Norman Ramsey

ISPS: A Retrospective View

Alice C. Parker and Donald E. Thomas
Electrical Engineering Department
Carnegie-Mellon University

Stephen Crocker
Information Sciences Institute
University of Southern California

Roderic G. G. Cattell
Computer Science Laboratory
Xerox Palo Alto Research Center

1. ABSTRACT

This paper describes the authors' experiences with applications of ISPS. The main application areas discussed include description, simulations and emulations, hardware synthesis, software synthesis and verification. The paper concentrates on semantic inadequacies of the language; desirable features of the language for the applications discussed have been described elsewhere. Finally, a proposed language extension, the PROCESS construct, is presented.

2. FOREWORD

ISPS has provided a vehicle for some exciting research - simulations of new architectures[15], automatic design of digital circuits [2], [19], [11], [18], [16], automated emulator generation [12], compiler-compiler research [8], verification of machine language programs [9], verification of microcode[1], and automated generation of diagnostics [14]. In turn, each of these projects has exercised the ISPS language and supporting software to an extent not possible with paper designs of HDL's. Furthermore, the use of the language and software in an undergraduate classroom setting has greatly enhanced our understanding - as well as highlighted our misunderstandings - of the ISPS semantics.

The goal of this paper is to provide the reader with a better understanding of the requirements of a language for the applications stated above. First, however, we should lay the groundwork by discussing the intent and philosophy underlying the ISPS language. Then, we will enumerate the applications of ISPS the authors have been involved in, and describe the experience with ISPS for these applications. Finally, we draw on our experience with these applications to discuss the inadequacies of the language in its present form.

THE ISPS PHILOSOPHY

In the past, confusion about the intent and philosophy of ISPS has arisen, and some criticisms of ISPS have resulted from the attempted use of ISPS for reasons other than that for which it was intended. Thus, before we begin the critique of the language, it is appropriate to discuss the history and goals of ISPS.

The original intent of ISP as described by Bell and Newell [7] was to aid in their exposition of instruction sets. It was used to produce a behavioral description of the functioning of a CPU, as seen by the machine language user. ISPL, the first software supported version of ISP, began under this definition, but almost immediately was used for descriptions of hardware other than CPUs. ISPS, the newest version of ISP, was invented with the intent of using it for many applications of machine description languages. These applications include description of the *behavior* of arbitrary register-transfer level circuits: processors, display processors, elevator controllers, video terminals, and disk controllers. Description of the physical *hardware* itself, however, has not been one of the goals of ISPS and attempts to apply ISPS in this manner have resulted in a set of pathological examples which illustrate the inadequacies of ISPS for this purpose. When ISPS was designed, simulation, machine description for a compiler- compiler, and hardware synthesis were three of the applications considered. ISPS was designed to be application independent through the use *qualifiers*, a simple syntactic extension to the language allowing insertion of application-specific information in curly brackets(i.e. { and }). The semantics of ISPS were to be loosely defined, so that the applications essentially provided the definition. Preliminary studies were done to determine the problems with ISPL, in order to correct these problems *a priori* with ISPS. ISPS was then defined, and feedback was obtained from a group of users via a series of workshops. Finally, ISPS was frozen syntactically.

The ability to describe behavior rather than structure of a digital system is the overriding advantage of ISP languages. It has allowed the measurement of instruction set performance independently of implementation details; it has provided the functional specification necessary for digital designs to be automatically produced and optimized; it has obscured unnecessary detail so that the problem of microcode generation was simplified; it has provided structured control flow, thus allowing the possibilities of symbolic execution and verification.

