Modules which behave exactly the same but create different modules should be capable of being specified once only. This would allow sharing of code, etc., in an emulation, and it would make it explicit that the two modules did, indeed, behave the same. Presently this is not possible. One solution would allow module names as parameters and allow a higher module to supply them. This is analogous, in programming languages, to passing procedure names as parameters. The disadvantages of this approach were judged serious enough that it was not proposed.

This problem leads to the following effect: all or most of the actions of the system being described are in the lowest level modules, that is, in modules which create no others. All modules above these serve only to define storage and provide interconnection paths. This effect is another way in which the language chosen determines the description strategy.

A more general interconnection strategy was considered. This one separated the creation of a module from the act of supplying data connections to it. A module would be created by one other module, but this would supply no data connections, just a globally available name by which the module would be known. A module would declare and name a set of sockets, each consisting of a list of data items. These are external and are attached to actual data items by external means. Finally a module would include a set of connections to other modules, naming the global module name and the socket name as well as the list of actual data items. A module showing one possible syntax for this follows.

```
M: -
begin
external socket q1, q3, q5;
external module X3
.
.
socket P1 (-)
socket P2 (-)
.
.
create A named A1
create A named A2;
```

```
create B named B1;
.
.
connect q1 of A1 to (-)
connect q3 of A2 to (-)
connect q5 of X3 to (-)
.
.
end
```

This method allows much freer specification of interconnections in non-hierarchical systems. But it does require use of the given name of a module to identify the object of the connection. It still does not solve the problem of duplicated modules creating others, for in this case the module created by each instance will have the same global name. Thus, the same strategy as before, of putting all the code of a system in the lowest level modules, will tend to be used. And if it is, then the interconnection generality introduced will tend to go unused.

These considerations, along with the feeling that it was undesirable to require the use of the global module names to describe the system itself, led to the rejection of this approach in favor of the former one.

Another approach, not seriously considered, was the use of an entirely different language for interconnections. One such approach would be to have external sockets named and declared in an ISPS module. A separate program could create, and name a set of modules and connect named ports in each to each other. It may or may not be allowed to reserve local data. It might be allowed to itself name sockets, giving again a module as

the result of partial binding.

Essentially, this proposal enforces the rule alluded to before--that all code be at the bottom level. The interconnection descriptions would be ISPS descriptions of the previous, generalized variety, without action sections. In this form, ISPS modules would not be able to create others, thus assuring they will be on the bottom level. It was thought that a single integrated language was advantageous and that inventing a second language would add nothing to the expressive power. Thus the separate-languages approach was rejected.

Program Descriptions

Two prototype programs, a code generator (GDBXBLI) and a linker (LINKER), were developed as part of the MMPE effort. The operation of these two programs and their associated restrictions are discussed in the following paragraphs.

GDBXBLI: Multiprocessor emulation ISPS code generator

OPERATION

The code generator is called from the executive by typing ***TRAN***. The program announces itself by typing ***ISPS code generator and linker***. To start the code generator type ***TRAN***. The program asks the user for the name of a file where the GDB source may be found by typing ***enter filename:*** The filename can be typed, using the standard filename-recognition conventions. Confirmation will be requested. The extension defaults to GDB and the version to the highest existing version. If the file cannot be found, the request will be repeated.

When an acceptable file has been entered, the code generator will start into operation. It begins by checking the GDB type as given in the heading. The code generator accepts only the B version. This is produced by the compiler switch /-O. If the type is wrong, the code generator will type a message and halt; otherwise it types the header and starts translating. Any errors that are encountered are typed on the terminal as they are encountered. The errors are of two types -- those representing unimplemented constructs and those representing invalid GDB trees. Normally the format of the GDB tree should be guaranteed good by the program that generated it, so errors should be rare occurrences. The error mechanism is that of LISP and is not usable by one unfamiliar with the translator itself.

After the program is translated the resulting BLISS program is written to a file. The root file name is the same as that of the GDB file, but the extension is BLI. If there were no errors detected, a background job is started to compile the BLISS program. Next the generator writes out linkage information on a file named the same as the GDB file but with extension INF. Finally the CPU time used is typed, the program types ***Done***, and control returns to the keyboard. The user may at this point translate other programs, initiate the linker, or exit (by typing ***QUIT***).

RESTRICTIONS

The code generator does not generate code for the full range of ISPS constructs described in the reference manual. Some restrictions have been made to save implementation effort, and their implementation is being currently considered. Some restrictions are the result of shortcomings in ISPS or restrictions on our extensions made by the parser.

Array bounds: The generator expects array lower bounds are zero. This restriction is made to save implementation effort.

Bitscripts: Bitscripts must be in one of three forms

1. <> - taken to mean unspecified length, not one bit
2. <n> - same as <n,n>
3. <n1,n2> - n1, must be greater than or equal to n2

Furthermore bitscripts are limited to be less than or equal to 35 by the implementation of the simulator. These restrictions are made to save implementation effort. The interpretation of the null (<>) bitscript is due to the need to express a carrier of undefined length, which is not available in standard ISPS. It is anticipated that this

restriction will be lifted later.

E-FS-maps: not implemented. E-FS-maps result from use of the concatenation operator in declaration statements. This restriction is made to reduce implementation effort. The ultimate worth of this feature is doubtful.

Array mappings: Mappings changing the word structure of a variable are not allowed. Mapping of simple variables onto particular array elements is allowed. This restriction is made to save implementation effort. The worth of this feature is significant, but it considerably complicates the translation process.

RESTART and RESUME: The control actions RESTART and RESUME are not implemented. This restriction is made to save implementation effort. The worth of these features is unknown, but they considerably complicate the run-time environment.

