AN ABSTRACT OF THE THESIS OF

Mark R. Milden for the degree of Master of Science in Electrical and Computer Engineering presented on February 10, 1984.

Title: An Approach for Selecting a Language for Computer Hardware Description and Simulation.

Abstract approved: _____
V. Michael Powers

This paper corrects an apparent deficiency in the published information concerning Hardware Description Languages (HDLs) by introducing and discussing an approach for selecting an HDL for use in a design project. Although three classes of HDLs are discussed in this paper, High-Level Languages (HLLs), General Purpose Simulation Languages (GPSLs) and Computer Hardware Description Languages (CHDLs), the CHDL class has been most heavily emphasized. These have been emphasized because they have been found to be the most suitable for use as a digital design tool. The emphasis is realized by including a chapter reviewing CHDL fundamentals, by presenting several CHDL examples and by aiming the HDL selection approach toward choosing a CHDL. HLLs and GPSLs are appropriate selections for some digital design environments. Therefore these classes of HDL have also been discussed in this paper.

AN APPROACH FOR SELECTING A LANGUAGE FOR COMPUTER HARDWARE DESCRIPTION AND SIMULATION

by

Mark R. Milden

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed February 10, 1984

Commencement June 1984

APPROVED:

Redacted for Privacy

Associate Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented February 10, 1984

Typed by Jackie Morse for Mark R. Milden

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (con't)

LIST OF TABLES

AN APPROACH FOR SELECTING A LANGUAGE FOR COMPUTER HARDWARE DESCRIPTION AND SIMULATION

1. INTRODUCTION

The introduction of VLSI into the world of digital design has provided hardware engineers with a good reason to develop a new approach for designing digital hardware. In the days of SSI and MSI it was possible for a designer to keep track of the overall structure and behavior of a hardware design by using logic diagrams, Boolean equations and his own memory. As systems grow in complexity in the VLSI era, it becomes increasingly more difficult for a designer to keep track of a design, resolve design problems or communicate the design to others without the assistance of a computer. A new hardware design approach must include the ability for computer-aided design (CAD) work.

With many of the current design approaches it is popular to partition the digital design world into several layers of abstraction. The lower levels of this hierarchy (the circuit and logic levels) are well defined and in use throughout industry on design projects. The next higher level, the register transfer (RT) level, is a relatively new level with respect to its use in digital hardware design projects. However, there are several advantages to doing design work at the RT level. The purpose for moving up to this level is to abstract (i.e., disassociate from any specific entity) [WO77] the structure and behavior of the lower levels and focus the design onto a higher-level picture of a digital system.

A new design approach that utilizes the RT level of abstraction is beginning to gain acceptance with designers.

The design tools that are needed to allow useful work at the RT level are: a concise, yet precise, hardware description at this level and a method for performing RT level simulations of the hardware description. The need for a concise method of describing a digital design with thousands of gates is very apparent. This description must also be a precise one in order to reduce ambiguity in the design and allow for development of the software in unison with the hardware. The use of simulation aids designers by increasing their understanding of how a design will actually behave, and thereby allowing them to locate and concentrate on the critical areas of the design. Simulation will also reduce the number of prototypes that will need to be built, saving money in design costs and getting the product out the door in less time.

There are three classes of formal languages that can fulfill both the description and simulation requirements of the register transfer level design tools. These classes are grouped together into a set called Hardware Description Languages (HDLs). The three individual classes of languages that make up the set of HDLs are: the High-Level Languages (HLLs), the General Purpose Simulation Languages (GPSLs) and the Computer Hardware Description Languages (CHDLs).

HDLs provide a means of attaining computer assistance in hardware design. Thus, the use of an HDL-based design approach is an efficient method for dealing with the complexity of VLSI design

projects. An HDL design approach is realized by implementing an HDL on a host computer which then becomes an HDL-based design system. Several papers have been published that present the many advantages of using an HDL design system for digital hardware design [Br66, Br72, Ba75, Sh79a].

There are several implementation-related questions that need to be answered when developing an HDL-based design system. The purpose of this paper is to answer one of these questions. This topic question is:

HOW SHOULD AN HDL BE SELECTED FOR USE IN A DESIGN SYSTEM?

This is an important question to answer as it deals with the selection of the best available language for satisfying the requirements of a proposed design system. It is also a question that has not been well addressed in the current HDL-related literature. To answer this topic question a three-step approach for selecting an HDL will be presented.

The first step of this approach is to select one of three classes of HDLs as being best suited for a design task. This is done by comparing the advantages and disadvantages of using each class for hardware design purposes. The second step is to select a small group of languages that possess a majority of the technical features needed to make a language useful for the particular design environment. The third, and final, step of this selection approach is to develop an order of preference within the group of useful languages based on some practical considerations.

This approach will include making a selection from any of the three HDL classes. However, the CHDL class will be the most heavily emphasized because of the design industry's lack of familiarity with CHDLs and the languages' superior suitability for hardware description and simulation purposes. This emphasis will be carried out throughout the entire paper.

This paper has been written with two secondary purposes in mind. The first of these is to increase a designer's familiarity with CHDLs by reviewing some of the historical background, general language constructs and specific examples of this class of HDL. The second purpose is to compile a bibliography of CHDL-related literature to serve as a useful starting point for further research.

Chapter Two is devoted entirely to a review of CHDLs, as they are relatively new and not well known by designers. On the other hand, HLLs and GPSLs are older and thus better known by the design industry. This chapter provides information aimed at helping the reader to more familiarity with the fundamentals of CHDLs. It includes an examination of three early CHDLs, a look at the syntax of a general CHDL and description and examples of two popular CHDLs.

Chapter Three presents some of the languages that are available for use in hardware description and simulation projects. This is accomplished by discussing the advantages and disadvantages of using each of the three classes of HDLs and then examining specific languages from each class. HLLs and GPSLs are discussed first as they are already familiar to most designers. Only two example languages are presented for each of these classes because languages

within each class are very similar both in structure and use. CHDLs offer a wide range of structures and uses. No single language can be used to describe the entire class, so ten individual examples are presented. This section also introduces a four-dimensional CHDL space. This is used in Chapter Four to aid in the selection of languages based on technical considerations. Following the ten relatively detailed examples, 14 other languages are briefly discussed to provide a more complete listing of available CHDLs. All 24 CHDLs are summarized in Table 3, located at the end of this chapter.

Chapter Four introduces and discusses the HDL selection approach. This chapter is divided into four sections; the first two are aimed at all three classes of HDLs while the last two emphasize the CHDL class. The first section reviews some of the reasons for using an HDL for design purposes. Section 2 discusses the selection of an appropriate class of HDL for particular design environments. Sections 3 and 4 examine the selection of individual languages based on the fact that a CHDL is to be selected. An example of selecting an HDL for use in an academic environment is developed throughout this chapter.

Finally, Chapter Five gives a summary of the ideas presented in this paper.

2. CHDL REVIEW

While most hardware designers have had some hands-on experience with HLLs, very few have ever worked with a CHDL. To increase a designer's familiarity with this class of languages, several aspects of CHDLs will be reviewed. This chapter has been organized into three main areas, each presented in a separate section. The first of these sections discusses three "historic" CHDLs. Special emphasis is given to constructs that were introduced by these languages and have become common features of modern CHDLs. The second section uses many of these original constructs to develop a set of general features that can be used to characterize individual languages. The final section of this chapter will review two popular CHDLs with respect to the features presented in Section 2.2.

2.1. Early Languages

The languages from the first generation of CHDLs can be classified into three groups. This classification scheme is based on the approach used to develop a particular language. The first group includes those languages that were developed from scratch as dedicated hardware description languages. The second group includes the languages that were developed as high-level programming languages that had some applications in the hardware description area. The final group includes those CHDLs that were adapted from high-level programming languages and converted into hardware oriented languages with the addition of special data structures for hardware description.

This section will review one language from each of these groups. The languages chosen (RTL, APL and LOTIS) are usually referred to as the first CHDL to appear from each of the corresponding groups. For each language an explanation of the reasons leading up to the development of the language and a brief look at some of the unique features of the language will be given. The purpose of these introductions is to present some general features common to all CHDLs rather than to examine the languages in detail.

A. Introduction to RTL

Perhaps the earliest example of any CHDL is the Register Transfer Language (RTL). This notational scheme was developed in the 1950s and early 1960s by I.S. Reed. The RTL language was first presented in an ACM report in 1952 [Re52] and then used 10 years later in two textbooks on digital computer design [BL62,Ch62]. The philosophy behind the development of RTL was that all operations of a digital computer could be expressed by means of transfers between the registers of the system. RTL was created as a notational means for recording these transfers and thus became an early description language for digital computers.

In a design procedure based on the use of RTL the following three design phases were introduced: [BL62]

(1) The System Design phase - which outlines the overall configuration of the machine and the class of hardware structures to be used.

(2) The Structural Design phase - which describes the system in terms of the transfer relations between the registers.

(3) The Logic Design phase - which realizes the transfer relations by means of Boolean equations.

The RTL language was the key factor in making this digital design process work. In fact, the entire structural phase depended on the use of RTL to provide the register transfer descriptions. The descriptions at this level detailed the registers and the allowable transfers between those registers. A digital machine was regarded as a set of registers communicating with one another by means of the transfer operations allowed for that machine. Thus, the operation of a computer could be described by a set of transfer relations written in RTL.

The main advantage to using this type of descriptive technique for design work is in the notation. The RTL description is a shorthand way of designating a potentially complex set of Boolean equations, which may themselves specify a complex set of electronic operations. These transfer descriptions are intended to provide only the minimum amount of detail necessary to specify the machine at the register transfer level.

Once the RTL description is available, the individual transfers may be implemented by translating the transfer relations into Boolean equations and then by realizing the equations as logic components. Although RTL did not provide a means for automatically translating the register transfer descriptions into logic equations, an algorithm for completing this translation by hand was presented

with the RTL language [BL62].

The basic statement of the RTL language is the transfer statement. All of the various RTL statements are of this same format. The general form of this statement is:

CONTROL CONDITION / f(set of registers) ⟶ destination register

The control condition is used to allow for synchronous operations by requiring a clock-like condition signal to be set true before the register transfer is allowed to take place. The · / · symbol is used to denote the division between the condition section and the register transfer section. The actual transfer operation is some function of one or more registers with the result being sent to a single destination register for storage. A typical example of an RTL statement, shown below, reads as follows: when the signals f and p are both

$$f \cdot p \,/\, A \longrightarrow B$$

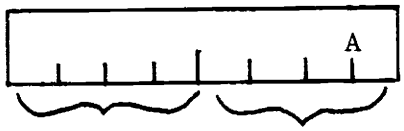
set true then the content of register A is transferred to register B.

The RTL language allows for only four elements that can store information. These four elements are memories, registers, subregisters and dependent registers. The most important of these is the register which is denoted by a capital letter. The memory unit is thought of as an array of registers and is usually denoted by a capital M. A particular word in the memory is referenced by placing the address of the word into a general purpose register that can serve as an address register. A subregister is a section of a register and is denoted by a two letter identifier followed by the register of interest as a parameter. A dependent register is a

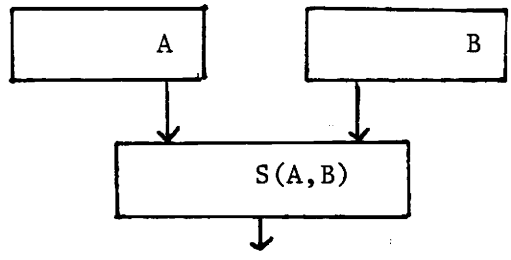
special type of register that is dependent on inputs from two or more registers, such as an adder. The notations for both the subregister and the dependent register are given below. Single bits of any of these elements can be specified by using a numeric subscript following the identifier for that element. The subscript represents the particular bit or bits of interest.

RTL has no provisions for declaring the word lengths of memory or the sizes of individual registers or subregisters. Declarations such as these, which are very important to provide a precise RTL description, are made by developing a block diagram of the object machine and including the necessary information on that diagram. When a designer wants to create an RTL description of some machine he must first develop the block diagram for that machine. This diagram should include identifying names for all of the registers and memories that will be referenced by the RTL description. The sizes of the individual registers (in terms of the number of bits stored by each) and the allowable data paths between the registers must also be indicated on the block diagram. Once the block diagram (or an equivalent type of documentation) has been completed, the RTL description can be easily worked out. The block diagram serves as the structural description of the system and the RTL description serves as a behavioral one.