The software to support ISPS at Carnegie-Mellon consists primarily of a parser, code generator and interpreter [3] and [4]. The parser accepts ISPS descriptions and outputs parse trees known as *"GDB trees"* (GDB stands for "Global Data Base"). It was intended that the applications programs would consume this intermediate form, thus decreasing the work to build a new application and providing some force for unifying the semantics of the language. Although we intend to confine our comments to the ISPS language and not the underlying software, the design and use of the GDB as a common intermediate form proved to be an excellent choice: widely ranging applications have been able to use this common description language representation quite easily. However, the applications have also pinpointed several semantic inadequacies of the language. It is the purpose of this paper to relate some of our experiences and suggest semantics to strengthen the base language. By its nature, this paper will necessarily focus on the shortcomings and difficulties we have seen. However, it would not be worth relating these criticisms if the language did not already provide very useful capabilities. In fact, no existing machine description language has been used for so wide a range of applications.

[1] research in progress

CH1436-5/79/0000-0021$00.75 © 1979 IEEE

3. ISPS/ISPL APPLICATIONS

The application areas we will discuss here can be subdivided into the problem domains of:

DESCRIPTION
- digital systems
- control engines
- interfaces, I/O and buses

SIMULATION / EMULATION
- digital systems
- control engines
- interfaces, I/O, buses

HARDWARE SYNTHESIS
- digital systems
- interfaces

SOFTWARE SYNTHESIS
- compiler generation
- assembler generation

VERIFICATION
- machine language
- microcode

Description of digital systems has been done for simulation and automated design purposes, and will be discussed under these topics. Description of control engines was undertaken in order to determine the capabilities of ISP for these purposes, for pedagogical purposes, and for architectural evaluations. Since ISP is a procedural language, the control function and hence the structure is implicit in a conventional ISP description. Any assumptions about the nature of the control must be made on the basis of the control flow of the ISP description. Description of the control engine *explicitly* allowed more accurate simulation measurements and therefore more accurate architecture evaluation to be made. Simulating the control also allowed students to write and debug microcode easily, providing a wide range of microprogrammable machines in a microprogramming course.

Interface and bus descriptive capabilities were first motivated by the desire to automate interface design, and the need to simulate multiple function-unit systems. To this end, several abortive attempts were made to describe the UNIBUS in ISPL. Finally, an existing I/O language, GLIDE[1] [17, 21], was examined to determine missing ISPL capabilities. A compiler was constructed to translate this language to ISPL in order to formalize the relationship between the two languages[13]. The process of producing this compiler affected the development of ISPS and made explicit the limitations of ISP family languages for description of device-level interfaces (i.e., GLIDE semantics for which there were no corresponding ISPL semantics),some of which are still valid with regard to ISPS.

Digital system description and simulation using ISPL and later ISPS formed the core of the evaluation procedure for the Military Computer Family effort[5]. In addition, the same ISPL and ISPS simulators provided a teaching tool in the undergraduate Computer Structures course at Carnegie-Mellon. Course students simulated a PDP-8-like processor, and then simulated the modified processor with added I/O capabilities communicating with another processor or external device.

[1] GLIDE has since been renamed SLIDE, for structured GLIDE

Fast emulation of an architecture can be obtained by generating microcode directly from the machine description, rather than simulating the architecture through interpretation of an intermediate representation of the description. The generation of MLP-900 microcode [12] from ISPS allows the automatic creation of an emulation of any system described in ISPS.

The synthesis of digital systems using a formal machine description began at Carnegie-Mellon with Darringer[10], continued with Barbacci[6] and is now a large project involving many students and faculty. The latest synthesis program designs data paths from an ISPS description, and provides feedback on the qualities and capabilities of the language [11],[16].

Software synthesis research has focused on several areas: generation of assemblers from a derivative of ISP [22], the attempted use of ISPS to describe a machine for input to a compiler-compiler, and the use of ISPS to generate assertions. The use of ISPS in the compiler-compiler work was not investigated completely due to lack of time, but the requirements for such an input description were given [8]. Oakley [14] performed symbolic execution of ISPS descriptions in order to generate a non-procedural "assertion description" suitable as input to programs that require knowledge of a target machine's instruction set (for example, the compiler-compiler).

