THE REQUIREMENTS FOR EFFECTIVE HARDWARE DESCRIPTION LANGUAGES

by

John A. N. Lee, Deborah Macock,
Peter Marks, T. C. Wesselkamper

Language Research Center
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA, 24061

Report No. CS 75011-R

June 1975

The work reported herein was sponsored in part by the National Science Foundation Grant No. DCR74-18108.

Abstract

The design of hardware description languages (HDL's) is considered with respect to their structural and functional properties rather than their syntactic forms. The contents of an idealized HDL are contrasted with those of nine existing languages, chosen so as to typify a wide range of usage, numerous citations in the literature, and a diversity of source.

Keywords:

Hardware description languages, processor design, language design, language properties, register transfer languages, syntax, semantics.

CR categories: 6.0, 5.21, 5.23, 5.24, 5.26

Introduction

Hardware description languages (HDL's) are descriptors of the structural and functional properties of digital systems. Their creation arises from the natural desire and need to describe these systems in terms of some formal symbolism. Thus the same need for formally describing algorithms that characterizes current developments in programming languages, exists also amongst those who are concerned with the structure and behavior of digital systems. That is, an adequate study of computer hardware is not possible unless some means of formal description exists primarily.

With this fundamental need in mind, considerable effort has been directed towards the development of HDL's within the computer science and engineering community. Notably, since 1973 there have been annual workshops on this subject both in the U.S.A. and in Germany, and a special issue of the IEEE "Computer" (November,1974) was devoted entirely to HDL's. Furthermore a number of textbooks for use both as pedagogical aids and for the professional market have espoused the concepts of HDL's as a means of presenting the structure and behavior of digital systems [1,2,3,4,5].

The high level of activity in this area and growing interest is evident in Su's brief survey [6] of HDL's in use (or available) in the U.S.A. He lists 22 languages that have been or are used as hardware descriptor systems. Various other HDL's are identified as being in use in

Canada, Japan and Europe.

One may well question the rationale behind this apparent frenzy of activity. It is now recognized that HDL's can serve as an important instrument in computer engineering - or more specifically - in digital systems design and implementation. The remainder of this section is devoted to an exposition of the possible applications of HDL's as evidence in justifying the need for such languages. Hopefully an adequate HDL should be able to satisfy the majority of these needs.

Primarily an HDL can be applied as a tool in effective communication and as a mechanism for valuable simulation. As a means of effective communication an HDL can serve as a common language amongst diverse groups of individuals. In a discipline as multi-levelled as digital systems, it is of fundamental importance that ideas be conveyed via a common means of description.

In the field of education, an HDL can become an effective teaching tool used both in the classroom and in supportive material such as textbooks. Where the lack of funds might prohibit the implementation of any large scale (or even medium scale) digital systems in terms of hardware, it would be possible to evaluate student and research HDL descriptions 'on paper' or by means of an interpretive simulation. However, even where hardware is available, the desk checking style of design through the use of an HDL is

obviously most cost effective.

Amongst designers, an HDL could well become a common medium of expression allowing for the comparative discussion of digital systems. The use of an HDL description insures a precise yet concise description of a designer's thoughts. Moreover, the designer's ideas may be more readily and unambiguously understood from an HDL description. Indeed, at the 1973 Rutger's workshop, a number of participants reported that design errors were first noted when a formal description of the design was prepared.

Between the designer and the user of digital systems, an HDL becomes a documentation aid. Its use permits the user to understand more clearly and precisely the structural and functional elements of the system. For the user too, an HDL description can serve as a means for comparing different systems and architectures. As an example of this designer-user communication, Digital Equipment Corporation's handbook for the PDP-11 system [7] contains an HDL description of the effects of executing the machine language instructions. The description is written in ISP, the HDL created by Bell and Newell[4].

As a simulation tool, an HDL can be cost-saving in two respects. Firstly, the modelling of new concepts and designs written in an HDL is less expensive than the actual implementation of a prototype. This method of design also allows the designer more freedom, and thus more creativity

in his approach, particularly where the HDL can be written as such a high enough level that the actual implementation of certain elements is irrelevant. By this means, a general system can be designed to take advantage of yet unspecified (or even uninvented) subsystems.

Secondly, the simulation of digital systems allows the evaluation in terms of cost-effectiveness of those systems before their prototype is constructed. Some HDL descriptions can be mechanically converted into Boolean expressions [8] which may then be analyzed as to their cost of implementation.

In the realm of simulation, an HDL can serve also as a tool in computer aided design. That is, systems could well be built interactively by using a computer and the resulting design be tested subsequently on-line.