One of the most important functions for a behavioral description of a digital computer is to describe the instruction fetch, interpretation and execution sequences used by a particular design. These sequences are basic to the functioning of a computer and the



Subregister Identifiers



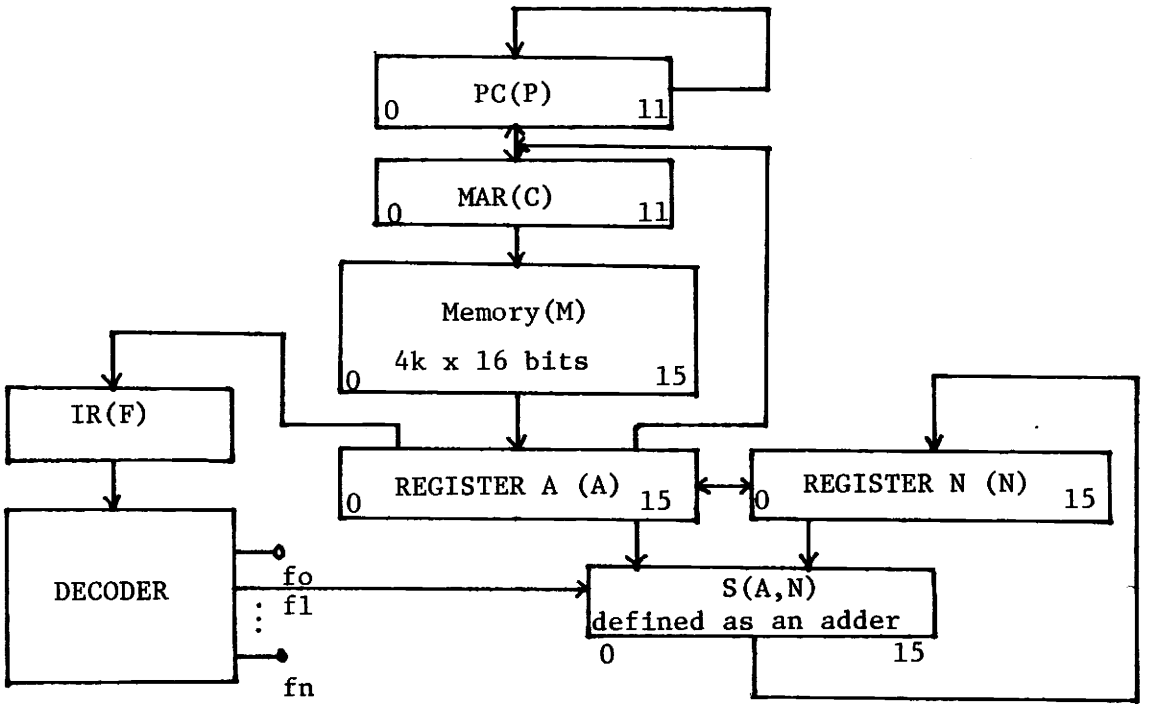
Dependent register: S(A,B)=A+B defines S(A,B) as an adder.

designer must have a detailed understanding of how these sequences behave. This understanding should include the register transfers for each of the sequences and a timing study to determine if any timing problems exist.

RTL has features to provide a means for performing such timing studies. Timing is carried out by defining an amount of time equal to one clock cycle and then writing all register transfers in reference to that amount of time. Multiple phase clocks can also be denoted in RTL.

Two sample sequences of the simple computer shown in Figure 1 have been written in the RTL notation. The first sequence (Figure 1b) describes an instruction fetch routine for the computer. The second sequence (Figure 1c) describes a typical machine instruction involving both an instruction and operand fetch.

The typical addition instruction for this computer might read as follows: add the number in memory location X to that in register A and store the sum back into register A. This instruction will take three clock cycles to complete: the first to obtain and interpret the instruction, the second to fetch the operand and the third to execute the addition. Notice that after the instruction



Assume 3-phase clock (ϕ_1,ϕ_2,ϕ_3)

ϕ_1 / M<C> ⟶ A

ϕ_2 / P + 1 ⟶ P

ϕ_3 / Ad(A) ⟶ C Op(A) ⟶ F

Figure 1b.

ϕ_1 / M<C> ⟶A

ϕ_2 / P + 1 ⟶P

ϕ_3 / Ad(A) ⟶C Op(A) ⟶F

$f_1 \cdot \phi_1$ / M<C> ⟶A

$f_1 \cdot \phi_2$ / P + 1⟶P

$f_1 \cdot \phi_3$ /

$f_1 \cdot \phi$ / S(A,N)⟶N

$f_1 \cdot \phi_2$ /

$f_1 \cdot \phi_3$ /

Figure 1c.

RTL Description of a Simple Computer
Figure 1

has been decoded, the control line f is asserted and ANDed with the clock phases to allow activation of the remaining statements. No information concerning the amount of time required to complete an operation is given in these examples. After the instruction has been decoded, the correct control lines will be asserted (f_1 in this case). This control line is ANDed with the clock to provide a new set of control signals for the next sequence of register transfers.

The RTL language introduced many major concepts to the family of CHDLs. The most important of these was the idea of, and a method for, describing the behavior of a digital machine. The language also allowed for timing considerations, individual bit operations, and concurrent activities. All of these features were first introduced by RTL and are still considered to be essential features of today's CHDLs.

Despite the initial promise of Reed's language there were several items lacking from it. RTL lacked a structural description that could be incorporated into the language descriptions, a method for describing iterative sequences (i.e. branches), a method for decoding instructions efficiently and a method for describing complex addressing modes. Several second-generation CHDLs were developed, based on the RTL language, to correct these faults [Sc64b,Ga62,Pr64,St70].

B. Introduction to APL

The earliest example of a programming language that was used for hardware descriptions was A Programming Language (APL). APL

was developed in the early 1960s by K.E. Iverson [Iv62a,Iv62b]. The original purpose of APL was to provide an effective notation for the description of programs for digital computers. A new type of language was needed to move away from the lengthy English and imprecise flow-chart descriptions popular at the time. This new language needed to be precise, concise, provide symbols that were similar to the types of data they were to represent and be independent of any particular data representation. While this language was developed for program descriptions it possessed many of the same features desired in a hardware description language.

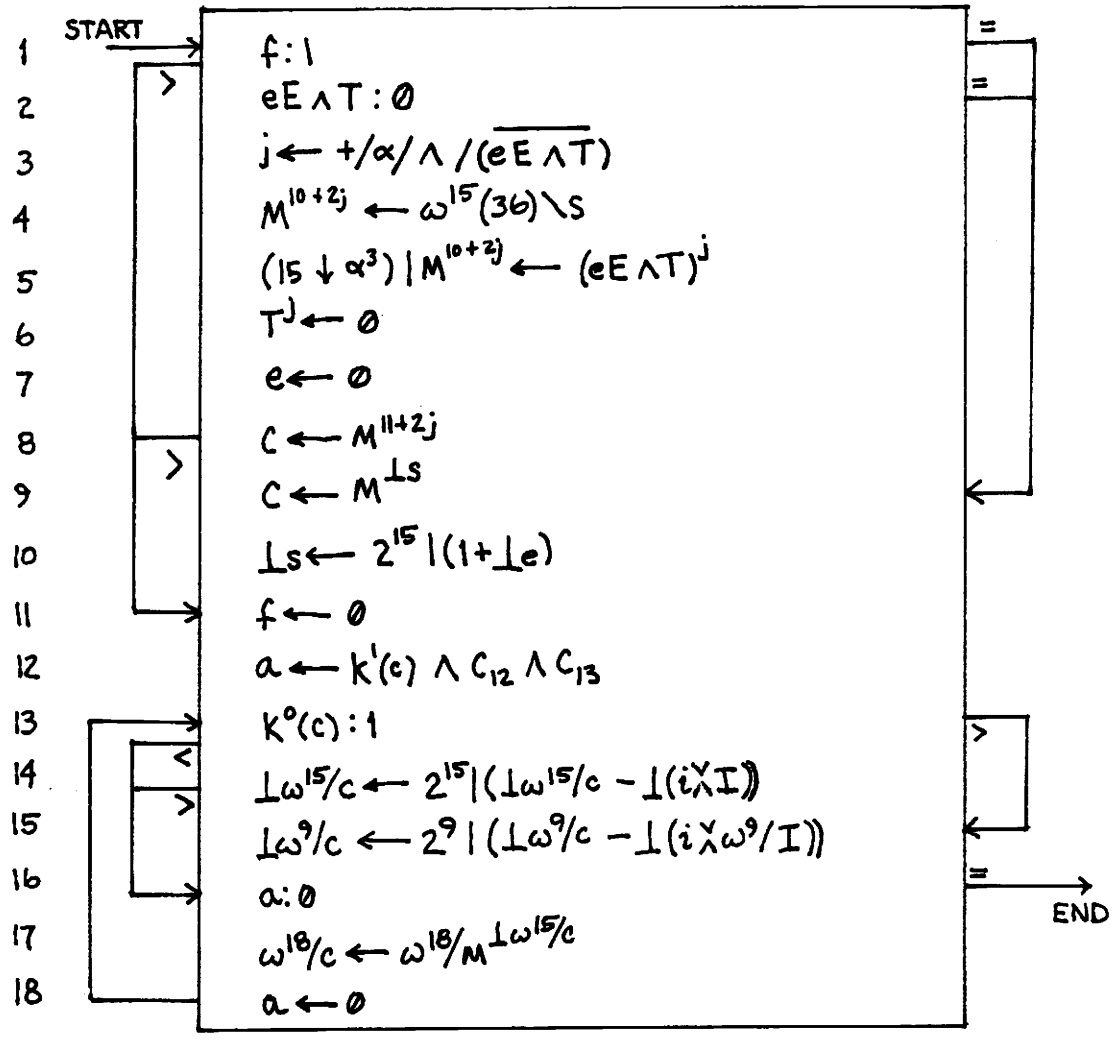
In the design and development of a computer it is important to have a very precise description of the system so that hardware and software designers can work toward their common goal without needing to know the details of what the other groups are doing. This can be accomplished by using a language that fits the needs of both design groups. APL was proposed as such a language at a conference in 1962 [Iv62b].

If a program for a digital computer is considered to be a sequence of instructions and if those instructions are considered to be a sequence of register transfers, then the method of applying APL to hardware descriptions comes to light. APL deals with register transfers by using one and two dimensional arrays (to represent registers and memories) and by representing data in binary notation. In APL, a single register transfer description is called a microprogram statement and a sequence of these is called a microprogram (which is equivalent to an instruction).

The APL language provides a rich set of data operators, an efficient facility for handling arrays of various sizes and an indexing facility. This made the APL language popular for hardware descriptions. Unfortunately, APL was also developed with conciseness in mind which usually lead to some long and complex statements. This fact combined with APL's non-standard precedence algorithm (right to left) makes APL descriptions hard to read. As an example of this an APL description of a complete instruction fetch microprogram is provided in Figure 2. There are several good sources explaining APL statements and data structures from a programming language approach [Pa76a,Le78,Iv62a].

In addition to providing hard-to-read hardware descriptions, the APL language has some other drawbacks. The most obvious of these is the inability to describe any parallel system. Other drawbacks include a lack of timing capabilities, no way to declare register sizes and a non-block structure. These problem areas have been accounted for in newer CHDLs that are based on the APL language [HP73].

APL was an important step in the development of CHDLs as it increased the number of data operators available beyond the set provided by the RTL language. At this point designers had two languages to work with. Designers preferred the RTL format and readability and the APL data operators and array handling capabilities. These two areas were combined with the introduction of the LOTIS language.



Lines 2-8 Channel trapping

Lines 9-10 Instruction fetch

Lines 11-18 Instruction preparation (with indirect addressing)

APL Description of Complete
Instruction Fetch
Figure 2
(taken from example presented in Iverson, 1962a)

C. Introduction to LOTIS

A third example of an early CHDL is the LOTIS language that was introduced by H.P. Schlaeppi in 1964 [Sc64a]. LOTIS is a formal language for describing the LOgical structure, the TIming and the Sequence of digital machines. This was developed to provide the designer with a precise notation for expressing both the structural and the behavioral properties of a digital design. LOTIS was the first CHDL to provide for structural description facilities [Sc64a].

The development of the LOTIS language was based on the following five goals:

(1) The language should be able to describe both structure and behavior.

(2) The elements of the language should correspond directly to the hardware units they are to represent.

(3) The notation of the register transfer level should abstract the details of the lower design levels.

(4) The description of a machine using the LOTIS language should have a hierarchical structure.

(5) Timing and concurrent events should be represented by this language.

The LOTIS language was developed from the ALGOL programming language. LOTIS combines the structure of ALGOL with several new data types and operators to create a hardware description language that meets each of the five goals. This development approach differentiates LOTIS from RTL which was developed from scratch and from APL which was developed as a high-level programming language.

LOTIS was an important step in the development of CHDLs since it effectively combined the best features of the earlier two languages. The type of easy-to-read statement formats and the timing capabilities presented by the RTL language were incorporated into LOTIS. The improved and varied data operators provided by a programming language (ALGOL rather than APL) were also used for the LOTIS language.

The description of a machine in this language is composed of a declaration section and a procedure section. The declarations are used to define the structure, logical and temporal properties of hardware units and also to assign names to those structural units. The procedural section is used to describe the behavior of the machine in terms of the declared hardware units.

The behavioral elements of LOTIS are register-to-register transfers. Each of these transfers is accomplished via a data path. These data paths, in combination with the declared hardware units, detail the structure of the digital machine. The data paths are defined by assignment statements in the procedural section of the LOTIS description. A group of these assignment statements makes up a sequence which is equivalent to an instruction.

Timing in a LOTIS description can be described either explicitly by associating a transfer timing delay with an assignment statement, or implicitly by using a delay time that was declared for the particular operator occurring in the assignment statement. The latter mode allows for descriptions of asynchronous events in an efficient manner. Timing can also be expressed by assuming time

intervals for each statement (such as one clock cycle for a simple register transfer); this form is useful for synchronous operations.

LOTIS was the first language to allow simulation of its computer descriptions on another computer. While the LOTIS description needed much additional overhead, including an initialization procedure and commands specifying what the simulation is to do, the first groundwork for simulating digital designs had been laid.

An example of an instruction fetch and an addition sequence for a simple computer are given in Figure 3.

From studying the reasons for developing the three previously described languages, a group of these that are common to each language stand out. These reasons or goals for CHDLs are:

1. To describe digital hardware of the register transfer level.
2. To provide behavioral descriptions of digital systems.
3. To make the behavioral description as precise and concise as possible.

These goals still remain today and are addressed by all of the languages presented later in this paper.

