A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

LA-UR-84-2343

CONF-840872--13

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36.

TITLE: DESIGN AND DESCRIPTIVE TOOLS FOR SYSTOLIC ARCHITECTURES

AUTHOR(S): Paul Steven Lewis

LA-UR--84-2343

DE84 015521

SUBMITTED TO: Conference on Real Time Signal Processing VII, Part of SPIE's 28th Annual International Technical Symposium and Instrument Exhibit, being held in San Diego, California on August 19-24, 1984.

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Design and descriptive tools for systolic architectures

Paul Steven Lewis

Los Alamos National Laboratory, Electronics Division
Mail Stop J957, Los Alamos New Mexico 87545

Abstract

Automated design and descriptive tools are essential for the practical application of highly parallel special-purpose hardware such as systolic arrays. The use of special-purpose hardware can greatly increase the capabilities of signal processing systems. However, the more limited applications base makes design costs a critical factor in determining technical and economic viability. Systolic systems can be described at several levels of abstraction, each of which has unique descriptive requirements. This paper focuses on the descriptive issues involved at the system architectural level. Tools at this level must bridge the gap between logic- and circuit-oriented computer-aided design tools and algorithmic descriptions of systolic architectures. Traditionally, hardware description languages (HDLs) have been used at this level to describe conventional computer architectures. Systolic architectures, however, have different requirements. This paper examines these requirements and develops a set of criteria for evaluating HDLs. Four popular HDLs are evaluated and their strengths and weaknesses noted. The final section of the paper summarizes ongoing efforts at Los Alamos to develop a systolic array HDL based on the CONLAN family of languages.

Introduction

Systolic principles[1] can be used to construct special-purpose real-time signal processing systems[2] that achieve high throughput by exploiting algorithmic properties. These principles of regularity, localized communications, and parallel/pipelined execution nicely match the capabilities of integrated circuit technology. Hence, systolic arrays are an attractive method for building high-speed special-purpose hardware to rapidly solve sophisticated problems. However, the use of special-purpose hardware limits the applications base, making fixed costs such as those associated with system design much more critical. Although design costs are in part reduced by the very nature of systolic systems, further reduction can result from the use of automated design and descriptive tools.

The design process stretches from the conception of the algorithm and its mapping onto an architecture down to the electronic implementation. Primary issues involved in the design of special-purpose hardware such as systolic arrays include development of the parallel algorithm, its mapping onto an appropriate architecture, and the design of the circuitry to realize the architecture. At each stage, automated design tools can greatly assist the designer and make the synthesis of special-purpose systems economically viable. For example, confirming the correct performance of an algorithm must precede putting significant effort into the design of the architecture and circuitry for its implementation. This process is complicated by the fact that the algorithm and architecture can be highly interdependent. Validating operation of an algorithm could be very costly if new hardware had to be designed and built to test it. Appropriate design tools can ease this burden by allowing an algorithm and its architectural mapping to be described and simulated in an abstract manner, with only enough detail to check the functioning of the algorithm, but without the extraneous details that would be necessary in the complete design of a hardware realization. Design tools can also assist in transforming the verified abstract architecture into a particular hardware realization. In general, a good set of design tools allows the designer to describe, test, and trade off only those factors that are important at that particular point in the design process.

A principal requirement in automating the design process is a formal notational mechanism that is capable of providing complete and unambiguous descriptions of the concepts being explored. This notational mechanism then provides a common basis for comparisons between alternate methods and an input mechanism to automated design tools. Hardware description languages (HDLs) and their associated design tools have long been used in this manner to aid in the design and description of conventional computer architectures. This paper identifies the notational features that are necessary for the description of highly parallel, regular architectures such as systolic arrays. A set of language criteria is developed. Four popular HDLs are evaluated using these criteria, i.e., their strengths and weaknesses are noted.

Hardware description languages

Digital systems can be viewed at numerous levels of abstraction, ranging from circuit specifications at the transistor level to mathematically oriented algorithmic descriptions. Special-purpose hardware can be divided into the levels defined below.