ISPS is used in the verification research [9] to generate *state deltas*, a form for representing segments of computation. A state delta is a first-order predicate which takes a precondition, a post condition, a modification list and an environment list. These state deltas are then used in a proof system.

Verification of the FTSC microprogram [20] was done by describing both the host and target machines in ISPS. The verification system accepts these descriptions along with a listing of the microcode and a set of proof commands.

4. THE ISPS EXPERIENCE

Our criticisms of the language could fall into several categories: semantics, syntax, and support software. Each of these in itself is important. We will emphasize, however, issues dealing with semantics because without these the other issues are moot points.

SEMANTIC INADEQUACIES

In some applications, ISP is viewed as a general digital system specification language. In other cases, it is viewed as a notation for specifying just the instruction set for the CPU of a machine. Considering that the language has these two main bodies of users (the general digital system designers and instruction set processor specifiers), we can argue the inadequcies from each point of view.

The digital system designers will find that the language is inadequate primarily in the specification of concurrency, timing and interconnection of processors.

- A simple "execute the following in parallel" construct is provided (";"); this really means "order independence" and not "at the same time". Nothing is said about the relative speeds of execution, the granularity of operations, or what happens if one of the actions fails to complete normally. It is clear that

  A <- 1 ; B <- 2

  can be executed without ambiguity, but also that it may not be different from

repres
abstra

```
A <- 1 next B <- 2
```

The code

```
A <- B ; B <- A
```

does not produce a well-defined result. Under some interpretations, it defines a register swap; in others, it is equivalent to

```
A <- B
```

and in still others the result is completely unspecified. A compiler could detect the independence of the two actions and allow parallel computation if the facilities permit; syntactic forcing of the concurrency is only useful where interference is a possibility. However, it is precisely in the case of interference that the definition of the language is silent.

- High level abstractions are not provided for signalling between processors, processes, or functional units. As a consequence, modelling the interaction between concurrent machines, e.g. a CPU and an I/O controller, requires the introduction of shared variables which are set, reset and tested by the communicating processes. As certain I/O controllers and other functional units become hardware primitives the description of systems containing these functional units along with their interaction is necessary. Protocols to implement the intended communication must be invented by the writer of the ISPS description. In some cases, these protocols may correspond to their actual implementation, but in many cases, the designer is forced to overspecify in their design and obscure his intentions. The language should allow him to functionally specify the interactions. In fact, the designer cannot even specify the names of inputs or outputs which are not registers or storage locations, except by means of the qualifiers ({ }), which are not part of the formal language. Thus description and simulation of interfaces and interconnections become difficult and in some cases impossible.

- No facilities are provided in the base language for specifying the length of time that operations take. This may not be known if the description is to be input to a synthesis program; then the timings might refer to required time intervals. Timing is essential for simulation and emulation fidelity, however.

- In the area of interconnection of processors, there is no way to parameterize a module and call for its instantiation in a system. In addition to needing to introduce multiple copies of modules, it is also necessary to parameterize their descriptions so that each use can be specialized. Simulation of large systems is therefore inconvenient.

From the point of view of ISPS as an instruction set representation, the chief weaknesses are (1) the lack of abstractions to cover certain operations common in hardware systems, and (2) the need for a constrained structure to the description. The latter problem can be remedied through the use of qualifiers identifying sections of the description (the address computations, the processor state, the instruction interpretation, etc.); this structure is explicitly *not* desirable in ISPS for general digital system design. On the other hand the former problem, concerning abstract operations, is shared by general system designers as well. For example:

- No operator exists for transferring a block of memory into another block, or extracting a a field from a variable position in a register. As a consequence, it is necessary to describe such instructions indirectly, with loops or shifts and masking. The detection of the higher-level functions expressed in terms of these more involved descriptions is difficult for both software and hardware synthesis programs. There are some hardware structures which are now considered to be hardware primitives. These include FIFO buffers and LIFO stacks, associative memories, and large AND-OR arrays of logic (PLAs). Descriptions of the functions these provide with ISPS produce lengthy code; many I/O operations such as code conversion and buffering require these structures.