Many improvements in the next-generation CHDLs quickly made these first three languages impractical for industry use. With the added improvements, CHDLs have acquired some very complex characteristics. A discussion about some of the general characteristics of CHDLs will be presented in the next section.

```
cpu simple /
  m(12b,16);
  2(16) = [0-(4), ad(12)];
  n(16);
  c(12);
  p(12);
  f(4);
  s(16);

seq instrfetch /
  1: a := m(c) /
  2: p := p + 1 /
  3: c := ad; f : = op / fin

seq add /
  1: a := m(c) /
  2: p := p + 1 /
  3: n := a + n / fin
```

Declaration Section

Procedure Section

LOTIS Description of Simple Computer Sequences
Figure 3
(taken from an example in Schlaeppi, 1964a)

2.2. General Characteristics of CHDLs

A CHDL can be characterized as having an alphabet, a syntax, and a set of semantics. An alphabet is a set of symbols that are used in the language. The syntax is a set of rules to be followed for combining the allowed symbols into meaningful language statements. The semantics are a set of interpretations for all possible statements. Since each CHDL is developed by a different person or for a different purpose, these three areas will be different for each language. Thus the alphabet, syntax and semantics of a language can be combined to act as a fingerprint for that language.

To describe a CHDL in terms of its alphabet, syntax and semantics is a very logical and straightforward process. However a complete description of this form would be much too long for this paper's purpose. Thus a short list of general syntactical and semantical features has been selected for discussion. This list will primarily include areas that an engineer would use as comparison points when selecting a CHDL. The discussion of these points will also serve to show how CHDLs differ from other classes of HDLs. The set of features chosen are:

Alphabet

Representation of Constants

Identifiers

Primitive Elements

Primitive Operators

Statement Types

Complex Operators

Order of Execution

Structure

Organization of Description

The ALPHABET of a language is a set of symbols that can be used to create descriptions in that language. Nearly all modern CHDLs use a subset of the ASCII character set. The following characters are common to all text-oriented CHDLs. Only additions to this character set will be shown in the language reviews following this section.

upper-case English letters (A,B,C,...,Z)

lower-case English letters (a,b,c,...,z)

decimal digits (0,1,2,...,9)

common punctuation (,/./;/:/(/)/+/-/=)

It should be noted that while these characters exist in all languages they do not have standard meanings in all languages.

The method a CHDL uses to represent CONSTANTS varies greatly from language to language. The most common constants are the Boolean scalars (0 and 1) and the decimal integers (0,1,...,9). Other types of constants such as Boolean vectors, floating-point numbers characters and character strings are allowed by some languages. Integer representation is often allowed in several common bases, including binary, octal, decimal and hexadecimal. The use of such notational flexibility is useful in making descriptions in that language shorter and easier to read.

An IDENTIFIER is a symbol used to name a variable or primitive element. Typically an identifier is a single letter or a letter

followed by other letters or digits. The use of a complex identifer scheme is useful for allowing mnemonic variable names to be assigned.

The PRIMITIVE ELEMENTS of a language are those data structures that are used to hold and carry system information. These elements are called primitives because they are used as building blocks to construct higher level data structures and cannot themselves be broken down into lower level elements. For example, in most CHDLs a group of registers (a primitive element) can be combined to form a processor while they cannot be reduced to the gate or circuit levels. Primitive elements can be split into two classes depending on their storage capability. Some elements are memoryless, such as wires, buses and decoders. This type of element is used to transmit information. Other elements have memory, such as registers and memories, and are used to hold information. The complexity of primitive elements can vary from single-bit object (wires and flip-flops), to bit vectors (buses and registers), to bit arrays (memories). Examples of some primitive elements can be found in the review of the two CHDLs in Section 3.

The PRIMITIVE OPERATORS found in each language are closely related to the particular elements used in that language. The operators are entities that produce new information by translating bit patterns stored in the elements according to a set of assigned meanings. The meanings are usually well understood by designers (e.g. AND, OR, NOT, ADD) but also typically have different representations for each language. Operators can vary in both size and type of transformation performed. The size can range from

single-bit operations and register operations up to memory (array) operations. A set of CHDL operators is developed by selecting one or more operators from each of the following groups:

Logical (AND, OR, NOT)

Arithmetic (ADD, SUB)

Comparison (GT, GE, LT, LE, EQ)

Rotational (SHIFT, ROTATE, CONCATENATION)

Special (COUNT-UP, SWAP)

The various STATEMENT TYPES are formed by combining constants, primitive elements and primitive operators together. These statements are used to describe the operation of the system and the interconnection of the system components. There are five common types of statements used by most CHDLs.

DECLARATION

DATA TRANSFER

CONDITIONAL

SEQUENCING

TIMING

Declaration statements are used to describe the system components and their interconnections. Data Transfer statements describe the action of operators. These statements also include information specifying when a statement is active. Conditional statements are used to allow iterative and decision making capabilities to the system description. Sequencing statements allow for correct operation of an algorithm providing for serial or parallel activities and unconditional jumps. Timing statements are used to

explicitly declare timing characteristics for components or activities of the system. To describe hardware completely a CHDL should have facilities for expressing each of these types of statements, although some languages may combine two or more statement types to create a more general statement format.

COMPLEX OPERATORS are formed as a collection of one or more statements. These operators can take the form of functions, procedures and subroutines. The use of these complex operators can be an efficient way to increase the readability and conciseness of a description.

The ORDER OF EXECUTION can be used to partition CHDLs into two groups. The first group consists of non-procedural languages, where some or all description statements require an attached condition label to be set true before the statement can be executed. The second group consists of procedural languages, those CHDLs where statements are executed in the order in which they are written in the description. For non-procedural languages sequencing is performed by modifying control variables used in the statement label to enable or disable a statement. The control variables can be changed explicitly by a condition statement or a data transfer statement, or implicitly by an independent hardware element such as a clock or decoder. Sequential and concurrent actions are allowed with both classes of languages although the statement forms will be slightly different.

The STRUCTURE of a description is that part of a language that allows for the physical aspects of the design to be reflected into

the description. Early CHDLs used a single block format in which all variables were global. Newer languages use the multiple hierarchical block structures common to most programming languages. Block structures can convey some of the modular aspects of a design and allow the description of multiple levels of detail within the same description.

The ORGANIZATION of a DESCRIPTION includes any conventions that might need to be followed when writing a description in a particular language. These conventions can include such things as rules for naming elements, an ordering of which statements should occur before others, and required components to perform specific tasks. Every CHDL uses a slightly different set of conventions thus making some languages better suited for special types of hardware design projects. Careful consideration should be given to this aspect when selecting or using a CHDL.

The list of CHDL characteristics presented in this section can be used as a guideline for learning about a particular language. These characteristics begin with fundamental language constructs and then use them to build a foundation on which to develop the more complex features and structures of a language. This approach will be used in Section 3 to review two common CHDLs.

2.3. Review of Two CHDLs

This section will review two important CHDLs to help the reader gain an understanding of some of the features and structures present in this class of HDL. The languages selected for use here, the

Computer Design Language (CDL) and the Digital System Design Language (DDL), were chosen because of their easily accessible documentation and thus, their relative popularity (or vice versa). The partial hardware descriptions presented for each language (different systems are described) should be especially helpful in giving a designer a feeling for the structure of CHDLs.

The Computer Design Language [Ch65,Ch70,Ch72a,Ch72b]

This language was developed to describe the structure and behavior of a digital system. The ease of reading CDL descriptions and the language's simple structure have made this language one of the most popular CHDLs [Sh79a].

The ALPHABET of CDL includes the standard set of ASCII characters and the following non-standard characters.

The blank is not a part of the CDL alphabet but it can be included in CDL descriptions (as with most CHDLs) to increase readability.

CONSTANTS in CDL are from one of two categories. The first consists of the Boolean constants 0 and 1 which correspond to FALSE and TRUE respectively. The second category includes integers from several common bases. The radices 2, 4, 8, 10 and 16 are allowed in CDL descriptions. The following examples are all valid CDL constants.

377_8 $4F2C_{16}$ 35_{10} 42

A CDL IDENTIFIER (or variable name) may contain up to six characters, the first of which must be alphabetic. An identifier may be followed by a subscript to indicate a specific bit position,

a range of values or a value from an array. The following examples are valid CDL identifiers.

A , START , MAR(16) , ADDR1(0-11) , M(16,0-15)

All PRIMITIVE ELEMENTS used in a CDL description must be defined and assigned a name with one of several types of declaration statements. These statements have the form

Element , body,

where the "Element" represents the type of elements being declared and the "body" of the statement is used to list the names and sizes of hardware components included in a description. A list of the element types and a brief definition of each is given below.

FLIP-FLOP	A single bit storage element.
REGISTER	A set of synchronized flip-flops.
SUBREGISTER	A smaller section of a register.
CASREGISTER	A cascaded set of registers.
MEMORY	An array of synchronized flip-flops.
SWITCH	An external input device.
LIGHT	An external output device.
TERMINAL	A combinational logic network.
DECODER	A device to select specific output lines.
CLOCK	A clock.
DELAY	A delay element.

Several of these elements are of special interest because of the frequency of their use or special abilities they possess. The first of these are the REGISTER and MEMORY elements which are the most frequently used elements in CDL descriptions. The subscripts

of the following examples are used to denote size and bit positions.

REGISTER, ACC(0-15), MAR(0-11), BUF(5-1),

MEMORY, M(MAR) = M(0-1023,0-15),

The SUBREGISTER is a useful element type that was first explicitly introduced with CDL. A subregister is a part of an already defined register that needs to be accessed separately from the entire register. The name of the subregister begins with the name of the "parent" register followed by the mnemonic for the subregister given in parenthesis. The size of the subregister and the particular bits of the "parent" register is specified in a manner similar to the REGISTER declaration statement. Thus,

SUBREGISTER, ACC(HEAD) = ACC(0-7), ACC(TAIL) = ACC(8-15),

declares the subregisters ACC(HEAD) and ACC(TAIL).

Two other elements that first appeared with CDL are external input and output devices, the SWITCH and LIGHT elements respectively. Both of these elements are considered to be accessible by an operator. These elements can be used to simulate the external operations of a digital system, such as start-up and power-failure procedures. Switches can have one or more positions and similarly lights can have one or more light conditions. The following examples are valid CDL declarations.

SWITCH, POWER(ON,OFF), START(ON),

LIGHT, WARNING(YELLOW,RED,GREEN), POWER(GREEN),

The PRIMITIVE OPERATORS found in CDL are symbols that represent the functions performed by various logic networks in one clock cycle. The operators that are encountered frequently have been given

specific symbols and are shown in Table 1.

Two types of STATEMENTS are allowed in CDL descriptions. The first type includes the previously discussed declaration statements which are used to declare system elements. The second type includes the execution statements that describe a system's register transfer activities. The general format of an execution statement is:

/ label / $operation_1$, $operation_2$,..., $operation_n$,

The "label" is some logic function that must be evaluated as being true before the associated operations can be executed. Only the first operation is required; all others are optional. An example of an execution statement is given below. Conditional operations

/ F*P(1) / A ⟵ A add B, PC ⟵ Countup PC,

are also used in CDL descriptions. An example of an execution statement with a condition operation is also shown.

/ F*P(2) / IF (C=0) THEN (A ⟵ A sub B),

CDL allows for one type of COMPLEX OPERATOR. This is the BLOCK/DO statement combination which allows a set of operations to be labeled and referenced by a symbolic name. A BLOCK statement is used to label the set of operations in the declaration section of a description. Then a DO statement is used to reference that BLOCK in the executable portion of a description.

CDL uses a non-procedural ORDER OF EXECUTION mechanism. This means that every execution statement must have a label associated with it to determine the correct order of statement sequencing. It is useful to describe a sequence of statements in a procedural manner when the designer is not concerned with control signal

GROUP	SYMBOL	NAME	EXAMPLE OF USE
	⟵	TRANSFER	A ⟵ B
LOGICAL	'	NOT	A ⟵ A
	+	OR	A ⟵ A + B
	*	AND	A ⟵ A * B
	⊕	EX-OR	A ⟵ A ⊕ B
	⊙	COINCIDENCE	A ⟵ A ⊙ B
ARITHMETIC	add	ADD	A ⟵ A add B
	sub	SUB	A ⟵ A sub B
COMPARISON[1]	=	EQUAL	IF (A=)) THEN (A ⟵B)[2]
	≠	NOT EQUAL	IF (A≠0) THEN (B ⟵A)[2]
ROTATIONAL	shl	SHIFT LEFT	A ⟵ shl A
	shr	SHIFT RIGHT	A ⟵ shr A
	cil	CIRCULATE LEFT	A ⟵ cil A
	cir	CIRCULATE RIGHT	A ⟵ cir A
SPECIAL	countup	INCREMENT	A ⟵ countup A
	countdn	DECREMENT	A ⟵ countdn A

NOTES: 1. In some more recent versions of CDL the comparison operators .NE., .EQ., .GT., .LT., .GE., .LE. were added to the language.

2. These are conditional operations.

CDL Primitive Operators
Table 1

analysis. CDL descriptions can also be written in a procedural manner if some slight description modifications are made. These differences have been listed below for a procedural description.

1. Declaration statements for generating control signals are not needed.
2. The order of statement execution is dependent on the order in which the statements appear.
3. A GOTO statement is created to change the order of execution from the ordered form.
4. Labels are required only for the statements that are used with the GOTO statements.
5. A semicolon is employed to indicate the end of an execution statement.

CDL uses a single BLOCK STRUCTURE. This limits the suitability of using CDL to describe digital systems in a modular fashion. There are no subroutines allowed in a CDL description. The variables (elements) used in a CDL description are global to the entire description due to the single block structure. It is also not possible to declare special hardware elements, such as ICs, with CDL.

The ORGANIZATION of a CDL description consists of a list of declaration statements followed by a list of execution statements. There is no provision for partitioning the description into blocks of related statements (subsystems).

To sum up the CDL features discussed above, portions from a CDL description of a simple digital computer [Ch72a] have been

given in Figure 4. For comparison with other languages CDL is again summarized in Section 3.3.