Data intensive problem. The definition of the problem being addressed by the special-purpose hardware.

Highly parallel algorithm. The general solution to the problem that will be implemented in the hardware.

Abstract machine. A high-level view of the hardware that will implement the algorithm. This includes the overall structure and data flow.

System architecture. Description of how the abstract machine will actually be built. This level deals with the hardware components to be utilized, the interconnection between these components, and their detailed operation.

Circuit/component. The design of the circuits and components used in the system architecture is defined. This encompasses the traditional logic and device levels.

System implementation. The lowest level of abstraction for digital circuits in which the hardware realization in a particular technology is defined.

Tools and notations exist for describing each of these levels. For the higher and lower levels many of these tools are widely accepted. For example, mathematical notation is normally used at the algorithmic level, whereas high-level programming and data flow languages are useful in describing the abstract machine. The circuit/component design level can be addressed by logic diagrams and Boolean equations for digital gates and by schematics and differential equations for their analog realization. In addition, most of the current computer-aided design (CAD) tools address this level.

The description of digital systems at the intermediate system architecture level is usually far less formal, many times consisting of block diagrams and notations in an engineer's notebook. HDLs have evolved over the past 20 years to formalize design and specification at this level. Some benefits that can be derived from the use of a more formal notation are listed below.

- A more precise and concise description of the system than is possible using informal documentation. The HDL can provide a major portion of system documentation and can also serve as a medium of communication among system designers and between designers and users.

- A mechanism to input a system description into automated design tools. The HDL can serve as the input specification to CAD and engineering tools such as hardware compilers and system simulators.

- A descriptive methodology that covers a wide range of levels of abstraction consistently and provides a well-defined interface to both lower- and higher-level descriptions. An HDL can provide a coherent framework in which to describe systems hierarchically and to express the interfaces between separate modules.

Evaluation criteria

The evaluation criteria developed in this paper are grouped into five major categories. These categories include topics that are important to HDLs in general and to the description of systolic systems in particular. The categories are (1) language range, (2) description structure, (3) descriptive methods, (4) timing, and (5) language provisions. A section is devoted to each category. Each section contains a description of what is covered, along with overall goals in the category and their justifications.

Language range

The range of a language is measured by how many of the descriptive levels of digital systems it can effectively address. Originally HDLs were aimed at describing register transfer in digital systems, although some were developed to describe the programming model of a CPU. The trend has been toward a widening of the scope of both traditional and newer languages. A wider range increases the generality of the language and allows it to provide a unified method for describing systems at any level of detail. By providing a uniform descriptive method, the language can support top-down, bottom-up, and combinational design strategies.

A wide range is desirable in an HDL that is to be used as a tool for the design of systolic architectures. Because the principal objective of systolic systems is to match the architecture to the problem, a top-down design approach is generally used. It is useful to be able to express the more abstract architectural structures of the solution first, then to refine them into more detailed hardware implementations within the same framework.

One powerful method of providing a wide language range is through abstraction of data and operations. This abstraction allows a single language to address lower levels of description with primitive data types and operations, and to address higher levels using abstract components defined in terms of the primitive constructs. Abstraction also helps control complexity by allowing detail to be hidden in higher-level descriptions.

Description structure

This refers to the properties of the hardware descriptions written in the language. A language ought to provide the ability to partition system descriptions in a hierarchical, modular manner. This capability allows a system to be described as a series of modules with more complex specialized modules built out of simpler ones. The language should also allow mixed descriptions so that different system components can be described at differing levels of detail.

To handle systolic arrays, the language must have constructs for describing regular architectures. These constructs must include capabilities for describing the regular topology of the architectures and for defining modules and then replicating copies of them. The language should allow control over whether references to a particular module cause a replication of the structure (instantiation of the module) or the sharing of an existing structure. Also, the capability of defining modules parametrically is desirable. Parametric definition allows the specification of module families. A specific family member is then selected upon module instantiation by the values of the parameters.