- No facilities are provided for describing floating point operations. Long sequences of operations are required to describe floating point add and multiply, leaving the reader to depend upon the mnemonics and comments for help in understanding. Similar but lesser difficulties attend the description of character and integer-oriented operations. A compiler-compiler requires higher-level abstractions.

- A compiler-compiler must have knowledge of the addressing computations available on the machine. In order to generate good code and simplify processing, it is essential to separate the description of the address computations from the description of the instructions themselves. This requires, in addition to the syntactic section separation mentioned previously, an abstraction to package an access mechanism as a single identifier. The fetch and store mechanisms would then be invoked when a fetch or store using that identifier is encountered. This allows the actual instructions to be expressed simply, e.g. Dest_Dest+Source for an addition on the PDP-11.

- There is no way to specify the meaning of declared variables or code segments except via qualifiers. (Again, this specification is not a part of the formal language.) For example, code generation programs need to know about the program counter, primary memory and main instruction cycle. For I/O emulations, the microcode generator needs to know about the interrupt line(s). Finally, compiler generators need to know the instruction formats.

- There are some ISPS constructs which produce unwieldy simulations, code or hardware. One of

these is the arbitrary mapping facility.[1] The hardware required to support such a mapping facility could be costly and inefficient, and simulations would also be slow. Microcode generated to support such a mapping of one memory size onto another is lengthy. Another construct which can cause difficulty if misused is the ability to declare fields of a register to be treated differently. If the fields are really functionally different, then they should be treated so in the description. While it is possible to "discover" this from the ISPS description, to do so would require quite a bit of processing overhead.

Under pressure from users with these different views, compromises have been made in the language design. The mechanisms provided for describing concurrency, timing and interconnection of processors are too weak to support the general digital systems designer. But, those needed specify more than the instruction set designer may want. The operations and data types present, while being adequate for general systems designers, have not kept pace with the requirements to describe some of the now "standard" instructions (e.g. floating point) of computers.

SEMANTIC FLAWS

We turn now to the language as it is, ignoring what it might be. As defined, there are some internal inconsistencies and ambiguities. These stem primarily from the lack of a clear, precise, formal semantic definition of the language. To address this point, a preliminary attempt has been completed at providing a formal semantic definition of ISPS, using the techniques of denotational semantics. For this purpose, Information Sciences Institute has developed a different parse tree form of ISPS, called AMDL [1]. For example:

Within an expression, the order of activation is not specified. Since calls to procedures are permitted within expressions and since procedures may have arbitrary side-effects, the order of evaluation is crucial. These matters are not application dependent and therefore should be a part of the formal language definition.

Instruction Set Processor Specifications

Space limitations do not allow a thorough analysis here of ISPS from the point of view of an application like a compiler-compiler. However, a relatively complete specification of requirements can be found in [8] and [14].

Basically, the required portions of a machine description for a compiler-compiler would be:

- A description of the set of registers and memories available, including their size and an indication of their use in some cases (e.g., the primary memory, from which instructions are fetched, must be distinguished).

- A description of the instruction fields and formats.

- Instruction descriptions. These can be found in current ISPS descriptions within a large case statement, decoding instruction fields. The remainder of the processor operation must essentially be ignored and assumed to be of a standard form.

- A description of the data types and formats, separate from the instruction descriptions (e.g., a floating point plus operator, "+F", might be defined).

- A description of the operand access processes, separate from the instruction descriptions.

Several extensions are needed to the syntax and semantics of ISPS to allow a description to be given in this form.

5. PROCESS LEVEL CONSTRUCTS, AN EXAMPLE

One of the main strengths of ISPS is in its description of single *Process* entities. By a process we mean a single control environment or state machine. This control environment, however, may interact in an asynchronous fashion with other control environments and thus the need for synchronization of the different machines becomes necessary. This synchronization is a standard operation in hardware (and software) and is instantiated, for example, in the data-ready and transmitter-ready signals of a UART. As hardware systems are built out of higher level functional modules, the need to describe the interconnection and synchronization of the units becomes apparent.