The Digital System Design Language [DD68,Di71]

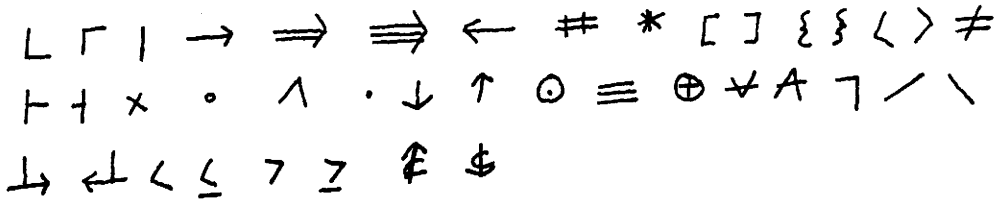
This language was developed to describe and simulate combinational and sequential logic networks by meeting four diverse goals. The first of these was to keep the language from being restricted to any single computer organization, timing mode or design procedure. This was important to insure that DDL remained a viable design tool as technology changed. The second goal was to make the language capable of describing digital systems at both an architectural "block" level and at a logical "gate" level. This feature will allow several groups of digital designers to use DDL and thus facilitate communication on a design team. The third goal was that the language should serve as the initial input for an automatic design process. The final goal was to have DDL description follow the actual system's structure as much as possible; if a system uses a multi-level block structure then the DDL description should be in a multi-level block format.

DDL can be a very useful tool for the design, documentation and simulation of digital systems. Descriptions can be made for both sequential and concurrent activities. The large number of operators, the conciseness of the language and the block structure of DDL make a good tool for dealing with systems in a complete and organized manner.

The ALPHABET of DDL includes the standard set of 70 ASCII characters and the following non-standard characters.

```
Register, R(0-23),                    $ buffer register
          A(0-23),                    $ accumulator
          C(0-14),                    $ address register
           :
Subregister, R(OP)=R(0-5),            $ OP-code part of register R
             R(I)=R(6),               $ indirect addressing bit
             :
Memory,   M(C)=M(0-32767,0-23),
 :
Clock, P(1-3),                        $ three-phase clock

/FETCH*P(1)/  C←—D, IF (G=0) THEN  F←—8),
/FETCH*P(2)/  R←—M(C), D   countup D,
/FETCH*P(3)/  F←—R(OP),C   R(ADDR),

/ADD*P(2)/    R←—M(C),
/ADD*P(3)/    A←—A add R, F←—9,
  :
/JMP*P(3)/    D←—R(ADDR), F←—9,
/JOP*P(3)/    IF (A(0)) THEN (D←—R(ADDR)), F←—9,
/SHR*P(3)/    A←—shr A, F←—9,
/CIL*P(3)/    A←—cil A, F←—9,
              END
```

Example CDL Description
Figure 4
(taken from example CDL description in Chu, 1972a)



The blank is not a part of the DDL alphabet but it can be included in DDL descriptions to increase readability.

CONSTANTS in DDL take the general form,

n R k

where "n" specifies the value of the constant in one of three numbering systems, "R" specifies the particular numbering system and "k" is a positive decimal integer giving the number of bits in the binary form of the constant. A constant can be written in binary, octal or decimal notation (R=B, O or D respectively). When a constant is expressed in binary the question mark, "?", may be used to indicate that the value of a bit is unknown or a "don't-care." The following examples are all valid DDL constants.

REPRESENTATION	BINARY EQUIVALENT
10D4	1010
1204	1010
0101B3	101
10?0B4	10x0

A DDL IDENTIFIER (or variable name) may contain up to eight characters, the first of which must be alphabetic. An identifier can be followed by a subscript to indicate a specific bit position, a range of values or a value from an array. The following examples are valid DDL identifiers.

A , START , MAR[16], ADDR1[0:11] , M[16,0:15]

ALL PRIMITIVE ELEMENTS used in a DDL description must be defined and assigned a name with one of several types of declaration statement. These statements have the form

<DT> body.

Where "DT" represents the declaration type, the body of the statement denotes the hardware elements that exist of type DT and the period signifies the end of the statement.

The declaration type consists of at least the first two letters of words that describe the hardware types. For example when registers are being declared, at least "RE" must appear inside the angle brackets. The body of the statement is used to either list the names and sizes of hardware components or to indicate how already declared units are interconnected. A list of the types of elements and a brief definition for each is given below.

<RE>	REGISTER	A set of synchronized flip-flops.
<ME>	MEMORY	An array of synchronized flip-flops.
<TE>	TERMINAL	A set of wires.
<TI>	TIME	A clock
<DE>	DELAY	A delay element.
<BO>	BOOLEAN	A combination logic circuit.
<EL>	ELEMENT	An off-the-shelf component.

REGISTER and MEMORY element types are the most frequently used elements in DDL descriptions; valid examples of these declaration statements are given below. Subregisters are implicitly declared

<RE> GO, MAR[16], IR[0:4]

<ME> M[0:4095, 16]

in a REgister declaration statement with the use of the concatenation operator. In the following example the subregisters IR and ADDR are declared as portions of register BUF.

<RE> BUF[16] = IR[4] o ADDR[12]

Another useful type of data element that first appeared with DDL is the ELEMENT declaration. This declaration provides a means of introducing a hardware block into a system description without indicating how the block is constructed or what functions it performs. In this statement the block is named and its output and input terminals are defined.

<EL> JKFF(Q1, NQ1: C, J1, K1)

It should be noted that connections to these "block boxes" may be defined and translated into Boolean equations, but simulation of descriptions that include these blocks are not allowed.

DDL contains a large number of PRIMITIVE OPERATORS to aid in developing clear and concise hardware descriptions. The terminals of the primitive elements serve as the operands of operations which express the interconnections and interactions between those elements. The operators determine the nature of these activities. A list of the frequently used operators is provided in Table 2.

There are several different types of STATEMENTS allowed in DDL descriptions. These can be divided into two groups. The first includes the declaration statements already described, while the

GROUP	SYMBOL	NAME	EXAMPLE
	=	CONNECTION	A = B (A is connected to B)
	←	TRANSFER	A←B (contents of B goto A)
LOGICAL	¬	NOT	A ← ¬A
	∨	OR	A ← A∨B
	∧, ·	AND	A ← A∧B
	⊕	EX-OR	A ← A⊕B
	⊙	COINCIDENCE	A ← A⊙B
	↓	NOR	A ← A↓B
	↑	NAND	A ← A↑B
ARITHMETIC	+	ADD	A ← A + B
	-	SUB	A ← A - B
COMPARISON	<, ≤	LT,LE	
	>, ≥	GT,GE	
	≐, ≠	EQ,NE	
ROTATIONAL	x→	SHIFT/CIRCULATE RIGHT	A ← c→ A
	←x	SHIFT/CIRCULATE LEFT	A ← ←e A
	(where x is either blank, 0, 1, c or e)		
SPECIAL	X	EXTENSION	A ← XA
	0	CONCATENATION	IR[3]=OP[0] X[2]
	\	SELECTION	V\A
	/	REDUCTION	V/A
	↑, ↓	COUNTUP, COUNTDN	A← ↑ A

DDL Primitive Operators
Table 2

second consists of the functional statements that are used to describe the behavior of a system. This second group includes Register Transfer, Connection, IF-THEN-ELSE, IF-VALUE and FOR statements.

The Register Transfer and Connection statements rarely appear without some condition that determines whether the operation is performed or not. Boolean expressions are used to represent these conditions. Identifiers in the expressions refer to registers, clocks and terminals previously declared.

The IF-THEN-ELSE statement is used to relate the activation conditions to the register transfer and connection statements. In DDL these statements appear as vertical lines enclosing a Boolean expression of the condition to be satisfied. The ELSE portion is expressed as a semicolon. A period is used to indicate the end of the statement. Thus,

$$F \longleftarrow |A|\ B\ ;\ C.$$

reads as follows: If A is true, then B is transferred to F, else C is transferred to F.

An IF-VALUE statement is used to describe the decoding of multivalued conditions. The general form of the expression is, ⌜ Boolean expression ⌊value 1 operation a ⌊value 2 operation b. If the Boolean expression has a value of "value 1" then operation a is performed where " ⌜ " represents "if" and " ⌊ " represents "then do." To show an example of this type of statement assume a three-way multiplexer, A, which must select one of three different registers.

```
<RE>   ACC[10] , MAR[10] , PC[10] , BUF[10].
⌈A   ⌊0 ACC ⟵ BUF  ⌊1 MAR ⟵ BUF  ⌊2 PC ⟵ BUF.
```

An optional ELSE condition can be added to the end of an IF-VALUE statement if desired. This type of statement can easily be thought of as a decoder.

When a designer needs to specify a parallel data transfer it usually involves the entire register, in which case no subscripts are needed. If a data transfer is valid for only certain subscript value or range of values, a FOR statement can be used. The general form of this statement is as follows;

{index variable = list of values} operations in terms of the index.

This statement is used to provide a more concise system description. As an example of this the following two equivalent examples are given.

```
{i = 1:3}  C[i] = C[i+1] · X[i].
```

```
C[1] = C[2] · X[1].
C[2] = C[3] · X[2].
C[3] = C[4] · X[3].
```

The DDL language allows for two types of COMPLEX OPERATORS. The first is the OPerator statement, which is used to define a block of combinational circuitry whose outputs will be used by one facility at one time and a different facility at some other time. This can be considered as a time-shared circuit that has been developed to reduce the amount of duplicate hardware in a design. A full adder could be described as an operator as shown.

```
<OP>  ADDER(SUM,CARRY).
      <TE>  A,B,CIN,SUM,CARRY.
      <BO>  CARRY=A·B+B·CIN+A·CIN
            SUM=A·B·C+(A+B+C)·¬CARRY.
```

In this statement the variables SUM and CARRY are local "dummy" variables that represent the output of the adder. The facilities declared by the terminal statement, which should be thought of as wires, are local to the operator block only.

The second type of complex operator is the IDentifier declaration. This statement is used when a series of operations is repeated often in a description. The <ID> statement gives a name to the series of operations and allows that name to be used in place of a long or complex set of operations. This statement is primarily used to reduce the amount of writing needed to make a DDL description.

The ORDER OF EXECUTION for DDL is dependent on the state transitions of subsystems called automatons. Since these transitions must be declared explicitly, DDL is a non-procedural language. This sequencing mechanism will be further explained in the next paragraph.

DDL is a BLOCK STRUCTURED language. It has been developed to accurately describe the modular structure of digital hardware designs. Many hardware systems can be considered to be a collection of semi-independent subsystems interacting with each other via some overall system facilities. This type of design structure is efficiently described in DDL with the use of the following four types of statements; SYstem, AUtomaton, STate and SEgment.

The overall system is declared with a SYstem declaration. This all encompassing block is used to provide all of the needed global elements and intercommunication systems. There is only one system declaration per description.

The subsystems that comprise the entire system are labeled with an AUtomaton declaration. A subsystem is typically a single finite state machine and it's private facilities. The body of an ⟨AU⟩ statement can contain facility declarations, operation statements and state and segment declarations.

A STate declaration is used to express the states of a finite state machine. Each state must have a set of operations to perform and specify a next state transition. A new operator is used to specify a state transfer. It is called the "go to" operation (⟶) and the only operand is the name of the next state to be activated.

If a group of states is divided into two or more groups of related activities they may be labeled separately with a SEgment declaration. An example of this would be a finite state machine that includes an input processing segment, a data manipulation segment and an output processing segment. An example of a DDL description using these four statements is shown in Figure 5.

The ORGANIZATION of a DDL description is based on the finite state machines used in a particular hardware design. The main system consists of declared facilities (RE, ME, TE, etc.) and one or more automaton. Similarly, the individual automaton consists of locally declared facilities and one or more states (with related groups of states split into segments if possible). This arrangement

provides for a very modular description of hardware. The timing in an automaton is carried out by IF-THEN-ELSE statements and state transitions. The timing mode can be synchronous, asynchronous or a combination of the two in any particular automaton. DDL allows for global and primitive facilities in its descriptions.

To show how the DDL constructs discussed in this section fit together in an actual hardware description, selected portions from a DDL description of a simple digital computer have been presented in Figure 5.

This section has developed specific information on two types of CHDLs. By using the construct discussions and the partial hardware descriptions, the features and structures presented in this section can be extended to develop an intuitive picture for other CHDLs introduced in the next chapter.

```
<SY>  EDC:  <TI>  P(1E-6).
            <ME>  M[0:1023,16].
            <RE>  IR[16]=OP[0:3]•IX[2] ADR[10],CAR[10],
                  ACC[16],MAR[10],....
            <AU>  CPU:P: <ID> Z=0D16.
            <ST>  IFL:| CLEAR|RUN←—0, CLEAR←—0, CAR←—0,...
                      | RUN |MAR←—CAR, ↯ CAR,—→1F2;—→1F1...
                      ⋮
                      EX: ⌈OP  ⌊0 |IN |M[MAR]←—INPUT, IN←—0,...
                               ⌊1 | ⅂OUT |OUTPUT←—M[MAR],OUT←—,
                                  ⋮
                               ⌊4 ACC←—M[MAR],—→IF1
                               ⌊5 ACC←—ACC+M[MAR],—→IF1
                               ⌊6 ACC←—ACC-M[MAR],—→IF1
                               ⌊7 M[MAR]←—ACC,—→IF1
                               ⌊8 ACC←—ACC.M[MAR],—→IF1
                                  ⋮
                        EXBIX:IN: M[MAR]  INPUT, ↯ ADR, IN←—0,—→EX.
...(end of ST,AU,SY)
```

Example DDL Description
Figure 5
(taken from example DDL description in Dietmeyer, 1971)

3. AVAILABLE HDLS

Before selecting an HDL for design purposes, it is important to be aware of what types of formal languages can be chosen to describe and simulate digital systems. This awareness could include knowing the advantages and disadvantages of using each type of language. The purpose of this chapter is to provide this kind of information for three classes of computer languages. Each class is capable of describing digital hardware at or above the register transfer level, thus they can be considered to be HDLs. The three classes of languages are:

1. High-Level Programming Languages (HLLs)
2. General Purpose Simulation Languages (GPSLs)
3. Computer Hardware Description Languages (CHDLs)

This chapter is divided into three sections, with a separate section being dedicated to each class of HDL. Each section includes a discussion of the advantages and disadvantages of using the particular class of HDL and specific examples of languages from that class. A glance at the names of the three types of HDLs may make it seem that the CHDL is the best suited for describing computer hardware. However, this is not always the case.