The HDL must also have the capability to describe various forms of communication between modules and within modules. This capability is critical for systolic arrays in modeling the data and control flow. In general, the language ought to allow the structure or partitioning of the description to match that of the hardware being described at the lower levels or that of the algorithm at the higher levels.

Descriptive methods

This refers to the methods available in the HDL to describe systems. The HDL must be able, at any level, to describe items by structure, function, or behavior. Structural descriptions are those in which a module is defined in terms of interconnection of previously defined modules. Behavior is implicit in the interconnection between and behavior of the components. A behavioral description defines a part in terms of its actions, as a kind of black box with outputs as functions of inputs and internal state. In this case, the structure either is left undefined or is implicit in the behavior. A functional description falls in between, usually describing a module in terms of conditional connections.

Flexibility of descriptive method is required in systolic array work so that the HDL may be used at different stages of the design process. When we first investigate a systolic architecture, the array topology is most easily described structurally with the processing elements modeled behaviorally. This permits the study of the overall data flow through the array. As the design gets more specific, it is desirable to model individual parts functionally or structurally, detailing more explicitly how the system will be built. Then we can study how the processing elements work and their interactions with their neighbors. In addition, being able to mix these descriptions is valuable. For example, the processing elements might be modeled by a detailed structure while the boundary cells and the host interface are modeled behaviorally.

The HDL must be capable of describing the data operations and control, separately or intertwined. In other words, it should be able to define the data flow and let the control be implicit or define both explicitly. The capability to define both explicitly is important in many systolic arrays where the control is not centralized but rather is spread over all of the processing elements. Control can also be distributed in levels, where some of the local control is spread over the processing elements but where more global control actions might originate from a central controller. For these reasons it is important to have flexible methods of handling control and data flow.

Timing

All hardware operates in the real world and has some sort of time properties associated with it. At the more abstract levels a determination of the order of operations may be all that is required. However, at lower levels a very accurate real-time model of gate delays can be necessary. An HDL for systolic arrays must be able to handle timing over all levels of abstractions covered; general time constructs must be available at higher levels and more detailed ones at lower levels. An HDL must be able to describe both the traditional synchronous systolic arrays and asynchronous variations such as the wavefront processor[3].

An HDL for systolic arrays must have powerful methods of describing concurrent activities. It must straightforwardly describe simultaneous events and the dependencies between them. This is especially critical in systolic array descriptions because they are based on highly parallel, pipelined computations that can have numerous levels of concurrent activities, both at the array level and within the processing elements.

Language provisions

Syntactic. In general the language ought to be clear, understandable, straightforward to learn, and easy to remember. It should be constructed coherently from a regular basis rather than being a collection of special cases. Since it is a medium of communication, it must be readable and writable by people. Similar things should be treated in a similar manner in the language.

Basic types. Because much of systolic array efforts are directed toward signal processing applications with heavy numerical emphasis, the HDL ought to support integer and floating point numbers of arbitrary sizes. This includes all the common arithmetic operations associated with these numbers. The HDL must also support basic binary data types and their associated logical operations to allow detailed logic descriptions of lower-level modules.

Language evaluations

The following sections evaluate four popular HDLs with respect to the above criteria. The languages covered are CDL, AHPL, DDL, and TOPS.

Computer Design Language (CDL)

Introduced in 1965,[4] CDL was one of the earliest HDLs and is probably the oldest still in use. CDL has evolved over the years into a simple yet fairly powerful design tool for the register transfer level of design. It is supported by a translator and a simulator.[5] Its simple structure and portable software have made CDL a popular language.

CDL is a nonprocedural language with global variables. CDL descriptions can describe a system at only one level. No subroutine facilities are provided. Structures such as registers and memories are explicitly declared at the beginning of a description from a set of nonextendible data types. Data paths between structures are defined implicitly by the specified data flow. Functional and control actions are described by built-in basic and user-defined operators. The structure of the control section is implicitly defined by the control operations it must perform.

CDL language range. CDL supports only a very narrow range of the levels of digital systems design. It is suitable for describing heterogeneous systems at the register transfer level. It provides no methods of data abstraction and only limited operations abstraction through user-defined functional operations.