Finally, simulation allows the designer to experiment more freely with creation and evaluation of theoretical systems. As illustrated above, HDL's can serve as an important tool in digital systems design and study. As noted, some applications have already been instituted. However, there is a noticeable lack, as of this date, of any published extensive applications of HDL's. With the notable exceptions of the ISP and VDL [4,1] descriptions of D.E.C. systems and the APL description of the System/360 [9], most published uses of an HDL consist of the descriptions of hypothetical or partial systems.

The lack of usage of HDL's may lie in the design of the languages themselves. Rather than designing an HDL from a syntactic point of view, we choose to look instead at the structural and functional requirements of HDL's, disregarding initially the question in which particular form of syntactic sugaring the HDL should be clothed. In the sequel to this section, such requirements are presented and are compared with the facilities provided by a selection of the HDL's in existence.

Declarative and Functional Requirements of an HDL

Before any attempt should be made to develop an HDL it is necessary to determine firstly the requirements of such a language. Once the needs for an HDL have been established, the basic requirement is the capability of the language to adequately describe the properties of the hardware. Essentially these properties can be separated into two classifications; structural and functional effectiveness. By themselves, these two facilities do not guarantee the effectiveness of the language nor do they provide any facilities which ensure the adequacy or completeness of the descriptions generated. Unfortunately, many previous authors have confused HDL requirements with the requirements for the adequacy of programming languages and thus the resulting languages have either been modifications of existing programming languages (e.g. CASD, AHPL) or have been modelled after such languages as ALGOL.

Whereas the purpose of (and hence the basic requirement to be fulfilled by) a programming language is to provide a description of a process to be executed by a computer, the purpose of an HDL is to describe a processor rather than a process. Hence it is not a foregone conclusion that a programming language is adequate as a media for describing a processor, except insofar as the effects of executing a program (i.e. a process) can be modelled by a programming language.

Totally separate from the requirements that an HDL should be capable of describing the structural (i.e., register, memory, semaphore, etc.) and functional (operational) elements of a processor, it should be also capable of describing the essential timing aspects of processor operations. That is, the language should contain explicit facilities for relating the activation of functional elements to some notion of time, without the user having to 'invent' his own models. Furthermore such a facility should not be so overbearing as to impose timing constraints which then permit the modelling of only certain architectures; rather timing should be a facility under user control. Within the general heading of timing, two subclassifications can be identified:

(i) the facility to describe the concurrency of execution and

(ii) the ability to relate the timing of activities to either a clock or to other activities.

Concurrent activities within a processor are common enough that any language which does not provide facilities for defining either a set of concurrent activities to take place during a given cycle or which does not provide the facility to spawn concurrent (co-operating) tasks, is inadequate. This does not require that any software implementation of an HDL must also provide concurrency, but must at least provide a simulation of concurrency.

Separate from, but related to, concurrency is the problem of providing a facility for synchronous or asynchronous

modes of activity. Asynchronous execution is most commonly provided in the primitive form of a conditional prefix to an activity statement, the object of the condition being some logical variable (possibly a semaphore) chosen by the user. This can be characterized by such programming language control structures such as 'WAIT UNTIL' or 'ON CONDITION'.

Having provided an asynchronous timing facility, a form of synchronous timing can be simulated by naming the clock pulses (e.g. t1, t2, etc.) and utilizing the pulse names in an (asynchronous) conditional expression.

However, this facility is not sufficient when timing is not to be related to (say) the origin of a train of pulses (such as the start of a sequence of pulses necessary to complete a fetch cycle), but instead is related to an independent clock which is activitated at an instant chosen by the user. We shall refer to these two forms of synchronization as being synchronization by 'pulse identification' or by 'discrete pulse counting'.

Where concurrent operations are possible it is necessary to ensure the integrity of the separate operations. For example, using a one port memory, once a memory reference has been initiated then all other calls upon that unit must be locked out. Similarly, restrictions may be necessary where a processor uses a single element in the performance of several functions. Hence an adequate HDL should include a provision to lock out references to certain structural and

functional elements to ensure the integrity of an activity which is currently underway. Conversely, if no lock out feature is provided by the language, then some facility must be provided to inform the user of potential conflicts, such as the use of a common register by two concurrent activities.

The lock out from the concurrent use of structural elements must also exclude the usage of single data paths or busses. Whilst certain activities may involve disjoint sets of registers, if those registers are all attached to a common bus, then the competition for the use of the bus may be disasterous. Hence an HDL should have the facility to declare the routing of data between elements, and in particular which other intermediate elements may be utilized. Whilst the existence of data paths may be derived from statements within the description, such as move operations, the use of intermediate elements may not be obvious. At a high level of description, the actual implementation of a data transfer between two registers may not need description. For example, at the user's level of description (as found in manufacturer's machine language manuals) a memory referencing instruction may be described as operating directly over the accumulator and the selected memory location, by such an expression as

contents(Mem[ea]) * contents(Acc) => Acc

where ea is the effective address of the command being

executed. At this level of description no mention is made of the necessity to jam the effective address into the memory address register before a reference can be made or that the complete read cycle activity actually consists of two steps - the fetch operation and the restore operation. Such a level of description ignores the fact that even after the fetch cycle has been completed and the data is available to the activity which initiated the read operation, the memory is still unavailable due to the ensuing restore activity.