Software systems have long seen the need for such mechanisms. Real time processing applications with interrupt procedures that service peripheral (asynchronous) devices, require synchronization between the producer of data, say the main program, and the consumer of the data, say the device. Catastrophic effects can occur if the proper mutual exclusions and signalling conventions are not used to maintain data integrity. We do not suggest that ISPS should be able to describe real time processing and operating systems. However, we do want to draw the analogy between the Process concept in hardware and software.

Here are two examples:

1) The buffer memory of a video terminal. One process or state machine (the consumer) responds to the video sync signals and accesses the buffer memory for a line of characters. Another process responds to the communications port and moves data from the port to the buffer memory, possibly with some processing.

2) The buffer queue in a disk drive. The producer responds during a read to a continuous stream of bits from the disk head, gathers a word, and puts it in the queue. The consumer takes words from the queue, requests a DMA, and writes the words into memory.

In each of the above cases, there are two processes sharing both data and control information. Control signals (synchronization) are needed to enforce the integrity of the data in the queue. In software, signals are set up to indicate the condition of the queue (e.g. queue.full, queue.empty) and one of the processes would be given exclusive access to the shared data and control.

Presently in ISPS, separate processes can be specified, and "started" asynchronously with busy-wait loops. However, it is not possible to suspend or terminate these processes external to the processes themselves. The simulator does have facilities for this due to an attempt to use ISPS for interface description, and to use the simulator for I/O simulations. We feel though that it is time that such capabilities be allowed in the description. Any sort of priority logic must be explicitly described in the ISPS

[1] The mapping facility allows registers and memories to be "mapped onto" previously declared storage. Thus, a 16 bit/word memory can be accessed in ISPS either by the word or byte.

language. The leave, restart and resume operators do provide methods for control over process termination within the process itself. However, these, along with the busy-wait initiation procedures, require each process to understand the overall priority structure each is embedded in. We propose that constructs to appropriately synchronize and control these situations be added to the base language.

Another example of a multi-process system is appropriate at this point to illustrate the need for new process level semantics. We provide this example to motivate the semantics; the syntax used here has been chosen for convenience. Take, for instance, the Intel 8086 which has an instruction and PC-relative prefetch unit, as shown in the figure. To accurately and appropriately describe the function of this unit and its interaction with the basic processor we need more than the present ISPS language. (This is also true of the previous examples).

It may be argued that functionally describing a prefetch unit suggests an implementation of an otherwise equivalent architecture. The depth of description however is necessary for the synthesis applications because a synthesis algorithm cannot be expected to automatically insert a new function of this type. Snow [18] showed how a single fetch-execute cycle could be placed in parallel but did not address the possibility of multiple prefetches.

The prefetch unit shares the program counter with the instruction execution unit (IEX) and uses it to fetch ahead for increased performance. The IEX contains all of the processor registers and logic for executing instructions. When instructions or data are fetched using the PC, the prefetch unit is requested for the information. All other operand fetching and storing is done directly with the bus arbiter and memory.

Several sets of information must be arbited for in this system: the main memory, since access is requested by the IEX and prefetch; and the prefetch buffer which is accessed by the IEX and prefetch. However, we will only consider the IEX and Prefetch interactions in this example.

Let us consider the prefetch unit which is itself a functional unit or process.

```
process prefetch := (   !notation
                        !indicates a
                        !process
        Q[words]<bits>,
        head<3:0>,
        tail<3:0>,
        INIT load()     !this tells us
                        !what gets
                        !started !

load := (
        head_tail_0 next        !make
                                !queue
                                !empty
        repeat begin
                wait q.not.full next
                qput()_bus.arbiter(pc)
        end
),      !end load
                )
```

A few words of explanation are needed here. The INIT command specifies where the process starts (or is restarted). The procedure load once init'ed is always running and if the queue is not full it will try to fill it by accessing memory and calling the qput() procedure.