CDL description structure. CDL provides no ability to modularly structure descriptions. Hence, hierarchical descriptions are not possible. It does not support separate definition and instantiation of modules. Each piece of hardware used must be explicitly described. Because all descriptions are on a single level, CDL cannot support mixed-level hardware descriptions. CDL also does not support parametric definitions.

CDL descriptive methods. CDL describes hardware functionally. It does not support structural or behavioral descriptions. Data structure is implicit in the register transfers and connections specified, whereas control structure is derived by conditional clauses on each statement. CDL has no capabilities to specify regular architectures other than explicitly defining every piece and all interconnections. Because it is nonprocedural, CDL is capable of describing the data and control parts separately and can describe decentralized control structures.

CDL timing. CDL is based on a simple model of time, supporting synchronous system description only. Each enabled statement is performed once on each clock cycle. CDL does support the description of concurrency. Its nonprocedural nature makes the description of parallel operations very straightforward.

CDL language provisions. CDL is a simple language that is easy to understand. It is internally consistent and has a common syntax. Only integer and binary data types are supported. Operations on these data types are fairly extensive.

CDL is a simple, straightforward language. This is both its strength and its weakness. CDL's inability to structure its descriptions modularly, along with its inability to differentiate between a module definition and instantiation, severely limits the class of problems it can address. The nonprocedural nature of CDL has both advantages and disadvantages: On the plus side, it allows for explicit definition of both control and data structure, which can be useful in distributed architectures such as systolic arrays. On the negative side, this nonprocedural method can be cumbersome when it is not necessary to define the control flow explicitly.

A Hardware Programming Language (AHPL)

AHPL, developed in the early 1970s,[6] is a procedural language aimed at the register transfer level of digital design. It is supported by both a simulator and a hardware compiler.[7] This evaluation focuses on the second version of this language, AHPL II, that is supported by the simulator and compiler.

AHPL is based on a subset of the programming language APL. The selection of the APL subset was determined by which features suggested a unique hardware realization. APL's notation and control syntax is carried through to AHPL. APL vector and matrix notation is extended to bit vectors (registers) and bit matrices (memories).

AHPL can describe a system at only one level. Descriptions can be broken up into modules, but a module cannot be defined in terms of other modules. No subroutine or instantiation facility is provided. Each module used must be explicitly defined. The exceptions to this are combinational logic units, which are memoryless and may be defined once, like a function, and used in a number of places. Structures such as registers and memories are explicitly declared at the beginning of a module description from a set of non-extendible data types. Data paths between structures are defined implicitly by the specified data flow. The control structure is defined implicitly by the control sequence of the AHPL statements.

AHPL language range. AHPL is aimed at the register transfer level of digital description. Recent work has extended this down to the gate level to some degree. AHPL is inappropriate for description at any higher levels. AHPL does not support data abstraction and only supports operation abstraction to the degree that combinational logic units can be defined. The current version of the simulator does not even support this amount of abstraction, as it does not handle CLUs.

AHPL description structure. AHPL does support the partitioning of the system description into modules. It does not support hierarchical design because modules cannot be defined in terms of other modules. The language does not differentiate between module definition and instantiation. Each module used must be separately defined. The language cannot support mixed-level descriptions because it provides only a single level of description. AHPL does not support parametric definition of either modules or units.

AHPL descriptive methods. AHPL provides only a functional mode of description. Hardware entities are defined in declarations, but the interconnections are implied by the control sequence. No constructs for the description of regular structures are provided. The compile-time operations of the CLUs offer some aid in the description of recursive and iterative combinational structures but are very awkward to use. Because the definition/instantiation concept is not supported, each element in a systolic array has to be explicitly defined. Mixed mode descriptions are not possible since AHPL provides only the functional descriptive mode. The language does provide adequate descriptive methods for control, but these are in an APL syntax and can be difficult to decipher. AHPL does not support decentralized control structures very well. AHPL descriptions imply a central synchronous control structure that generates the necessary control signals. Describing a set of linked, independent finite state machines, such as might be found in a systolic array, is difficult. The AHPL control structures must be bypassed, and the states and state transitions of each machine must be described as if one were describing data transfers.