There are two extremes at which an HDL may exist; the high level which is best suited to giving an understanding of the machine language to a programmer or the very low level of description which describes the processor in terms of the gates or logic levels. Whilst these two extremes are necessary by themselves, they represent only isolated instances of the total domain of an HDL. To be really effective an HDL should be capable of covering both these extremes of description, and innumerable levels between. Furthermore these levels must cover not only the ability of the language to provide many levels of operational description (such as by the use of procedures or functions) but also the ability to have levels of description of the structural elements. Whilst there may exist, at a high level, descriptions of registers in terms of character symbols and a set of operations over those registers in that format, lower levels must allow both an expansion of the operators into micro-operations and also the more detailed description of the registers in terms of (say) the ASCII

character representation or at a lower level still, in terms of bit patterns. Depending on the point of view of the reader, such a hierarchy of descriptive techniques can be considered as either the ability to provide an expansion of details or as the capability to suppress unnecessary detail. This facility should also contain the capability to allow the description to reference 'black boxes' which are defined only in terms of their input/output characteristics and (possibly) their delay time. This capability will permit the inclusion in the description of predefined elements such as chips whose inner construction is of no concern.

In summary, the facilities to be provided by an HDL must include:

1. the ability to name and DESCRIBE LANGUAGE ELEMENTS which correspond directly to the structural and functional elements of the processor being described,
2. the availability of MNEMONIC NAMES, sufficient to connote some of the purposes of the named elements,
3. facilities to describe the sequencing of activities and in particular (beside simply sequential operations) the specification of CONCURRENT ACTIVITIES, either by the explicit listing of those activities which can occur concurrently, or by the ability to spawn independent parallel activities,
4. language constructs to describe SYNCHRONOUS and ASYNCHRONOUS activities, synchronization being available by pulse identification or discrete pulse counting,
5. the ability to prevent interference between

concurrent activities by a LOCK OUT facility, or the reporting of potential conflicts, and the identification of DATA PATHS between elements to preserve the integrity of data transfers, and

6. language facilities to suppress unneeded details of both the structural components of the processor being described and of the operations of the processor, or the ability to expand on the descriptions and operations, so as to create a HIERARCHY OF DESCRIPTIONS.

Facilities provided by nine existing HDL's

Nine existing HDL's were chosen for the purposes of comparing the requirements developed above and the facilities provided by those languages. This particular set of languages represent, in the authors' opinion, the set which are most frequently cited in the literature and which cover the domain of styles of language. A summary of the features of these HDL's is contained in Table 1.

DDL [Duley, J. R. and Dietmeyer, D. L., A Digital Design Language (DDL), IEEE Trans. on Comp., Sept., 1968, pp. 850-861]

In DDL language elements can correspond directly to the functional and structural elements of the system being modelled. All structural elements are defined in the description by the use of a declarator (i.e., TErminal, MEmory, etc.), where the first two characters uniquely determine the type of element. The use and definition of mnemonic names is allowed with very little restriction as to the choice of names.

Functional elements may be defined as being blocks of logical statements. There is a large set of primitive operators available for the construction of logical statements. These blocks are viewed as behaving as automata. Structural and functional 'boxes' may be declared

to be of the block type Element, which effectively declares them to be primitives in successive statements.

DDL permits the concurrency of operations. Syntactically, this is specified by using a comma rather than a semi-colon to separate operations. It appears that concurrency is allowed only at the level or primitive operations - not amcng procedures.

Synchronous operations within blocks of automata is available. Synchronous timing is possible by either pulse identification or by the specification of a delay in terms of some multiple of clock pulses.

Asynchronous operations are effected by preceding statements with conditional expressions.

Data path specification is permitted and defined by the use of the TErminal declaration. Terminals, so declared, may be used in logical equations elsewhere within the model. No provision seems to have been made for insuring data-path lockout amongst functional or structural units.

DDL is adept at describing digital systems in a hierarchical manner. Being a block-oriented language, DDL allows the inclusicn of sub-structures and sub-functions within a larger block of description.

CDL [Chu, Yaohan, Computer Organization and Microprogramming, Prentice-Hall, Inc., New Jersey, 1972.]