3.1. High-Level Programming Languages

HLLs were first developed in the 1950s and 1960s (FORTRAN in 1954, ALGOL in 1960 and many more since then) to abstract the details of implementing machine and assembly language programs. The

object of the HLL was to allow the programmer to concentrate on the algorithm itself rather than on its machine implementation for a particular type of computer.

The same type of situation now exists in regards to digital system design. Logic level descriptions are becoming so large and complex that they can no longer be easily understood. The design industry can benefit from using a high-level description language that concentrates on the design itself rather than on the logic needed to implement the design. HDLs have been suggested as suitable languages for hardware descriptions because they are sufficiently general to provide the constructs necessary to describe digital hardware functions at the register transfer level [RD83].

As a general class, HLLs have facilities for performing loops, making conditional decisions and involving subroutines and procedures. Each of these features is vital for the development of a precise hardware description.

Many HLLs possess special features that make them particularly attractive for hardware descriptions. For example: [Li77]

ALGOL has a block structure useful for control procedures.

APL has a concise vector notation useful for working with registers.

C has both a block structure and a wide variety of useful operators.

PASCAL has an ALGOL-like block structure.

PL/I has a structure useful for declaring variables in a modular fashion.

In addition to providing many of the language constructs needed to represent hardware, there are several practical reasons for using HLLs. The first of these is the availability of HLLs in the design environment. Nearly every company with a fair-sized computer has access to at least one type of HLL. The use of a language that is known to be available is a plus in the planning stages of developing an HDL based design system.

A second advantage is that HLLs tend to be well known by engineers. This means that no time would be lost trying to teach designers a new language. If a specialized design language were used, time would have to be spent familiarizing designers with the new language and its uses. Working out any system bugs could be a long and tedious process. The use of an already known HLL can reduce these problems.

Related to the familiarity of an HLL is the abundance of software support packages available. A large portion of a design system is implemented via software. Software programs for performing data transformations or developing graphics capabilities may already exist or if not could be quickly written.

Another valuable advantage to using HLLs is that they come with verified compilers [RD83]. When an HLL description is being run on a computer, the compiler acts as the simulator. Using a proven HLL compiler to simulate a system will eliminate the need to verify that the simulator for a specialized description language is working correctly.

A final and perhaps the most important reason for using HLLs is that they are relatively low cost. HLL software packages have been on the market for so long that their cost is considerably lower than that of the software for newer forms of description languages.

In spite of the practical advantages of using an HLL, there are several serious drawbacks to using one for the description of hardware. While many HLLs have some of the constructs needed to describe computer systems, none seems to have all of the ones that are needed. There are also several hardware properties that all HLLs have difficulty expressing.

The biggest of these problems is in trying to use an HLL to describe concurrent operations. Most programming languages were developed to describe algorithms as procedures which are executed in a sequential manner. Thus developing hardware descriptions where concurrent operations are commonplace is an ineffective use of many HLLs.

Another disadvantage is that descriptions in an HLL tend to be much longer than in the other classes of HDLs. This is an important consideration since the longer a description is the more difficult it is to read and understand. As an example of the difference in size between an HLL description and a CHDL description, the way in which a 6-bit register is declared will be examined. For a CHDL description the declaration would take one line of code. To declare a 6-bit register in FORTRAN requires the use of an INTEGER statement which sets up the register as an integer variable having the word length of the host computer. In the actual hardware no

number with more than 6 bits could be stored in the register. A FORTRAN description must insure the same property holds by masking and shifting the right-most bits so that no overflow conditions violate the conditions of the register. This can take up to four lines of FORTRAN code for each register declaration. Possible language codings for this example are shown below. Simplifications of this coding result in shorter but imprecise descriptions.

CDL FORM	FORTRAN FORM
REGISTER, A(0-5),	INTEGER A
	EQUIVALENCE (A, REALA)
	DATA MASK/ZFC0000/
	REALA=AND(MASK,A)

A final disadvantage of using an HLL is that once the description is complete it is less likely to resemble the hardware it describes than a CHDL description. It can be very difficult to obtain an intuitive feeling for how the hardware behaves from an HLL description. In addition, very little structural information can be described by an HLL (except as comment statements).

As a summary of an HLL's usefulness for describing computer hardware, the advantages and disadvantages will be again listed. The advantages are:

1. Several of the constructs needed to describe hardware are available.
2. HLLs are frequently found in industry.
3. HLLs are already known by a large percentage of the designers.

4. They have a relative low cost.
5. The compiler acts as a simulator.
6. There is a large amount of HLL software available.

The disadvantages are:

1. HLLs cannot be used to describe several important hardware features.
2. The descriptions are long and imprecise.
3. The descriptions don't easily reflect the structure of hardware.

Examples

Two languages have been selected to represent the class of HLLs. The first language is the scientific-oriented language FORTRAN which is one of the oldest and most frequently used programming languages. The second language is C which is a newer general purpose programming language that is becoming popular within the computer industry.

FORTRAN (FORmula TRANslation) [Ca83,Ni80] is a high level programming language that was first introduced by J.W. Backus in 1954. Several improved versions of this language have been developed, the most recent of which is FORTRAN77. This language is one of the most frequently used for the programming of scientific algorithms. Figure 6 presents a section of a hardware description in the FORTRAN language.

The C programming [HK82] language was developed at Bell Labs in 1972. It is a descendent of the ALGOL (1960), CPL (1963), BCPL (1967) and B (1970) programming languages. This language allows the specification of algorithms at a slightly lower level of detail

FORTRAN	C	CDL
INTEGER A,B,OP,ADDR[1]	int a,b,op,addr[1]	REGISTER,
EQUIVALENCE (A,RA),(B,RB),		A(0-22),
(OP,ROP),(ADDR,RADDR)		B(0-22),
DATA MASK1/ZFFFFFE00/,		
MASK2/ZFF000000/,		
MASK3/Z00FFFE00/		
RA=AND(MASK1,A)[2]	a=& mask1	SUBREGISTER,
RB=AND(MASK1,B)	b=& mask1	A(OP)=A(0-7),
ROP=AND(MASK2,A)	op=a & mask2	A(ADDR)=A(8-22),
ROP=ROP/256	addr=a & mask3	
RADDR=AND(MASK3,A)		
RADDR=RADDR/256,		
RA = COMPL(A)	a =! a	A←—A',
RA = AND(A,MASK1)		
A = A+1	a =+ 1	A←— countup A,
A = MOD(A,256)		
A = A-1	a =-1	A←— countdn A,
A = MOD(A,256)		
RA = AND(A*2),MASK1)	a = << 1	A←— shl A,
RA = AND (A/2),MASK1)	a = >> 1	A←— shr A,
RA = OR(A,B)	a =¦ b	A←— A + B,
RA = AND(A,B)	a = & b	A←— A * B,
RA = AND((A+B),MASK1)	a =(a+b)&mask1	A←— A add B,
RA = AND((A-B),MASK1)	a =(a-b)&mask1	A←— A sub B,

Notes: 1. Assume 32-bit word length for computer.
2. AND,OR, and COMPL are library functions that act in a bitwise fashion and return real values.

HLL Descriptions and Corresponding CDL Description
Figure 6

than does FORTRAN. A description corresponding to the FORTRAN example is also shown in Figure 6.

3.2. General Purpose Simulation Languages

The purpose of this class of programming language is to provide a facility for describing and then simulating nondeterministic discrete systems. These languages were developed to reduce the total time spent in designing, programming and testing simulation models. Typically these languages are used for the simulation of queue-like systems, such as a single-line, single-server queuing system used to simulate a bank teller assisting bank customers. Since these customers arrive at the bank in a nondeterministic manner a GPSL is ideal for describing this type of system.

This class of HDL is very useful for describing computer systems at the architectural design level. At this level, the system is often modeled as a nondeterministic system due to the randomness of high-level parameters. At the register transfer level where a computer design is seen as a deterministic system the use of GPSLs is not as effective. It is ironic that the advantages of using a GPSL at the architectural level are similar to the disadvantages of using a GPSL at the register transfer level.

Some of the advantages of using GPSLs are:

1. They specialize in describing nondeterministic systems.
2. They provide a wide variety of statistical tools for system analysis.

3. Their data types are especially good for defining random parameters.

All of these advantages make the GPSL ideal for high-level simulations. At this high level, system parameters include access times, delay times and priorities all of which are nondeterministic in nature. System elements include memories, processors and I/O units. These elements are often requested, held and then released, operations that GPSLs excel at representing. While these languages are very useful for high level simulation, precise hardware structural descriptions at this level are not possible.

Other disadvantages are related to using GPSLs to describe a digital system at the register transfer level. At this level, where a system's activities are specified by a pre-defined set of rules (instructions) this class of languages is not well suited for hardware descriptions. Some of the disadvantages of GPSLs for the register transfer level are:

1. At this design level most digital systems are deterministic in nature.
2. GPSLs provide a wide variety of statistical tools for system analysis of nondeterministic events. These tools are of little or no value to the digital hardware designer.
3. The description of a digital system should include structural as well as behavioral information. GPSLs provide no constructs to describe the large number of devices and interconnections that are present in a complex digital system.

4. Simulating a digital computer involves scheduling a large number of operations. Most GPSLs are not designed to provide this type of control.

Considering these four drawbacks and the lack of familiarity by most hardware designers these languages are a poor choice for register transfer level descriptions and simulations.

Examples

Two languages have been selected to represent the class of GPSLs. The first language is GPSS, which is a discrete stochastic simulation language. The second language is SIMSCRIPT II which is a general purpose programming language with built-in simulation facilities. The examples given for these two languages are not computer design related and are intended to show the commands and structure of the languages. A third popular GPSL which is not presented here is the SIMULA language [DM70,BD73].

GPSS (the General Purpose Simulation System) [Ma80,Sc82] is a very popular simulation language that has evolved from a block diagram approach of performing simulations. For this reason, GPSS is particularly well suited for describing digital systems at the architectural level. Figure 7 gives a short example of a GPSS program that is used to simulate a bank teller window.

SIMSCRIPT II [KV69,Ki81] is a general purpose programming language with simulation capabilities. SIMSCRIPT II has the ability to be used as an HLL in some situations and as a GPSL for others. This language achieves a block structure with an extensive use of subroutine-like constructs. In Figure 8 an example similar to the

```
* PROGRAM TO SIMULATE A DRIVE-IN WINDOW AT A BANK
* ARRIVALS - 1 TO 5 MINUTES, UNIFORMLY DISTRIBUTED
* PROCESSING - 1 TO 3 MINUTES, UNIFORMLY DISTRIBUTED
* TERMINATION CONDITION - 250 CUSTOMERS PROCESSED
*     SIMULATE
1     GENERATE      3,2         CUSTOMERS ARRIVE
2     ADVANCE       1           CUSTOMERS DRIVE TO WINDOW
3     QUEUE         WINDO       CUTOMER WAITS FOR WINDOW
4     SEIZE         WINDO       CUSTOMER REACHES WINDOW
5     DEPART        WINDO       REMOVE CUSTOMER FROM QUEUE
6     ADVANCE       2,1         TRANSACTION IS PROCESSED
7     RELEASE       WINDO       CUSTOMER DEPARTS
8     TERMINATE     1
*
      START         250         PROCESS 250 CUSTOMERS
      END
```

GPSS Program to Simulate Bank Teller Window
Figure 7
(taken from program provided in Maryanski, 1980).

```
 1  PREAMBLE
 2     THE SYSTEM OWNS A WAITING LINE
 3     TEMPORARY ENTITIES
 4        EVERY CUSTOMER HAS A N. TRANSACTION AND AN ENTRY.TIME...
     :
     :
15  END

 1  MAIN
 2     READ MIN.ARRIVAL, MAX.ARRIVAL, AVG.TRANSACTIONS
 3     LET WINDOW = IDLE
 4     LET N.ARRIVALS = 0
 5     LET N. DEPARTURES = 0
 6     LET MAX.WAIT = 0
 7     SCHEDULE AN ARRIVAL IN UNIFORM.F(MIN.ARRIVAL, MAX.ARRIVAL, 1)
       MINUTES
 8     SCHEDULE A DEPARTURE IN UNIFORM.F(MIN.TRANSACTION, MAX.
       TRANSACTION,1)
 9     SCHEDULE A STOP.SIMULATION IN 8 HOURS
10     START SIMULATION
11  END

 1  EVENT ARRIVAL
     :
     :
18  END
:
:
 1  EVENT STOP.SIMULATION
     :
     :
 5  STOP
 6  END
```

SIMSCRIPT II Program to Simulate Bank Teller Window
Figure 8
(taken from program provided in Maryanski, 1980).

GPSS example is presented.

3.3. Computer Hardware Description Languages

CHDLs are used to describe and simulate computer hardware at or above the register transfer level. These languages were initially developed as a method to omit inessential low-level details from a system description. This allowed designers to describe a system's behavior in a more algorithmic-like form. CHDLs have been expressly developed to provide language structures for describing hardware, thus a more precise and concise hardware description is possible with a CHDL than with an HLL or a GPSL. The goal of the early languages of this class, RTL and LOTIS (both of which have been discussed in Chapter 2), was to describe computer systems with precision and efficiency. Today the goals of the CHDL have expanded to include system simulation and automatic translation capabilities.