DECODE: Name pairs and name lists are not supported for use in numbered lists for DECODE statements. Numbered lists must be numbered with single numbers, nothing, or OTHERWISE. This restriction is made to save implementation effort. It should be relaxed in later versions of the translator.

The LEAVE Statement: A LEAVE statement may be used only to exit from a labelled block, not from procedure. This restriction is made to save implementation effort. It should be relaxed in later versions of the translator.

Arithmetic: Only the default two's complement arithmetic is used. Qualifiers specifying other arithmetic are ignored. This restriction is made to save implementation effort. It is not clear whether the effort involved in implementing the various arithmetic forms is worth the energy expended.

Multiple assignments: The value resulting from an assignment is that of the left order, not the right. Thus truncation, extension, etc., are done assignment-by-assignment rather than broadcasting the right side value. This change in the semantics is made because it is felt to be is more natural than that specified officially.

Left Side Assignment Expressions: Expressions used on the left side of assignments may use only the concatenation operator. Bitscripts may be applied only to variables and not to values produced by concatenation. This restriction is made to save implementation effort. The official specification does not make clear if this is legal or not.

Procedure Call: All parameters to procedures are passed by reference except expressions and constants, which are passed by value. This restriction is made because it is not feasible to pass structured variables by value, nor is it wise to pass constants by reference.

Constants: Constants containing the question mark are not supported. Insufficient information was available to allow implementation of this feature of the language.

Implementation restrictions limit the maximum size of all values, declared and resulting in any expression, to 35 bits. This restriction results from the precision of the PDP-10, and any relaxation could result in considerable efficiency penalty.

There are a number of restrictions in the language extensions described earlier, resulting from the use of the standard parser. These are described below.

Module body: A module may not have a body, only a list of declarations. These declarations include register and procedure definitions. The effect of a module body can be simulated by means of a qualifier, {main}, attached to a procedure. There must be only one such procedure so qualified, and it may have no parameters or value.

Create statement: The keyword create is not allowed in this statement. Thus the statement:

local.name:=create module.name (params)

would appear as

local.name:= module.name (params)

Furthermore, only variables, and not constants, are allowed to appear as parameters in the parameter list. Constants must be assigned to variables, and those variables passed in the present version.

Concurrency: All procedures are given the indivisible quality by default. This means that they execute until they request a time delay or wait operation. No process should loop without passing time through a delay statement or waiting.

LINKER: Multimodule System Linkage Program

OPERATION

The linker is called from the executive by typing ***TRAN***. The program announces itself by typing ***ISPS code generator and linker***. To start the linker type ***LINK***. The program begins by asking for an input file. This file should be an INF file produced by the code generator. The file name may be typed using the normal TENEX completion conventions. The extension defaults to INF. The program reads the file and types out the name of the module(s) (which need not be the name of the file) that it finds in the file. The program keeps track of all modules required by those it has already seen.

After processing the INF file it prints ***Need modules: module list*** The module list is a list of the module names that are needed but not yet found. Then it asks ***More?*** To link more modules, type ***Y*** or <return> or <escape>. If this is done the program again asks for an input file. The filename now defaults to the first module name listed, so if the file name is the same as the module name, the entire file name can be defaulted.

If no more modules are needed or if the user responds ***N*** to the ***More?*** question, then the program requests an output file for the link information from the user. This file, given the default extension INF, can be used in subsequent sessions as a single entity containing the entire configuration specified so far.

If the configuration is complete (no further modules needed) then a symbol table is produced and a background job started to create a save file that can be run to do the emulation. When the linker is finished it types ***Done***.

ERRORS

There are several errors that may occur. These are:

1. Module not found: module name

   Trying to link a module the linker does not know about. Usually due to bad INF file format.

2. Actual and formal parameter lists do not match: formal parameter list, actual parameter list.

   The lists of shared storage found in the creation statement and the created module head must match in length. This error is given if they do not.

3. Two modules called the same name. module name

   This indicates that the local name of two separately created modules is identical.

4. Module connects to itself recursively. module name

   A module has, through a chain of creations, tried to recreate itself. This is illegal.

5. Cannot find definition. variable name

   Cannot find a definition for a variable, usually indicates invalid INF file format.

In addition there is a warning

6. Warning: Not main module.

The first module given is not a main module, which means it has parameters supplied by a creator. When linking a complete system, always start with the main module, or else the linker will assume this is a subsystem and issue this warning.

PROGRAM DESCRIPTION

The linker program has two phases. First it gathers up INF files onto a master list MODS. This is basically a list containing each INF file. The first entry is assumed to be

the main module. The function GATHER links one module in. GATHER takes as input a list telling the module name to be linked in, what the creator has named it, a list of all module names already created, and a parameter list. GATHER finds the required module on MODS, gives it its unique name, links the parameter lists, and enters any create statements it found on a list of things to do. The entries are in the form of items that can be given to GATHER as arguments.

The first phase collects INF files and calls GATHER whenever it has the modules named in the create requests. If the user asks to stop or no more modules are unknown, then a complete system has been built.

This triggers the second phase, in which gathering starts over again from scratch. The reason for this is that the order of gathering must correspond to the order of initialization of modules for the symbol table to be correct. Only by gathering a complete description is this possible.

A new INF file consisting of the MODS list is made, and if the system is complete a symbol table is printed and a batch job is submitted to do the LINK10 command.

References

[1] Barbacci, Mario R. et al, *The ISPS Computer Description Language*, CMU, August 1977.

[2] TRW, *SMITE Installation Analysis, TRW Technical Report 30417-6002-RU-00, August 1977.*