The Computer Design Language (CDL) is very similar to the ISP in its desirable characteristics and its limitations. The structural elements, which must be declared explicitly, include such components as registers, random-access memory, switches, and clocks. The functional relationships of these elements are described at a primitive level by commonly used basic operators; special operators may be defined for higher level descriptions in terms of the basic operators. The elements and the instruction set are given meaningful names and, like the ISP, the CDL develops a highly readable descriptive system.

The CDL allows for the description of both parallel and sequential operations and, at a high level, the suppression of timing information produces an asynchronous description. All synchronous description involves a conditional test for an appropriate clock pulse; as in the ISP there is no facility for counting discrete pulses. Valid data paths are not declared explicitly, but are specified implicitly whenever a data transfer is indicated; apparently there is no consideration of lock outs. Both structural and functional components may be described hierarchically, depending upon the desired level of description; however, again the expansion of detail and the degree of complexity are limited.

LOTIS [Schlaeppi, H.P., A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS), IEEE Trans. Electr. Comp., Aug. 1964, pp. 439-448]

By our count, this language is one of the most frequently cited in the literature on HDL's, and yet since the original publication in 1964, no further formal publications have been forthcoming. LOTIS is perhaps the most sophisticated HDL considered herein and best fulfills the requirements which are developed here.

A LOTIS description consists of declarative and procedural parts, the language having excellent facilities for hierarchical descriptions of both the structure and functions of the hardware elements. Sets of LOTIS statements can be executed concurrently and a succession of sets of statements can be defined to be activated sequentially. Furthermore, LOTIS has the ability to spawn groups of statements (syntactically similar to procedures) which can be executed asynchronously by use of the spawning process. Interlocks between active groups of statements are automatically included in the language and extra levels of lock out can be added by the users. Timing of the activation of sets of statements can be specified either in terms of named pulses or in terms of the duration of that set of statements which, in combination with the ability to sequence sets, can effectively provide synchronization by pulse counting.

Data path specification is provided only in terms of assignment statements between named elements (registers). Lock outs between paths are possible only if the condition of the use of (say) a bus is recognized by the user and a lock out key is added to the definition.

LOTIS contains a hierarchy of actions and descriptions from the "atomic" level of components such as registers and memory, to the level of named sequences of activities (steps) and procedures and functions. The latter differs from a procedure only in the fact that a function implies not structural detail to its implementation, thus facilitating a "top down" approach to description generation.

Data paths are defined implicitly when a data transfer is indicated, but there is no apparent explicit declaration of valid data paths. There is also no facility for the specification of lock outs. Although the ISP does provide for a hierarchical description of both structural and functional components, the attainable level of complexity and detail is limited. This exclusion of detail, although perhaps undesirable at lower levels of description, has contributed to the high readability of the ISP, a quality which is further enhanced by the use of mnemonic naming.

Data paths are defined implicitly when a data transfer is indicated, but there is no apparent explicit declaration of valid data paths. There is also no facility for the specification of lock outs. Although the ISP does provide for a hierarchical description of both structural and functional components, the attainable level of complexity and detail is limited. This exclusion of detail, although perhaps undesirable at lower levels of description, has contributed to the high readability of the ISP, a quality which is further enhanced by the use of mnemonic naming.

CASSANDRE [Bogo, G., et al, CASSANDRE and the Computer-Aided Logical Systems Design, Proc. IFIP, 1971, pp. 1056-1065]

CASSANDRE is a language for interactive computer-aided design. In general, CASSANDRE's language elements correspond to the functional and structural elements of digital systems. Structural elements are explicitly defined by the use of declarators (e.g. Unit, Memory, etc.). Functional modes of operation are also declared (e.g. Synchronous, Asynchronous, etc.). A fairly large set of primitive operators are available for use, including many elements which appear to be derived from programming languages (e.g. if, goto, stop, do, etc.).

The use of mnemonic names for structural and functional elements is permitted, although the examples in the reference do not appear to make much use of this facility.

Concurrent execution of operations is permitted and is denoted by a list of operations separated by commas. Synchronous operation is implemented by the use of the pulses from a declared clock. There appears to be no ability to count clock pulses. Asynchronous operation is effected by the appropriate use of conditional statements.

Data paths may be declared in a description (by the use of the Bus declarator), although it is not clear as to how these paths might be used in a system description. There appears to be no provision for handling lock outs by

CASSANDRE's interpreter.

The basic notion of CASSANDRE is a unit. Each unit or box is only accessible from the outside by a set of input-outputs. A complete CASSANDRE description is a set of trees of units (each unit may contain more interconnected units). Thus it is possible to describe systems at different levels of conception quite readily.