AHPL timing. AHPL contains no general time model and supports mainly synchronous system description. It provides some support for asynchronous system description with one-shot entity types and unclocked transfers. However, most of this is directed at the specification of asynchronous interface between internally synchronous modules. AHPL supports the description of concurrent activities fairly well. Each control sequence step can have multiple events happening in a clock period, providing local concurrency. Global concurrency is supported by different modules operating in parallel.

AHPL language provisions. AHPL is a fairly simple, consistent language. It supports no numeric types, only binary types. The operations provided for binary types are reasonably complete.

AHPL's greatest weakness is its lack of definition/instantiation support. A further flaw is its lack of data and operations abstraction mechanism. For describing heterogeneous hardware at the register transfer level it is adequate, but for describing highly parallel, regular digital systems across the spectrum of abstraction it is inadequate.

Digital Design Language (DDL)

DDL was introduced in 1968[8] and is supported by a translator[9] and a simulator[10]. DDL is based on the concept of block-structured automatons or FSMs. Hardware entities are called facilities and have scope over blocks, much as variables in ALGOL do. Facilities are accessible only within the blocks in which they are declared. By convention, the system block contains only facility declarations that, in effect, function as system global facilities do. DDL also provides operator blocks that are like function definitions in programming languages and segment blocks that are similar to the subroutine constructs of FORTRAN or the procedures of ALGOL.

DDL language range. DDL is a powerful and complex language that can cover a wide range of descriptive levels. When its basic nonprocedural constructs are used, it can explicitly describe data flow and control in the same manner that CDL does. By defining automatons and using state abstractions, modules can be defined procedurally, as they are in AHPL. By taking advantage of the block structure and the subroutine-like segment facilities, one can extend DDL definitions to more abstract levels than those possible with CDL or AHPL. DDL also supports data and operation abstraction of a sort with its ability to define complex facilities in terms of simple facilities.

DDL description structure. DDL allows hierarchical modular descriptions. It provides rudimentary support for the concept of separation of module definition and instantiation, although these constructs do not seem to be regularly used. Because of the modular hierarchy and wide range of DDL, mixed-level descriptions of systems can be written. In a single system one automaton could be described in great detail, with all of the control paths explicitly defined and with all operations of a primitive logic nature, while an adjoining automaton could be described as a higher-level FSM, with outputs determined by inputs and current state. DDL does not support the concept of parametric module definition, so it is not possible to define families of similar modules.

DDL descriptive methods. DDL is mainly geared toward functional definitions, much as CDL and AHPL are. However, its power makes both behavioral and structural definitions possible, although not straightforward. The division of a system into automatons is strongly structural, as is much of the block structure of the language. The specification of operations as a FSM is to a degree behavioral. Constructs are provided for specifying regular structure, both implicitly through staggered array connections and explicitly through a structural iteration construct. DDL also supports decentralized control structures very nicely, allowing each automaton and, to a degree, each block to function independently. DDL has a good set of control operations and conditionals that support both procedural and nonprocedural control descriptions.

DDL timing. DDL supports the description of concurrency on both the local and the global levels. Each statement can be given a unique conditional under which it occurs, and larger blocks can be set up to function in parallel. DDL supports both event-driven and cycle-driven systems. Its asynchronous latches and transport delay constructs allow it to model lower-level asynchronous circuits, while its registers and periodic clocks support synchronous systems. DDL supports multiple multiphase clocks and is able to mix clocked and unclocked descriptions.

DDL language provisions. DDL is a complicated language. It uses a very large, unique, and nonstandard character set. In addition, DDL has some unusual delimiter conventions. DDL's basic premise is fairly coherent, but its syntactic implementation is not. Many primitive objects are special cases of other primitive objects, and objects of the same class (e.g., facilities) have numerous and differing restrictions on how they can be

used. The semantics are reasonably unambiguous--but complex. DDL supports a rich set of primitive operators for binary types. It supports a more limited set for integer (not including multiplication) and does not support any floating point types.