The qput() and qget() procedures control access to a shared memory. Because they deal with shared information they must be placed in a mutual excluson block to indicate that both will not be working at once.

```
begin critical          !critical section

qput()<>:= (
        tail_tail+1 next
        Q[tail]_qput next
        signal queue.not.empty next
                        !possibly wake
                        !up the IEX
        if <not full> => signal q.not.full
                        !possibly wake
                        !up load
),
qget()<> := (
                <access done here>
),
end     !end critical section
```

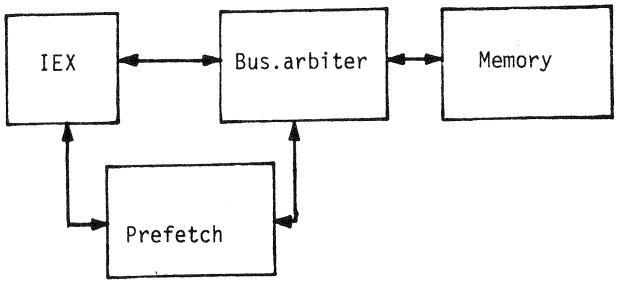
A new construct within the procedure is the SIGNAL construct which is used to provide an indication that some other entity will be waking up to perform an action. This could also be described by setting a flag to one and having the other entity busy-wait for the flag. However, such a description is not obvious in that it is not clear whether the flag is data (as in an overflow bit) or control (as in a data-ready bit). It also suggests an implementation.

The other part of the example is the IEX.

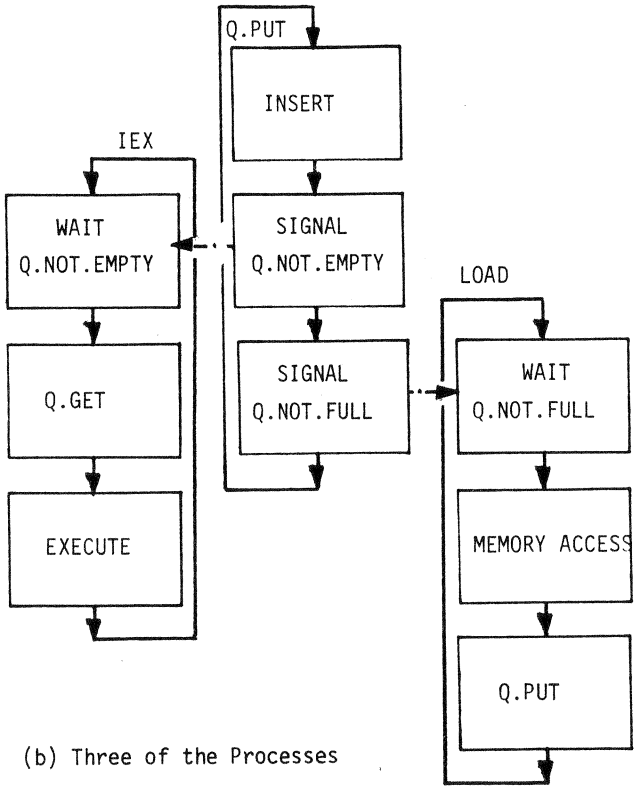
```
Process IEX :=(
!many declarations and procedures

if pc.access =>begin
        wait q.not.empty next
        op _ qget()
end
...
if jump.instruction => begin
        pc_new.pc next
        init prefetch
end
), !end IEX
```

The above process describes how the IEX unit may access memory. If a memory access is not using the PC, then the access is made directly through the bus arbiter. If the access is a PC access, then the system waits for a value to be available in the Q (q.not.empty) and then the qget() procedure is called. The wait is done outside of qget because allowing a wait in a critical section allows for the possibility of a deadlock. The last part indicates what happens when a jump instruction occurs: the queue is flushed by the reinitialization of the prefetch process.