CASD [Crockett, E.D., et al, Computer-Aided System Design, Proc. FJCC, 1970, pp. 287-296]

The Computer-Aided System Design (CASD) language is a variation of PL/I, developed specifically for use in computer design. Having basically a PL/I structure, a CASD description consists of several procedures, possibly nested, which represent the logical modules of the design. The data items operated on by the procedures (registers, terminals, etc.) are the structural elements of a CASD description; each must be explicitly declared and its logical and physical attributes described. Through this declaration, it is possible to assign meaningful names to the structural elements; in addition, meaningful labels are easily implemented to identify procedures. To facilitate the description of functional elements, the set of PL/I operators has been extended to include additional Boolean operators. Hierarchical description of structural elements is accomplished by the declaration of substructures; as in PL/I, functional elements are described hierarchically via the procedure.

The concept of concurrency is incorporated by defining the basic unit of work, the 'node', as a collection of actions performed simultaneously. The contents of a node are determined using the following rules: operations are written sequentially and the results will be the same as if they had been performed sequentially; sequential statements will always be combined into a single node (executed

concurrently) if there is no 'logical conflict'. Provision is made for overriding these rules by specifying that a labelled statement always begins a new node and by allowing the use of DO CONCURRENTLY to explicitly declare parallelism. The concurrent execution of several tasks can be accomplished by implementing the TASK option of the CALL statement.

This treatment of timing eliminates the necessity of explicit lock out checking; for, sequential operation is assumed whenever logical conflicts occur. The timing rules are furthermore independent of any physical clock - sequencing involves neither pulse identification nor pulse counting. Data paths are implicitly specified by the transfer of data items; apparently, there is no facility for declaring valid paths prior to the transfer of data.

RTL [Schorr, H., Computer Aided Digital System Design ..., IEEE Trans. Electr. Comp., Dec. 1964, pp. 730-737]

Of the many languages surveyed, RTL (Register Transfer Language) is not only one of the earliest but also is the most primitive. The description of a processor is based on the statement of sets of register data transfers supplemented by elementary operations and conditions. The elements of a set of transfers are assumed to be executed concurrently; the particular set of transfers chosen being dependent on the prefixing conditions. The ability to use meaningful names for processor elements is included in the language by the omission of any specific restrictions, though the examples cited by Schorr contain only very primitive naming schemas. The conditions prefixing a transfer set may be in terms of the contents of semaphores or the pulse time. By implication, pulse times are identified by the naming convention t<digit> and thus it would appear that RTL is restricted to synchronization by pulse identification and, lacking an ability to count pulses, to asynchronous operations. No formal abilities are provided to facilitate lock outs of concurrent activities, and it is not entirely clear whether the concurrent activities operate over a common initial machine state and generate a common final state, or whether all activities immediately affect the state of the registers.

RTL contains no declarative statements, the attributes of a register (including segmentation) and the valid paths

between registers being defined only by implication.

Schorr's RTL is actually an adaptation of an earlier version [5] and contains notations for various operations such as indirect addressing, decoding and so on. Since it is basically a primitive level language (i.e. the register level), no hierarchy of description is provided.

LOGAL [Lund, J.F., LOGAL, Sperry Univac TMA 00317 Revision B, 1974.]

The Logic Algorithm Language (LOGAL), primarily developed as an aid to logic design, is essentially a language for lower level descriptions. Structural elements are either register (memory) or non-register (link) and can be described as vector (up to 100 bits) or two-dimensional array. The assignment of names to these elements is highly restricted - each name consists of four parts which contain physical placement and descriptive information (e.g. F9 A3 R 02 describes bit 2 of the register A3 located in section F9 [14]), hence mnemonic naming is not a facility of the language.

All functional elements are either expression operators or transfer operators. The former group includes arithmetic, relational, and logical operators; these can be combined to form expressions which correspond directly to combinatorial networks in the machine. The transfer operators are used to describe the transfer of data between registers, and it is only when such a transfer is indicated that data paths are designated ; no facility exists for the specification of valid paths. Through the use of the operator definition statement, the user may define special operators in terms of the primitive function elements, hence there is a limited capability to describe functions hierarchically.

An event statement is always composed of a transfer of data and the conditions which must be met for the transfer to occur. There is no provision for parallel execution of events - the event statements are encountered sequentially and executed if all conditions are satisfied. A specific location in each event statement is reserved for an identified clock pulse. A '0' in this location indicates that it is not being used (asynchronous sequencing is possible); otherwise, the correct clock pulse must be present for the statement to be executed. In this manner, synchronization by pulse identification is possible; however, no facility exists for pulse counting. There is further no apparent consideration of the integrity of the system - the specification of lock outs is not a part of a LOGAL description.

AHPL [Hill, F. J. and Peterson, G. R., Digital Systems: Hardware Organization and Design, John Wiley and Sons, Inc., New York, 1973.]