DDL is one of the more powerful HDLs. Its capabilities include most of what is required for systolic array design and description. However, DDL also includes many features, aimed at general-purpose digital designs, that are not really necessary for systolic arrays. DDL suffers from a nonstandard character set and inconsistent syntax. In general, it does not seem to be a regular language but rather one of many exceptions and special cases.

Instruction Set Processor (ISPS)

The ISPS computer description language is based on the Instruction Set Processor (ISP) notation developed for the description of computer instruction sets. ISP, originally an informal notation, was formalized into the ISPL language and later upgraded into the current ISPS language[11] and simulator.[12]

The original intent of ISP was to aid in the specification and description of instruction sets. The language was used to produce a behavioral description of the functioning of a CPU, as seen by the machine language user. ISPS has been expanded to allow its use in many applications of machine description languages, even down to the register transfer level. ISPS supports only behavioral and functional descriptions; it has no facilities to describe the structure of the physical hardware itself.

An ISPS description consists of a set of entities (registers, memory, etc.) and an algorithmic description of the behavior. ISPS is block structured in the spirit of ALGOL. It handles concurrency at both the statement and the block levels. At the statement level all statements are by default concurrent with any necessary ordering explicitly described by the "next" statement. Statements can be blocked together, and these blocks can be executed concurrently or sequentially.

ISPS language range. ISPS is a powerful language and can cover a wide range of the higher descriptive levels. Its lack of structural description limits its usefulness at levels below behavioral register transfer descriptions. It supports some limited data and operation abstraction, allowing function-like definitions.

ISPS description structure. ISPS supports modular hierarchical descriptions. Descriptions can be of mixed levels, with modules described at differing levels of detail. ISPS does not support separate definition and instantiation of modules or parametric module definition.

ISPS descriptive methods. ISPS supports only behavioral and to some extent functional descriptions of hardware. Mixed behavioral and functional mode descriptions are possible. For these modes ISPS provides an excellent collection of control primitives. ISPS has no special provisions for the description of regular architectures.

ISPS timing. ISPS offers little support for timing [illegible] t has only a very general primitive in which one can specify the average [illegible] ution time for a block. Although ISPS is based on an inherently synchro[illegible] t the statement level, it does not really support either synchronous or asynchr[illegible] gn explicitly. ISPS descriptions are really a level of abstraction above that p[illegible] lar issue. At this abstract level ISPS does support the description of concurrency at both the local and the global levels.

ISPS language provisions. Syntactically ISPS is reasonably clear and straightforward. It has been used with great success to describe instruction sets of numerous computers. The language has a simple structure, and its syntax is regular, predictable, and readable. ISPS does have some ambiguities in its semantics. It supports integer but not floating point numeric data types. For integers, it provides all of the necessary primitive operations. ISPS supports binary types and a rich set of logic operators for them.

ISPS is a very good language for its intended applications, but it is not well suited for systolic arrays. Its main drawback is its lack of support for structural description. Almost as important is its inability to define and then instantiate parameterized (or even nonparameterized) modules. In perspective, it is easy to see why ISPS is not well suited for systolic arrays. Its driving rationale has been the description of the operation of programmable CPUs, whose purpose is the implementation of a specific instruction set. Much of the rationale behind systolic arrays is the specialization of the hardware to the task. Any generality in this kind of approach occurs not in the hardware, but at the design level above.

Conclusions

This paper has explored the utility of HDLs to describe systolic architectures. The rationale for this exploration can be described as follows: Systolic architectures are good candidates for the implementation of special-purpose hardware. The use of special-purpose hardware can increase the capabilities of systems. However, special-purpose hardware has a limited applications base, making design costs highly critical. Design costs can be minimized by the use of automated design tools. The first step toward automated tools is the development of an appropriate formal descriptive notation.