CHDLs are especially useful for describing hardware because they have been specifically developed to overcome the disadvantages of the other two classes of languages. CHDLs are able to express the concurrency, control activities and timing modes that are integral parts of a computer's description and are an improvement on HLLs in this respect. Some CHDLs are also able to describe a system at several levels of the design hierarchy and in this way are an improvement over GPSLs.

The advantages of using a CHDL are:

1. It provides a precise, yet concise hardware description.
2. Descriptions can be made at several levels.

3. Descriptions tend to be simple and easy to read.
4. Structural and behavioral features can be described.
5. Descriptions tend to resemble the hardware they describe.
6. It can represent pure hardware structures (in concurrency, etc.).
7. It can take a procedural or nonprocedural form.
8. The description is in a form suitable for simulation.
9. It can be used as the input stage to an automatic design system.

These advantages present a strong agreement for using CHDLs in design work and have been discussed in detail by several papers [Br72,Ba75,Su77,Sh79a,va79].

There are also some negative points to consider about CHDLs. These are:

1. Implementation of a CHDL based system can be expensive.
2. Time must be spent learning a new language and design system.
3. Design information may be abstracted too much for efficient design.
4. CHDLs are not as easy to simulate at the architectural level as GPSLs.

Although these disadvantages can be trivial in the long run, they should be considered when first implementing a CHDL based design system.

CHDLs can describe computer hardware in a more efficient manner than can either HLLs or GPSLs. With this in mind, CHDLs have become a popular area of research in academia and industry alike [Du67,Ch69,

Bl71,Ch76a,va77,Si81,EG76,St77]. As the number of research projects increase so it seems does the number of published CHDLs. Currently over 50 different CHDLs have been formally introduced and are available for use as design aids. There are three reasons that explain why such a large number of CHDLs have been developed. The first reason involves the design level hierarchy. Some CHDLs are aimed at the register transfer level, others at the programming level and still others at the architectural level. Another reason involves the dichotomy between structure and behavior in computer systems. Many languages were developed to emphasize the structure of a design while others are better suited to describe the behavioral details. A final reason deals with a language's facilities and constructs. No single CHDL seems to have all of the features to satisfy a researcher's picture of what the "ideal" CHDL should be. To correct this problem a new language is developed with all of the "right stuff" and thus another CHDL appears. To reduce the number of new languages that are being developed, a committee is writing a consensus language (CONLAN) that will provide a wide range of facilities in one vehicle [PB80a,PB80b,PB80c,PB82].

One problem with having such a large number of CHDLs available is that it is difficult to gain an understanding of several languages by studying the details of entire languages. Instead some form of simplification must take place that leads to a better understanding of the languages and their relationships to each other. One way to do this is to break the class of CHDLs into smaller groups with similar properties. A four-dimensional space has been developed

with which the capabilities of a CHDL can be characterized as a CHDL-space vector and plotted as a point. Points that lie near each other have similar properties. The concept of a CHDL-space is useful for characterizing individual CHDLs and is used in the next chapter as a selection tool.

The dimensions of this space are features of CHDLs that are possessed by all languages, but yet are features that are independent of each other. Each dimension must be capable of partitioning the CHDL-space into two or more smaller groups. For example, the feature that a CHDL can be used to describe hardware is not useful to partition CHDLs since all CHDLs can describe hardware. On the other hand, CHDLs can be partitioned into groups based on the hierarchical level at which a description is made. CHDLs can be partitioned by the following four language features:

1. level of description
2. type of description
3. sequencing mechanism
4. type of timing specification

These four features have been chosen as dimensions for the CHDL-space because of their relative independence to each other. The "level of description" dimension is based on the hierarchical design levels and as such can be considered a vertical partition. The other three dimensions involve more horizontally based divisions as they are not very hierarchical in nature.

An individual CHDL would be represented as a point in this four-dimensional space. To do this each of the coordinates must be

specified exactly. However, most CHDLs have capabilities that allow a range of points for any particular dimension. Thus a four-dimensional figure will be formed as a boundary to all of the points representing a particular CHDL. This can be considered to be a sub-space of the four-dimensional CHDL-space.

The first dimension involves the level at which a hardware description is made. There are four discrete levels at which a CHDL can be used. These are the: logic, register transfer, programming and architectural levels. Often a CHDL will be capable of describing hardware at a range of points. This range can mean expressing varying degrees of detail at the same hierarchical level or the possibility of descriptions at more than one level.

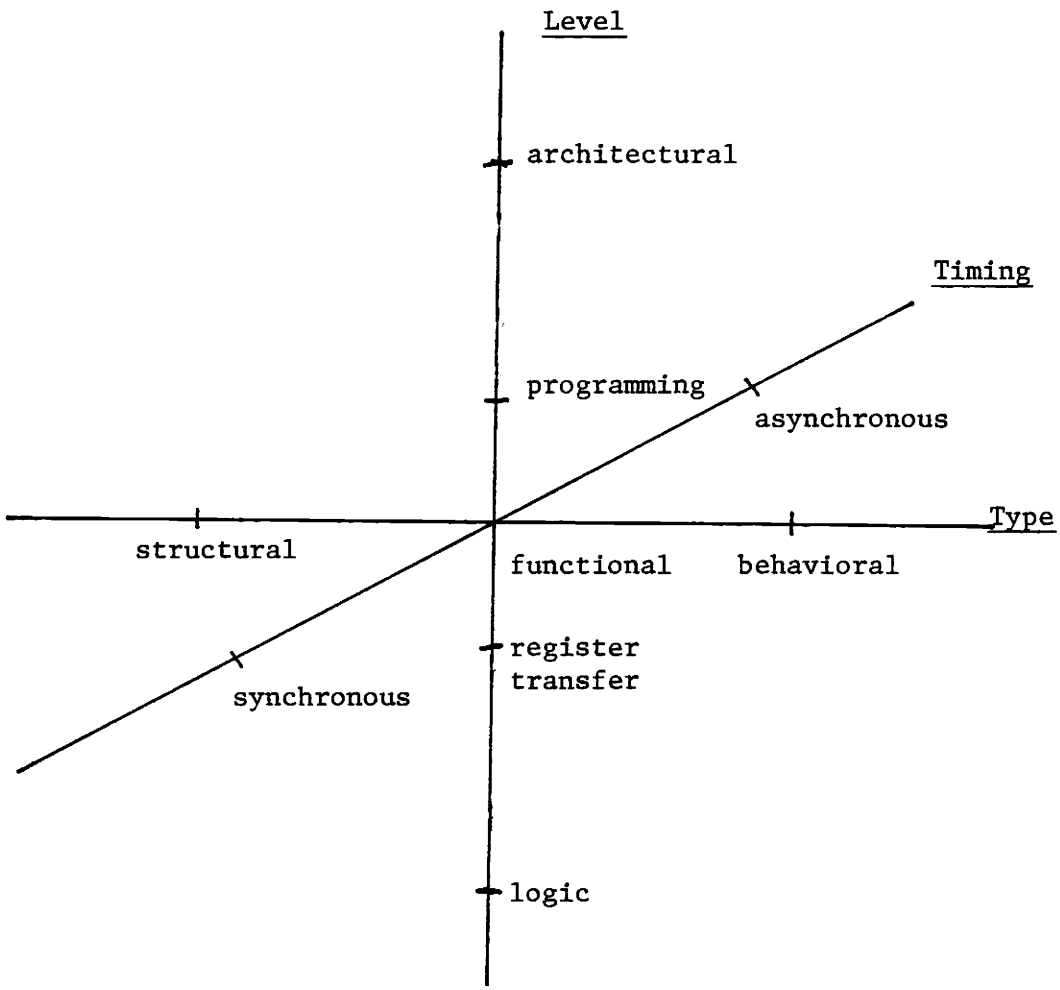
The second dimension involves the type of description a CHDL is geared toward expressing. This dimension has three discrete "types" of descriptions that a language can emphasize, structural, functional and behavioral. A structural description represents a system in terms of the actual hardware components. Interconnection of the structural elements is stressed in this type of description. A functional description suppresses low-level structural details so that a designer deals with registers and logic networks rather than individual flip-flops and logic gates. The action of a system is described as an algorithm involving these higher level components. With this type of description the actual data bits stored in a system's elements are of major concern. The behavioral level offers an even higher level of abstraction. This type of description is concerned with the order in which operations take place, but not

necessarily with the values of the data that is being manipulated by the digital system.

The third CHDL-space dimension involves the sequencing mechanism used by a CHDL. This mechanism can be either procedural, non-procedural or strongly non-procedural [ST81]. A procedural language uses the order that statements are placed in a description to determine sequencing information. Non-procedural languages are those block structured languages that provide explicit sequencing information for each block and rely on procedural sequencing internal to the blocks. Strongly non-procedural languages require an explicit sequencing label for every statement in a description. This dimension is usually specified as a range of points within one of these types of sequencing mechanisms.

The fourth and final dimension involves the type of timing specification used by a CHDL. This dimension has two discrete levels, the representation of either synchronous or asynchronous systems. Most languages are best suited for describing only one of these types of systems and allow a range of points within that one group. Although, some languages are capable of describing both types of activities.

The examples of this section will use the first, second and fourth dimensions from this space to develop a three-dimensional view of the example CHDLs. These three axes are shown as a three-dimensional space in Figure 9. This will present a simple view of how the examples fit into the overall CHDL-space and their relative locations in this space. A more detailed discussion about selecting



Three-Dimensional View of CHDL-Space
Figure 9

CHDLs based on these four dimensions is presented in Chapter 4.

Examples

CHDLs have been expressly designed to describe digital systems and they do so very well. Because they are the best-suited for this type of design work CHDLs have been more heavily emphasized than either the HLL or GPSL classes. In this section 10 different CHDLs will be discussed. With each of these examples a portion of a hardware description made in the particular language is given. These descriptions are intended to give the reader only a feel for how a description looks in that language; thus, no explanation of the hardware description is given. A reference that gives a detailed explanation of the hardware description is cited for each example. Following these presentations, several more CHDLs will be mentioned and references given to aid in the secondary purpose of this paper to serve as a bibliography for CHDL related information.

ADLIB

ADLIB (A Design Language for Indicating Behavior) [Hi79b,Hi80] was developed to describe the behavior of computer hardware at multiple levels of the design hierarchy. The language was intended for descriptions at the architectural, register transfer, logic and circuit levels. ADLIB is an extension of PASCAL so it possesses characteristics that allow hierarchical descriptions, software descriptions and individual component type specifications [Co81].

An early version of ADLIB was based on the SIMULA67 [DM70] programming language but it tended to create large and bulky descriptions with long execution times. The language was redesigned as

a PASCAL extension, which provided a much simpler and faster language for hardware descriptions. The ADLIB language includes several PASCAL constructs, including the PROGRAM statement, data structures and global procedures and functions. ADLIB extends PASCAL with constructs that are useful for hardware description such as control and data flow facilities. The following constructs have been added to this language [ST81].

The components of an ADLIB description communicate via a "net" which can be thought of as a set of interconnecting wires in a system. The ASSIGN statement is used to update these interconnections between hardware components. The form of the statement allows the update to be performed in either a synchronous or asynchronous manner.

ASSIGN < expression > TO < net name >< optional timing expression >.

The WAITFOR statement is a type of monitor that will keep an expression from being executed until a Boolean control expression is evaluated as true.

WAITFOR < Boolean expression >< control expression > .

The SENSITIZE, DESENSITIZE and DETACH statements are used to control the dependency of hardware components on their input connections (nets). These statements can be used to delay the operation of a component until the inputs have been updated.

SENSITIZE < component > .

The UPON statement is used for defining random or independent activities. This statement evaluates a Boolean expression every time a net in its checklist is updated. The associated statement(s)

is executed any time the Boolean expression is evaluated as true.

UPON < Boolean expression > <check list> DO < statement >.

The TRANSMIT statement is used to provide monitoring for the UPON statement. The TRANSMIT statement is activated when a "net" in the UPON statement is updated. This statement specifies the net and the value it is to assume (an expression) according to a scheduling time expression.

TRANSMIT < expression > TO < net name >< scheduling time expression >.

The PERMIT and INHIBIT statements are used for starting and stopping subprocesses in an ADLIB description.

This language has been combined with a second language, the Structural Design Language (SDL) to create a multi-level digital simulator system called SABLE [Hv79,Co81]. In the SABLE system ADLIB is used to express only the behavior of a digital system while SDL is used to describe the structure of the system [Co81]. The ADLIB language is used to construct small behavioral blocks called comptypes. An AND gate, JK flipflop , REGISTER and CPU are all examples of possible comptypes. The SDL language is used to describe how many components of a particular comptype exist in a design and how they are all interconnected. A "D" flipflop and a NOR gate comptype are given as examples of the ADLIB language in Figure 10. A CHDL-space representation of ADLIB is given in Figure 11.

AHPL

AHPL (A Hardware Programming Language) [HP73,Hi74] was developed as a tool to describe and simulate the behavior of digital hardware.