A Hardware Programming Language (AHPL) incorporates the notation of APL and those APL operations which satisfy the constraints of available hardware. Its basic structural elements are registers, flip-flops, busses, and main memory; and basic functional elements are described in standard APL notation. The structural elements and special operators, which are declared and defined as subroutines, may be assigned meaningful names, although in examples [2], this was rarely done.

One of AHPL's more desirable characteristics is its flexibility in the amount of detail that can be suppressed or included at different levels of description. It has the ability to describe both parallel and sequential operations, either by suppressing timing information entirely or by incorporating it to a high degree. Asynchronous operations are described, not only by using conditional tests, but also by implementing completion signals and the use of WAIT to indicate delay. In synchronous description, the facility exists both to test for identified pulses, and also to count pulses and delays.

Although not required in a formal description, it is claimed [2] that bus load and bus logic declarations are

possible (no examples of this were found), allowing explicit declaration of data paths between registers; however, there is no apparent facility for indicating lock outs. Hierarchical description of both structural and functional elements is possible in AHPL, and the descriptive system is quite flexible in the level of description for which it can be used.

Table 1. Features provided by existing HDL's

Y - yes
N - no
I - implicit
C - claimed, but no examples given

			DDL	CDL	LOTIS	ISP	CASSANDRE	CASD	RTL	LOGAL	AHPL
1. ELEMENT CORRESPONDENCE	Structural		Y	Y	Y	Y	Y	Y	Y	N	Y
	Functional		Y	Y	Y	Y	Y	Y	I	I	Y
2. MNEMONIC NAMES			Y	Y	Y	Y	Y	Y	Y	N	Y
3. CONCUPRENCY			Y	Y	Y	Y	Y	Y	Y	N	Y
4. TIMING	Synch by	pulse id	Y	Y	Y	Y	Y	N	Y	Y	Y
		count	Y	N	Y	N	N	N	N	N	Y
	Asynchronous		Y	Y	Y	Y	Y	Y	Y	Y	Y
5. INTEGRITY	Lock outs		N	N	Y	N	N	Y	N	N	N
	Data Path Defn.		Y	I	I	I	C	I	I	I	C
6. HIERARCHY	Functions		Y	Y	Y	Y	Y	Y	N	Y	Y
	Components		Y	Y	Y	Y	Y	Y	N	N	Y

Language Forms

Having determined the set of elements to be included in the HDL in order to satisfy the structural and functional requirements, there remain two further language design considerations. Leaving aside the syntactic form of the language, of primary concern to the designer next must be the general control form of the language.

In the introduction, we attempted to put aside the notion that an HDL should have the attributes and properties of a programming language, on the basis that the object being decribed by an HDL is a processor rather than a process. That is, an HDL describes properties rather than actions, even though the properties include events anticipated to occur during the period of activation of some functional unit. The confusion that can arise in this connection may lie with the notion that a processor can be simulated in terms of a program written in a high level programming language. However, the authors believe that this form of a description is overly restrictive and suggestive, and may by its own nature, influence the descriptions to be generated. Taking an example from programming, it is well known that the language chosen for implementation of an algorithm can seriously affect the style and form of the program which solves some particular problem. In certain instances the language chosen can also require a modification in the algorithm chosen for the solution to the problem. This effect can be seen even more clearly if the form and

properties of the programming language are examined. For example, the style of programs written in APL and the algorithms used for solution of a problem are normally significantly different from the solution to the same (programming) problem written in PL/I. Leaving aside the obvious syntactic differences between the two languages, the proficient users of the two languages will generate different solutions because of the language used rather than the problem to be solved. This difference arises not only from the structural and functional properties of the two languages, but also from the styles of the two languages. The major difference between these two particular languages is the control structures in which the structural and functional facilities can be embedded. Whereas an APL program is fundamentally a sequence of expressions over the unified domain of n-dimensional (regular) arrays, function descriptions and expression identifiers with only one syntactic form of statement, PL/I is a conglomerate of syntactic forms each of which is applicable to a restricted domain of discourse. Consequently, where the APL program can embed the control structure of a solution within the imperative statements, the equivalent PL/I program contains two disjoint parts; the actions to be taken over the domain of data elements and the control portion.

The same effects can be expected to occur between two instances of an HDL; two designers given the same requirements can satisfy them in radically different manners, such that the resulting HDL's will have distinct

effects on the descriptions which will be generated using them. Let us consider several forms of descriptive systems.

1. An HDL could be a totally declarative language, a generated description being centered around the key structural elements of the processor. The description associated with a structural element could then be divided into three parts:

(i) the description of the structural form of the element,

(ii) a set of applicable operations over this element and their effects on this element and its external connectors, and

(iii) a listing of the external connections between this element and other elements (though the latter may be restricted to refer only to common pins or busses) of the processor, showing not only the output from this element, but also the inputs which will affect or stimulate it.