Systolic architectures can be described at numerous levels of abstraction, each requiring different types of descriptive tools. At this time, the level most lacking in a formal descriptive notation is the system architectural level. HDLs have been widely used in the description of traditional systems at this level; hence, it makes sense to investigate their utility for systolic architectures.

The use of traditional HDLs for the description of systolic systems at the system architectural level has been examined at Los Alamos. HDL evaluation criteria have been developed in the areas of (1) language range, (2) description structure, (3) descriptive methods, (4) timing, and (5) language provisions. These criteria were then used to evaluate the CDL, AHPL, ISPS, and DDL languages. None of these languages provided all of the features necessary for the description of systolic architectures. This points out the need for a language specifically oriented toward systolic architectures.

A Systolic Array Hardware Description Language (SAHDL) is currently under development at the Los Alamos National Laboratory.[13] SAHDL is being developed as an application language in the Consensus Language (CONLAN) family of languages.[14] CONLAN is a development methodology for HDLs based on the concept of abstract datatypes. CONLAN family languages are HDLs developed for specific applications but tied together by a common core syntax and common semantic definition system.

SAHDL attempts to provide all of the capabilities outlined in this paper, along with the basic capabilities needed for the complete description of digital systems. A translator is being written for the language, and a simulator driven by SAHDL is planned.

References

1. Kung, H. T. and C. E. Leiserson, "Systolic Arrays (for VLSI)," Sparse Matrix Proc., Society for Industrial and Applied Mathematics, 1979, pp. 256-282.
2. Speiser, J. M. and H. J. Whitehouse, "Parallel Processing Algorithms and Architectures for Real-Time Signal Processing," Proc. SPIE, Vol 298, Real-Time Signal Processing IV, 1981, pp. 2-9.
3. Kung, S. Y., R. J. Gal-Ezer, and K. S. Arun, "Wavefront Array Processor: Architecture, Language, and Applications," Proc. Conf. Advanced Research in VLSI, M.I.T., Cambridge, MA, 1982.
4. Chu, Y., Computer Organization and Microprogramming, Prentice-Hall, 1972, Chap. 1.
5. Bara, J. and R. Born, "A CDL Compiler for Designing and Simulating Digital Systems," Proc. Int. Conf. CHDL's Applications, 1975, pp. 96-102.
6. Hill, F. J. and G. R. Peterson, Digital Systems: Hardware Organization and Design, Wiley, 1978, Chap. 5.
7. Hill, F. J., R. Swanson, and Z. Navabi, "User Manual for AHPL Simulator (HPSIM2) AHPL Compiler (HPCOM) " Engineering Experiment Station, Univ. Arizona, 1981.
8. Duley, J. R. and D. L. Dietmeyer, "A Digital System Design Language (DDL)," IEEE Trans. Computers, Vol. C-17, No. 9, Sept. 1968, pp. 850-861.
9. Dietmeyer, D. L., "Digital Design Language Translator, DDLTRN 9.80," Univ. Wisconsin-Madison Report ECE-80-35, Oct. 1980.
10. Shah, A., "Digital Design Language Simulator," Univ. Wisconsin-Madison Manual, ECE Dept., Nov. 1980.
11. Barbacci, M. R., G. E. Barnes, R. G. Cattell, and D. P. Siewiorek, "The Symbolic Manipulation of Computer Descriptions: The ISPS Computer Description Language," Carnegie-Mellon Univ. Report CMU-CS-79-137, 1979.
12. Barbacci, M. R., A. W. Nagle, and J. D. Northcutt, "The Symbolic Manipulation of Computer Descriptions: An ISPS Simulator," Carnegie-Mellon Univ. Report, 1981.
13. Lewis, P. S., "Hardware Description Languages for Systolic Architectures," Master's Thesis, University of New Mexico, Dept. Electrical and Computer Engineering, July 1984.
14. Piloty, R., M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, and P. Skelly, CONLAN Report, in Lecture Notes in Computer Science, Vol. 151, G. Goos and J. Hartmanis (eds.), Springer-Verlag, 1983.