The control structure of the system (shown in the figure) is now apparent: the waits and signals correspond and indicate in no uncertain terms a synchronization of two entities operating asynchronously with respect to each other. To be complete, functional assertion and release of the signals must be possible. Note that this example assumes



(a) Hardware Organization of the Example



(b) Three of the Processes

Figure 1. Example Illustrating Multiple Processes.

that when a wait responds to a signal, the signal becomes released. This is not always the case and thus there may be other interpretations to consider in implementation of such a construct.

The addition of process level constructs to the ISPS language will add the following benefits:

1) Clarity of description of large systems of interacting processes. As we strive for higher levels of functional descriptions, this becomes the highest level control construct necessary at the functional level. As functional units become integrated on chips and systems are built with them, it becomes necessary to describe large systems as the interconnection of process level components.

2) Design synthesis systems will be able to recognize the "handshake" as a functional operation with a variety of possible implementations rather than the setting and clearing of flags. Also, verification programs will be able to detect process communication.

We have shown that the introduction of the semantics to cover process level description can broaden the horizons of the language. As this paper is not intended to be a language proposal, we have not set about to define a complete set of semantics and syntax for these constructs. However, we feel that such work should be undertaken. Many of these limitations we have discussed have been overcome in one way or another in the applications described here. Either the applications have operated in a limited fashion or the necessary semantics have been defined by a particular application. Only in the case of process termination,suspension and priorities has little been done. In this case, since most of the multi-process descriptions have involved I/O and interfaces a separate HDL, SLIDE has been proposed [21]. However, it is desirable to have a single language to describe digital systems and thus it is time that such constructs be incorporated into ISPS.

6. THE IDEAL BEHAVIORAL LANGUAGE

Using ISPS in a variety of applications has allowed us to understand some specific as well as global requirements of a behavioral language. A discussion of these would require a paper in itself. Here, we just list these and indicate why they seem reasonable to us. They are:

1. An abstraction facility for adding new primitives to the language[1] as hardware becomes more complex.

2. The ability to specify behavior without specifying structure.

3. Support for structured programming constructs.

4. The capability to specify additional information which may be application specific (e.g. with qualifiers).

5. The capability to express concurrency more precisely than the ISPS semicolon now allows.

6. The capability to describe multi-process functionality, including terminations, suspensions, and priorities.

[1] These primitives may not be functions of existing primitives, so simple text substitution is not all that is implied here.

7. The capability to express synchronization primitives explicitly.

8. Formal semantic definition of the language operators to the greatest possible extent.

It should be noted that ISPS incorporates items two, three and four now as three of its main advantages.

7. CONCLUSIONS AND ACKNOWLEDGEMENTS

ISPS has proven invaluable in providing a useful tool for digital description, synthesis and simulation. A great deal of research would not have been possible without ISPS and the efforts of Mario Barbacci to continually support the language. The success of these applications speaks to the advantages and usefulness of ISPS; because of that we can concentrate here on the limitations. We feel that the best language development can continue by testing, trying and exercising an existing language in order to discover both desirable and undesirable qualities. Users (particularly undergraduate students with homework due) are the cruelest of critics - and the most honest. The ISPS user community had enormous effect on this paper; virtually every point raised here came from this group. Vittal Kini, Gary Barnes, and Andy Nagle provided many of the ideas and comments. Lou Hafer, Bill Morgart,and John Oakley also participated in many discussions. Additional comments were provided by Pete Alfvin, Charlie Hayden, Bill Overman, Leo Marcus, Sarma Sastry, Dono Van-Mierop and Will Sherwood. Valuable feedback came from many industrial CAD groups.

References

1. Alfvin, Pete. A Formal Definition of AMDL. Master Th., Computer Science Department, UCLA, 1979.

2. Barbacci,M., Siewiorek,D. The CMU RT-CAD System: An Innovative Approach to Computer Aided Design. American Federation of Information Processing Societies Conference Proceedings vol.45, Amer. Fed. of Information Processing Societies, June, 1976, pp. 643-655.