The control structure operating over such a description, so as to demonstrate the effect of executing a program, is the repeated simultaneous activation of ALL elements of the processor, some of which will be operative whilst other elements will be dormant.

A Description of a single component of the processor being considered
The set of operations which are applicable to this component, and their description in terms of elementary operations. Note that these descriptions can reference only "local" elements and the interfaces to other components.
A listing of the set of connections between this element and the other components of the processor.

Figure 1. The declarative form of an HDL

2. An HDL may be composed of a set of structural declarations of the elements of the processor and a CASE statement. This form of description may be centered not on the structural elements of the processor but instead on the major conditions which can occur during the process of executing the instructions found in the machine being described. The action associated with each element of the case statement may itself be a case statement, thereby subdividing the operative actions further.

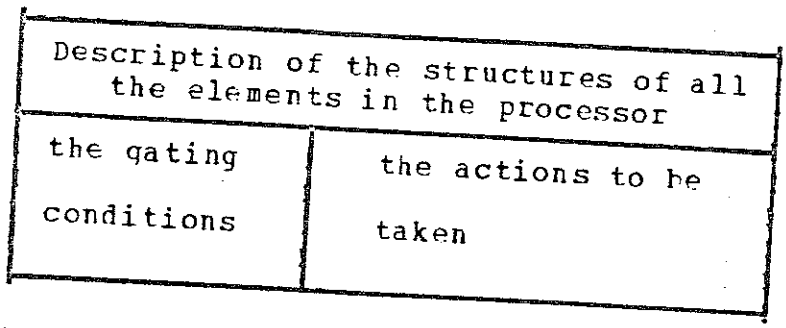

Figure 2. The case statement form of an HDL

3. An HDL may be composed of a description of the structural elements of the processor and a listing of the events that can occur during a highly ordered sequence of time intervals. Such a description may be based on the major cycles which are said to occur during the execution of instructions, such as the fetch and execute cycles.

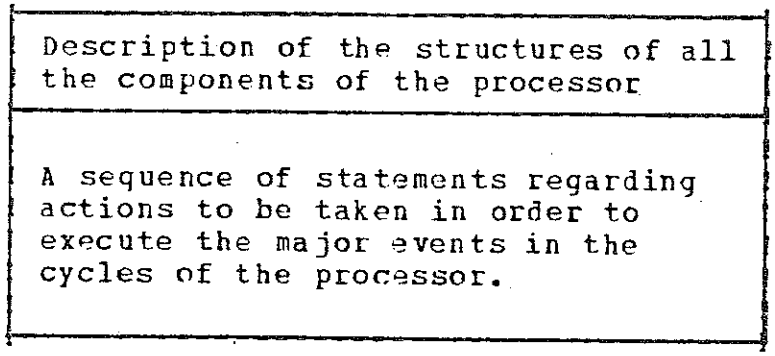

Figure 3. The sequential form of an HDL

4. Finally an HDL may be some combination of the above three forms so that a hierarchy of descriptions can be achieved. For example, a descriptor may be sequential in nature at a high level (so as to more closely align with a programmer's notion of operations), be a set of case statements at the level of logical (architectural) design, and be declarative in form at the level of module description (that is, the "gate" level.)

We have purposely avoided the question of the syntactic form of an HDL in the preceding discussion, even though it is this element of a language which initially makes it acceptable to a potential user. In the main, syntactic sugaring provides the clothing which makes the resulting description palatable. In the discussion of the functional and structural requirements of an HDL, the only reference to syntax was in the form of the requirement that naming conventions should be such as to allow the mnemonic expression of identifiers so as connote the usage of the named element. Given this primary requirement together with a certain amount of common sense and a feeling for what is and what is not readable, the resulting concrete form of the language should be acceptable.

Chomsky [15] describes two models of language which are particularly applicable here; the competence model and the performance model. By his description, the competence model is fundamentally the knowledge of the valid syntactic forms of the language used to the stage where the utterances are at least grammatical. However, the ability to perform well in a language requires knowledge of the domain of discouse and breadth of understanding of the concepts to be transmitted in a communication. It is important that any language used for the description of hardware be capable of sustaining a user to be both competent in the forms of the language and to be capable of performing well.

This paper has sought, primarily, to satisfy the requirements of a performance model by seeking out those requirements of an HDL which will best serve the needs of the user community and be capable of adequately describing a wide range of hardware features. The set of requirements listed herein are not intended to be a maximal set; rather, this set represents that which the authors believe to be the minimum required to support a skeleton descriptor. As with programming languages, features may be added for the convenience of the user, and it is for this reason that there exists the requirement of the hierarchical nature of an HDL.