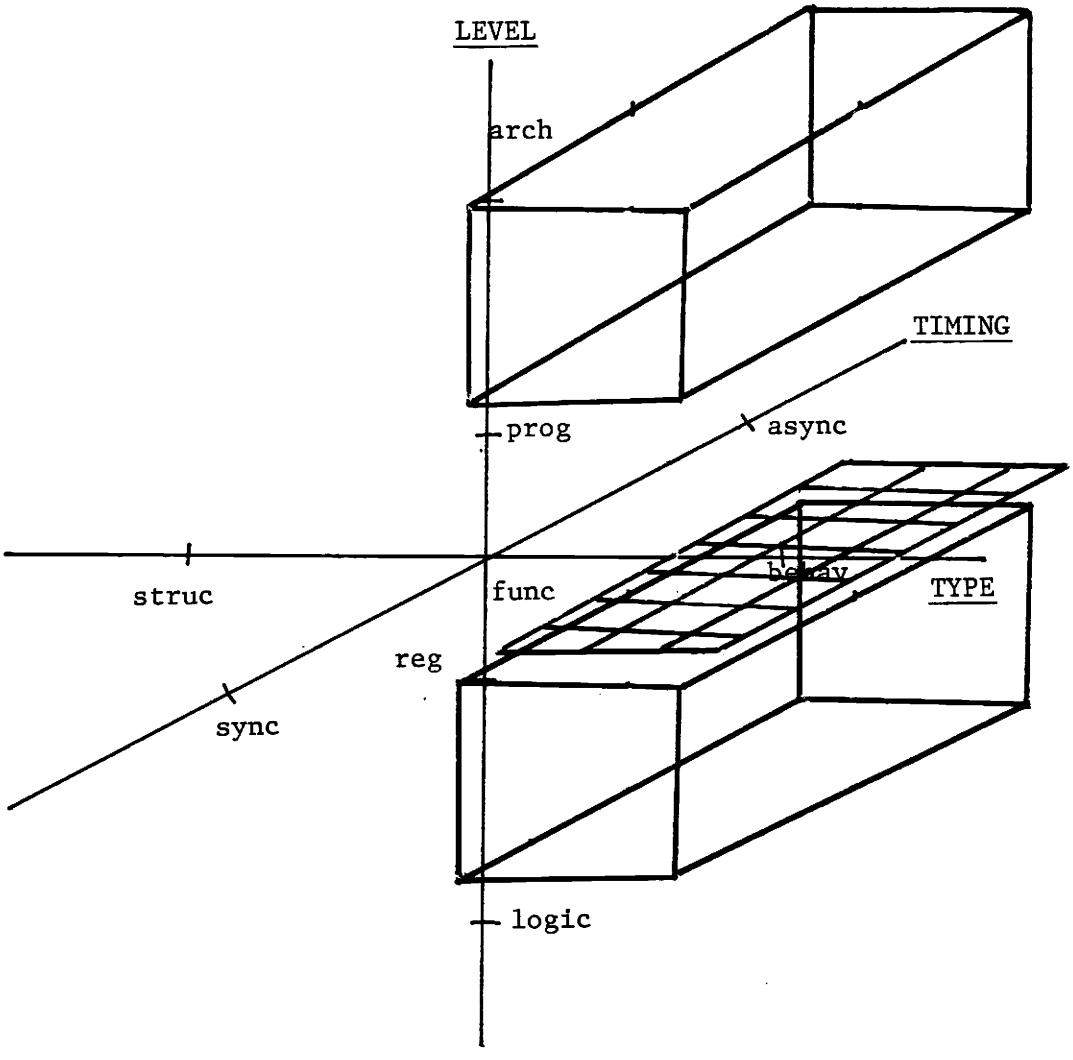
```
comptype      Dff;
inward        Clk,D:BoolNet;
outward       Q:Boolnet;
begin
while         TRUE do begin
        waitfor   clk   check clk;
        assign    D  to  Q;
        end;
end;
```

ADLIB D flip-flop comptype

```
comptype      Nor;
inward        A,B:BoolNet;
outward       Y:BoolNet;
begin
while         TRUE do begin
        assign NOT (OR(A,B)) to Y;
        waitfor TRUE check A,B;
        end;
end;
```

ADLIB NOR gate comptype

ADLIB Descriptions of Two Logic Components
Figure 10
(taken from example given in Cory, 1981)



ADLIB Representation in CHDL-Space
Figure 11

A powerful set of operators allow for the description of system behavior at several levels of detail. However, AHPL is primarily used for design work at the register transfer and logic levels. This language has been used extensively as a teaching aid for computer hardware design.

AHPL is based on the notational conventions of APL. Only the APL structures that can be interpreted as hardware constructs have been incorporated into AHPL. Additional constructs have been developed to represent hardware activities not covered by APL. These include parallel control sequences, facilities for asynchronous actions and conditional transfer sequences [Hi74]. The AHPL language uses the economy of the APL notation to provide a compact hardware description. The vector capabilities of APL are very useful in AHPL for the description of registers and memories (primitive data types).

A unique feature of AHPL that sets it apart from most other CHDLs is the partitioning of data and control statements. The data statements include registers and memories that store information, while the control statements include sequential logic to control the register (data) transfers. The control statement is either a conditional or unconditional branch to a data statement. Each data statement is followed by a control statement to specify the correct order of execution. The combination of a data followed by a control sequence is called a control sequence. Including more than one data statement in a control sequence allows for concurrent actions. The use of control expressions to insure the correct sequencing characterizes AHPL as a non-procedural language.

An AHPL description consists of a declaration section, a series of control sequences and a section of "always active" statements [NH79]. Allowable AHPL declarations include flip-flops, registers, memories, buses, inputs and outputs. The series of control sequences are sequentially numbered by statement line and follow the declaration section. There are no branches among the "always active" statements. These are active when the corresponding condition portion of the statement is evaluated as true.

AHPL descriptions use an implicit single phase clock which does not need to be declared. Control operations are triggered by the trailing edge of this clock signal. All data operations are triggered by the trailing edge of a positive control pulse. Each control sequence takes one clock cycle to execute. AHPL descriptions can also be written in an asynchronous format.

Subroutines are used to develop a description modularity similar to that of the hardware being described. Due to the single-block structure (as in APL) all variables (declared elements) are global.

In addition to description of hardware, AHPL was developed for compilation and simulation. A compiler and simulator have both been developed for this language [Ge71,SN77,NS78,Ro78,NH79]. HPSIM2 is a second generation simulator for AHPL that is currently in use at the University of Arizona [NH79]. This software package is implemented in FORTRAN and is currently running on CYBER 175 and DEC-10 systems. HPSIM2 simulates a designer's AHPL hardware description and if the results are correct invokes a hardware compiler to

generate a wiring list for the design.

A simple AHPL description of an 8's complement logic network is given in Figure 12. Note that no "always active" statements are needed in this description. A CHDL-space representation of this language is shown in Figure 13.

CDL

CDL (the Computer Design Language) [Ch65,Ch72b,Ch74a,Ch74b] was developed to describe the functional organization, algorithms and sequential operations of digital computers. First introduced in 1965, CDL is one of the oldest CHDLs still active in the hardware description field. It is also one of the most popular due to its simple structure and FORTRAN implementation [Sh79a]. This language has been used as a teaching tool in textbooks on computer organization [Ch70,Ch72a] and implemented for several simulation and design automation projects [Me68a,Me68b,HC77,Ch69,BB75].

CDL was developed with a structure similar to that of the ALGOL-60 programming language. Its structure is like that of a single ALGOL block with global variables, although it has no provisions for nested functions or subroutines. The language has a wide variety of computer elements which simplify the writing of a hardware description.

A hardware description includes the declaration of the computer's elements and a series of CDL statements that define the micro-operations of the design. Each of these statements is preceded by a Boolean expression that controls the execution of the statement, making CDL a non-procedural language. Multiple statements

INPUTS: x, start

OUTPUTS: Z

REGISTERS: Y(2)

1 $Y \leftarrow x, \emptyset$

2 (start X 1 + $\overline{\text{start}}$ X 5)

3 $Y \leftarrow x, Yo; Z=Y_1$

4 (start X 1 + start X 5)

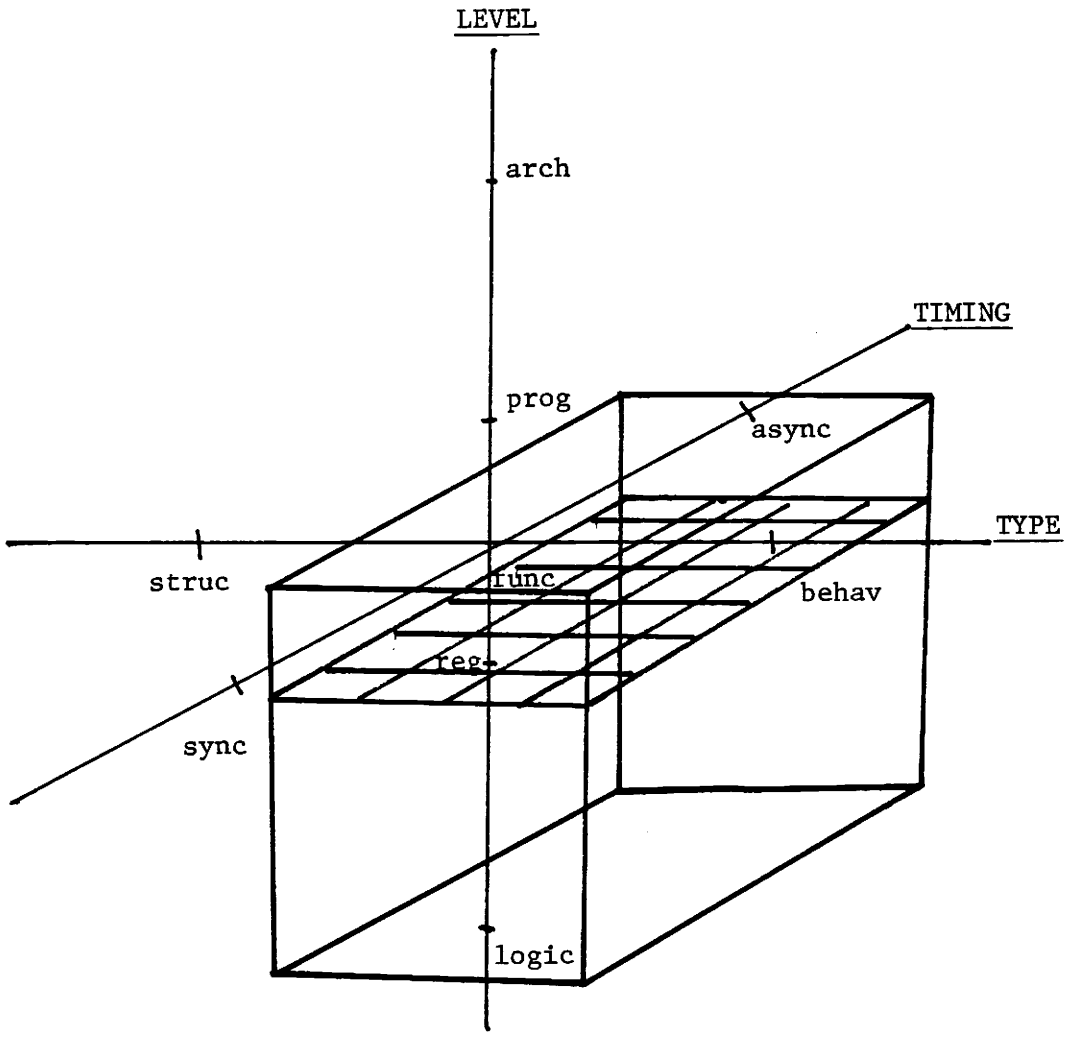
5 $Y \leftarrow x, Yo; Z=Y_1$

6 (start X 1 + $\overline{\text{start}}$ X 7)

7 $Z=COMP_2\ (x,Y)$,
$Y=a^2/COMP_2(x,Y)$

8 (start X 1 + $\overline{\text{start}}$ X 3)

AHPL Description of 8's Complement Logic Network
Figure 12
(taken from example given in Hill, 1974)



AHPL Representation in CHDL-Space
Figure 13

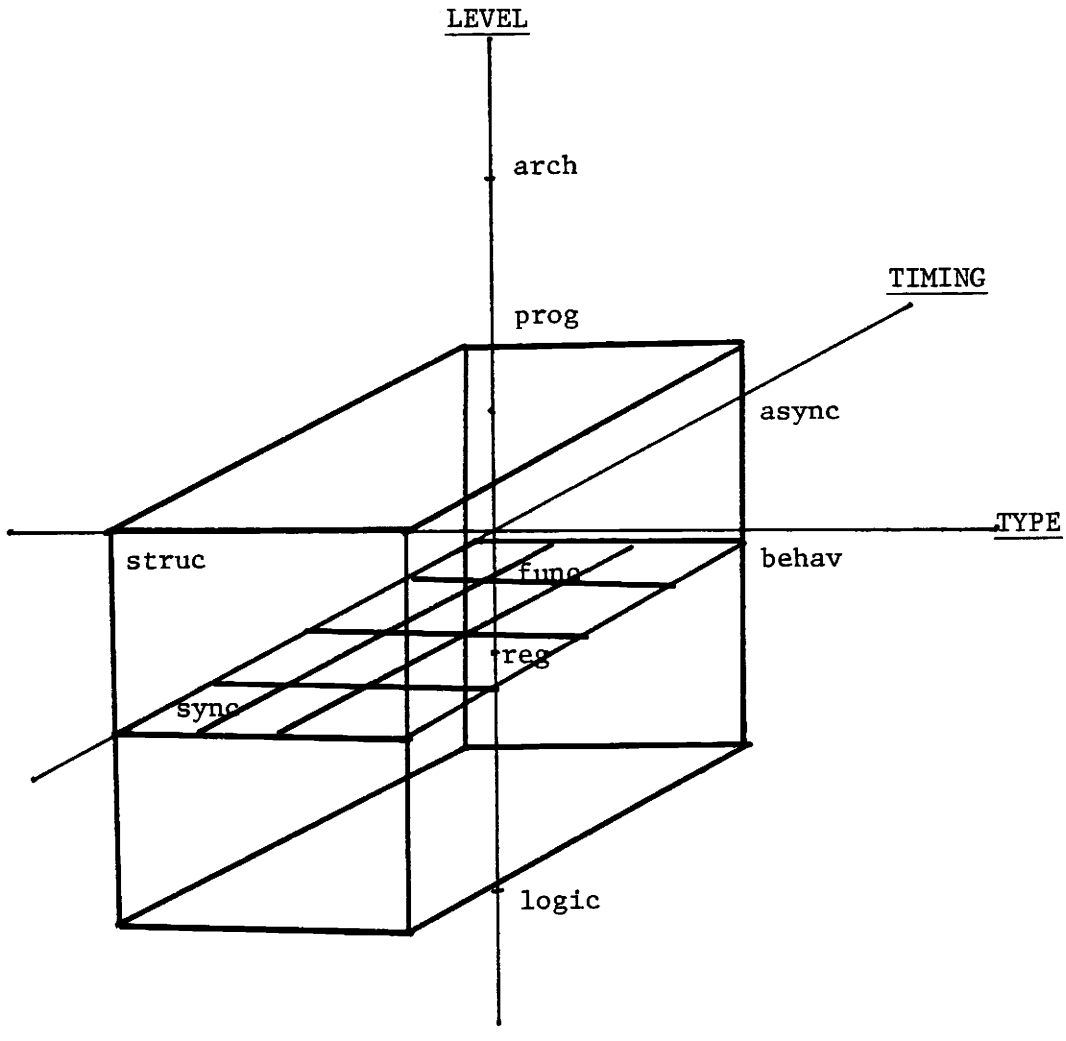
can be activated concurrently by dropping the control "label" of the secondary statements and using correct CDL punctuation. A multiple-phase clock can be explicitly declared in CDL descriptions and used for control expressions. CDL is not well suited for describing hardware in a modular fashion as it lacks function and subroutine facilities [Sh79a].