3. Barbacci,M., Barnes,G., Cattell,R., Siewiorek,D. The Symbolic Manipulation of Computer Descriptions ; The ISPS Computer Description Language. Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., March, 1978.

4. Barbacci,M., Nagle,A. The Symbolic Manipulation of Computer Descriptions ; ISPS Application Note: An ISPS Simulator. Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., March, 1978.

5. Barbacci, M., Siewiorek, D. An Architectural Research Facility - ISP Descriptions, Simulation, Data Collection. *AFIPS Proceedings 46* (1977), 161-173.

6. Barbacci,M. *Automated Exploration of the Design Space for Register Transfer (RT) Systems.* Ph.D. Th., Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., November 1973.

7. Bell, C., Newell, A. *Computer Structures: Readings and* Examples. McGraw-Hill Book Co., New York, 1971.

8. Cattel,R. *Formalization and Automatic Derivation of Code Generators.* Ph.D. Th., Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., April 1978.

9. Crocker, S. *State Deltas: A Formalism for Representing Segments of Computation.* Ph.D. Th., Computer Science Department, UCLA, 1977.

10. Darringer, J. *The Description, Simulation and Implementation of Digital Computer Processors.* Ph.D. Th., Department of Electrical Engineering, Carnegie-Mellon University, 1969.

11. Hafer,L., Parker,A. Register-Transfer Level Digital Design Automation: The Allocation Process. Design Automation Conference Proceedings no. 15, ACM SIGDA, IEEE Comp. Soc. Tech. Com. on Design Automation, June, 1978, pp. 213-219.

12. Morgart, W. Automatic Emulator Generation. Work in progress, Department of Electrical Engineering, Carnegie-Mellon Univ., Pittsburgh, PA, April, 1979

13. Nagle, A. An Investigation of GLIDE - A Generalized Language for Interface Description and Evaluation. Master Th., Dept. of Electrical Engineering, Carnegie-Mellon University, Sept. 1976.

14. Oakley, J. *Symbolic Execution of Formal Machine* Descriptions. Ph.D. Th., Computer Science Department, Carnegie-Mellon University, April 1979.

15. Parker, A.C. Description and Simulation of Microcode Execution. Proceedings of the 5th Annual Computer Architecture Symposium, ACM SIGDA, IEEE Computer Society, April, 1978.

16. Parker, A.C., et al. The CMU Design Automation System . Design Automation Conference Proceedings No. 16, ACM SIGDA, IEEE Tech. Comm. on Design Automation, June, 1979.

17. Parker, A.C. Digital Interface Description. Proceedings COMPCON, IEEE Computer Society, February, 1978.

18. Snow,E. *Automation of Module Set Independent Register Transfer Level Design.* Ph.D. Th., Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April 1978.

19. Thomas,D. *The Design and Analysis of an Automated Design Style Selector.* Ph.D. Th., Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April 1977.

20. vanMierop, D., Crocker, S. and Marcus, L. Verification of the FTSC Microprogram. Proceedings of the 11th Annual Microprogramming Workshop, ACM SIGMICRO and IEEE Tech. Comm. on Microprogramming, November, 1978.

21. Wallace, J. and Parker, A. SLIDE: An I/O Hardware Descriptive Language. to be published in the Proceedings of the 1979 International Symposium on Hardware Descriptive Languages, Palo Alto, CA, October

22. Wick,J. *Automatic Generation of Assemblers.* Ph.D. Th., Dept. of Computer Science, Yale University, New Haven, Conn., 1975.

Proceedings of the

4th International Symposium on

Computer Hardware Description Languages

Palo Alto, California

October 8-9, 1979

W. M. vanCleemput
Conference Chairman

D. Dietmeyer
Program Chairman

IEEE Catalog No. 79CH1436-5C
Library of Congress No. 79-87963

Additional copies available from:

IEEE Computer Society
5855 Naples Plaza, Suite 301
Long Beach, CA 90803

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854