The ability of a hardware description language which is based on the requirements espoused herein, to fulfill the needs of the user community can be judged in two manners. Firstly, as pointed out in an earlier section, the requirements suggested were compared with a set of existing languages. The mere fact that these requirements appeared as facilities in these languages is at least some encouragement that the set is necessary, if not sufficient. Secondly, the correctness of the set of requirements can only be determined by the use of a language over a period of time in the environment for which it was designed. In this case, no actual language has been designed or implemented but the experience of the authors in using existing languages to describe various minicomputers (and at least one micro-processor) leads us to believe that given those

requirements in a language the task of description would have been much simpler and decidedly more straightforward.

Notwithstanding the containment of our requirements in the languages considered, a large number of them fail to meet even a majority of our criteria, and in fact, not one of the languages tested, nor any other language of the authors' knowledge satisfies all the listed requirements. Apart from the hierarchy of language forms, the major difference between the properties of existing languages and the set of requirements is the explicit declaration of data paths between structural components. This requirement was included on the basis of the experience of the authors with the intention of fulfilling two basic needs:

(i) the need to identify the totality of components which are required to perform the data transfer,

(ii) the ability to identify common data paths used by several components and thus to prevent conflicts between concurrent activities.

In several cases where existing HDL's were used to describe processors it was realized that data transfers expressed simply as an assignment statement, such as MBR => ACC , were insufficient when such a transfer utilized other elements of the system such as a common data bus. In other instances, manufacturer's descriptions of data transfers omitted to mention that the transfer was performed by way of an intermediate register (presumably for economic reasons rather than speed.) Combining these difficulties with the problems of specifying the lock out conditions for certain

processes, it was obvious to the authors that explicit data path determination was a necessary requirement of any future HDL.

The set of HDL's considered in this study may be subdivided into two groups; the high level descriptors and the register transfer languages. By our requirement that an HDL be hierarchical in nature, both with respect to the functional and the structural descriptions, this distinction should disappear, the register transfer level being a hierarchy within the generalized descriptors.

The design of any language, be it for the purposes of communicating ideas, concepts or instructions, cannot be commenced at the level of the concrete syntax of the language. To pick upon some syntactic structure and then to ascribe semantics to that entity must eventually lead to a language which contains features which have been chosen on the basis of their syntactic elegance rather than their usefulness. Whilst we recognize that languages are learnt (most often) from the point of view of syntax followed by semantics, or from the point of view of Chomsky, from the use of a competence model to the generation of a performance model, the design of a language is not so ordered. Similarly, to decide how to say something before one has decided what to say, is an inverted process. This paper attempts to start at the "what" end of the design of an HDL and suggests that having satisfied that need, the "how" can be added as a last step of the production process.

References

[1] Lee, J.A.N., Computer Semantics, Van Nostrand Reinhold Co., New York, 1972.

[2] Hill, F.J. and Peterson, G.R., Digital Systems: Hardware Organizatien and Design, John Wiley & Sons, Inc., New York, 1973.

[3] Chu, Yaohan, Computer Organization and Microprogramming, Prentice-Hall, Inc., New Jersey, 1972.

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

[5] Bartee, T.C., et al, Theory and Design of Digital Machines, McGraw-Hill Book Co., New York, 1962.

[6] Su, S.Y.H., A Survey of Computer Hardware Descriptions Languages in the USA, Computer, Dec., 1974, pp. 45-51.

[7] Anonymous, PDP-11/20,15,R20 Processor Handbook, Digital Equipment Corporation, Maynard, Mass., 1971.

[8] Duley, J. R. and Dietmeyer, D. L., Translation of a DDL Digital System Specification to Boolean Equations, IEEE Trans. on Comp., April, 1969, pp. 305-313.

[9] Falkoff, A. D., Iverson, K. E. and Sussenguth, E. H., A Formal Description of SYSTEM/360, IBM Systems Journal, Vol. 3, No. 3, 1964, pp. 198-263.

[10] Schlaeppi, H.P., A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS), IEEE Trans. Electr. Comp., Aug., 1964, pp. 439-448.

[11] Bogo, G., et al, CASSANDRE and the Computer-Aided Logical Systems Design, Proc. IFIP, 1971, pp. 1056-1065.

[12]Crockett, E.D., et al, Computer-Aided System Design, Proc. FJCC, 1970, pp. 287-296.

[13]Schorr, H., Computer Aided Digital System Design..., IEEE Trans. Electr. Comp., Dec., 1964, pp. 730-737.

[14]Lund, J.F., LOGAL, Sperry Univac TMA 00317 Revision B, 1974.

[15] Chomsky, N., Aspects of the Theory of Syntax, The M.I.T. Press, Cambridge, 1965.