A translator and simulator have been developed for the CDL language. The translator is used to perform a syntax check on the system description and translate it into Polish string logic equations [Sh79a]. The simulator executes the output of the translator via a set of simulation commands. During the execution of the simulator, data is collected and can be displayed at every clock cycle [Ba75]. Some very complex digital systems have been simulated using various versions of the CDL simulator; for example, the simulation of 20 instructions from a PDP-8 took over two hours of CPU time on an IBM 370/155 [HC77]. Both the translator and simulator have been implemented in FORTRAN.

An example of CDL description is given in the more complete CDL review in Chapter 2 of this paper. A CHDL-space representation of CDL is given in Figure 14.

DDL

DDL (the Digital Systems Design Language) [Du67,DD68,Di71,Di74] was developed to describe and simulate combinational and sequential logic networks. The language allows for hardware specification at several levels of detail, including the architectural, register transfer and logic levels. A textbook has been published that uses



CDL Representation in CHDL-SPACE
Figure 14

DDL as a teaching aid, and DDL was chosen for use in NASA's Computer-Aided Design and Test System (CADET) [Sh79a,Sh79b].

DDL is based on the idea that hardware systems can be split into semi-independent subsystems. These subsystems are represented in a hierarchial block structure in DDL descriptions. The control portion of each subsystem is viewed as a finite state machine. Sequencing in DDL descriptions is characterized by state variables and state transitions. Because the state transitions must be written explicitly in this notation, DDL is a non-procedural language. Parallel and asynchronous operations can also be specified in DDL descriptions.

A DDL description provides the explicit declaration of the structural elements used in a design, including registers, memories and terminals. The behavioral specifications of the system consists of transforming the declared elements with operators from a large set. The control section uses state variables to insure correct sequencing of the behavioral statements.

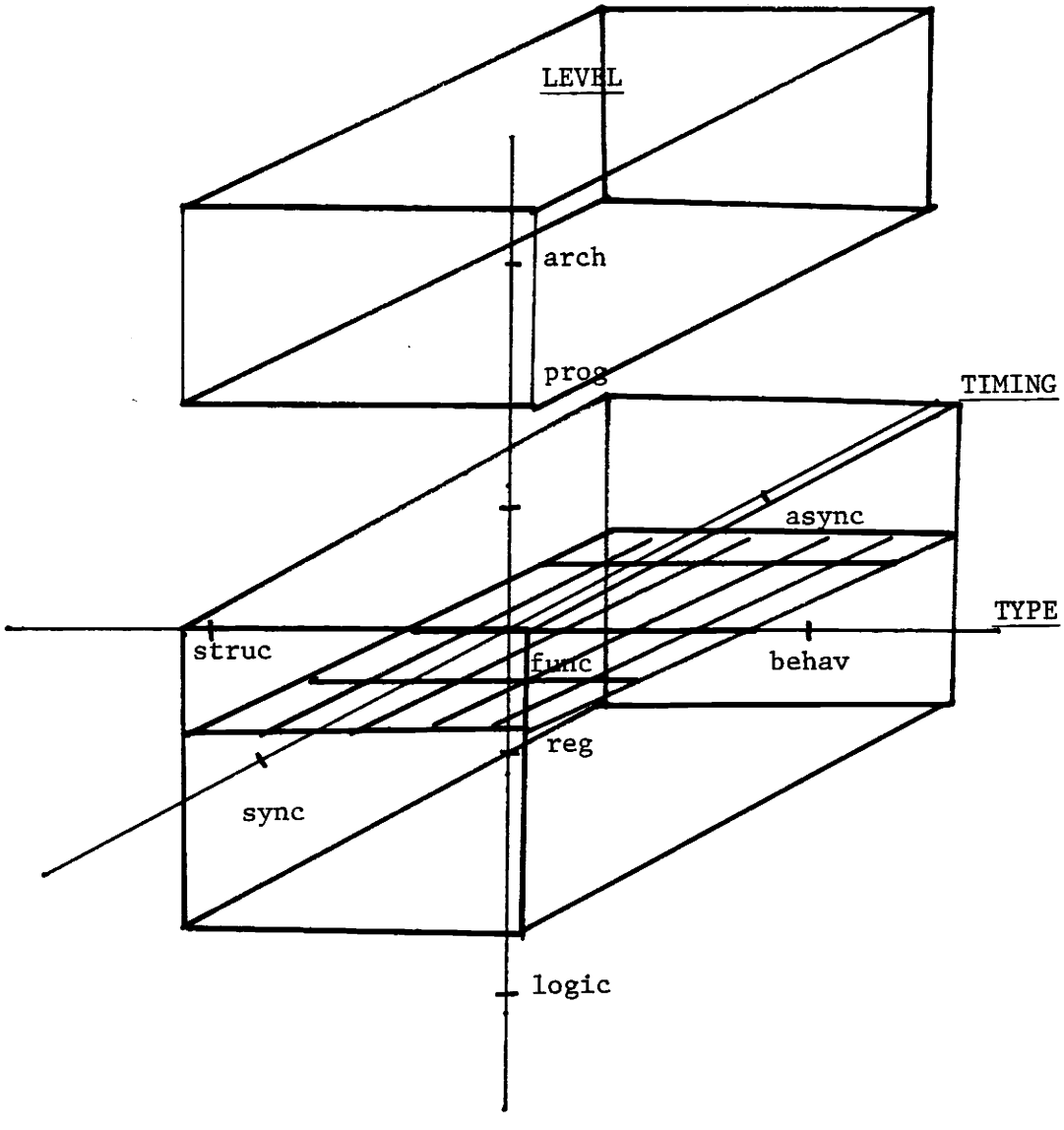
A translator and simulator for this language have been implemented in FORTRAN [DD69] and PASCAL [CD79a,CD79b]. The translator (DDLTRN) [Di71a] translates a DDL description into a set of Boolean equations and register transfer statements [Sh79b]. The simulator (DDLSIM) [Di71b] uses the output of the translator to simulate the design. This simulator possesses a simple yet powerful "free-format" command language that allows a designer to have total control over the simulation process [Sh79b].

An example DDL description is given in the more complete DDL review in Chapter 2 of this paper. A CHDL-space representation of DDL is given in Figure 15.

FLOWWARE

FLOWWARE [Om73,CT77] was developed to describe digital systems in a graphical manner and to allow simulation from that representation. FLOWWARE is used to develop descriptions at the register transfer level of the design hierarchy. A register layout diagram is used to represent the hardware structure and organization accompanied by a flowchart to specify the correct control sequence to be followed. FLOWWARE was first introduced in 1973 and in 1976 it was expanded to its current structure [Ch76a,Ch76b,Ch76c].

The FLOWWARE description process is split into two phases. The first phase is an information flow phase that includes declarations of registers, memories, clocks, etc. and drawing lines to connect these declared elements. Declarations are made by instructing the host computer to place a specific symbol on a layout diagram. The computer will then draw the appropriate figure on the CRT. Lines are also specified by the user and drawn by the computer. A description at this phase represents the structure of a digital system. The second phase is a control flow phase that specifies a control algorithm with a flowchart. Common flowchart elements include the function, decision, decode, start and terminate blocks. The register transfer activities of the system are written inside these blocks. A description at this phase represents the behavior of a digital system.



DDL Representation in CHDL-Space
Figure 15

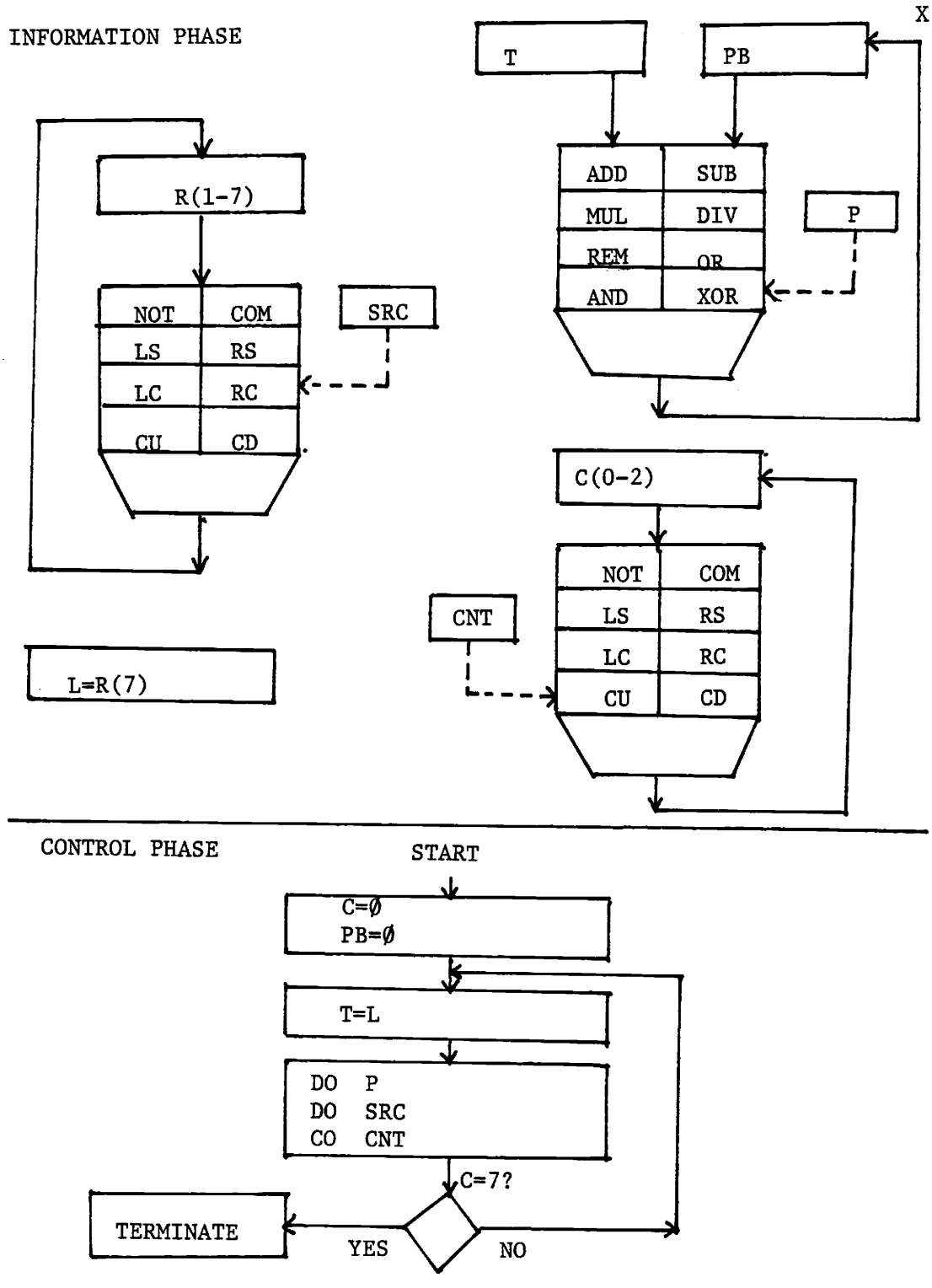
The FLOWWARE translation and simulation processes require the use of a second software package called IDDAP (Interactive Digital Design Assistance Package) [CT70]. IDDAP uses a subset of CDL and acts as an interactive translator and simulator for a FLOWWARE description. Input information to IDDAP and output information from IDDAP are in a textual form so FLOWWARE must both pre- and post-process a description to be simulated. The simulation occurs in a textual form while the input and output are in a graphical form. Simulations can be clocked or unclocked although all clock changes must be explicitly declared in the FLOWWARE description [CT77].

The FLOWWARE/IDDAP system is implemented at the University of Missouri-Rolla. The computer used is an IBM 360/50 linked to several Data General NOVA 800 minicomputers. A Tektronix T4014 graphics terminal is used for the user interface. As implemented, FLOWWARE consists of two programs running simultaneously on the host IBM and remote NOVA computers. The IBM 360/50 program is written in PL/1 while the NOVA is programmed in assembly languages [CT77].

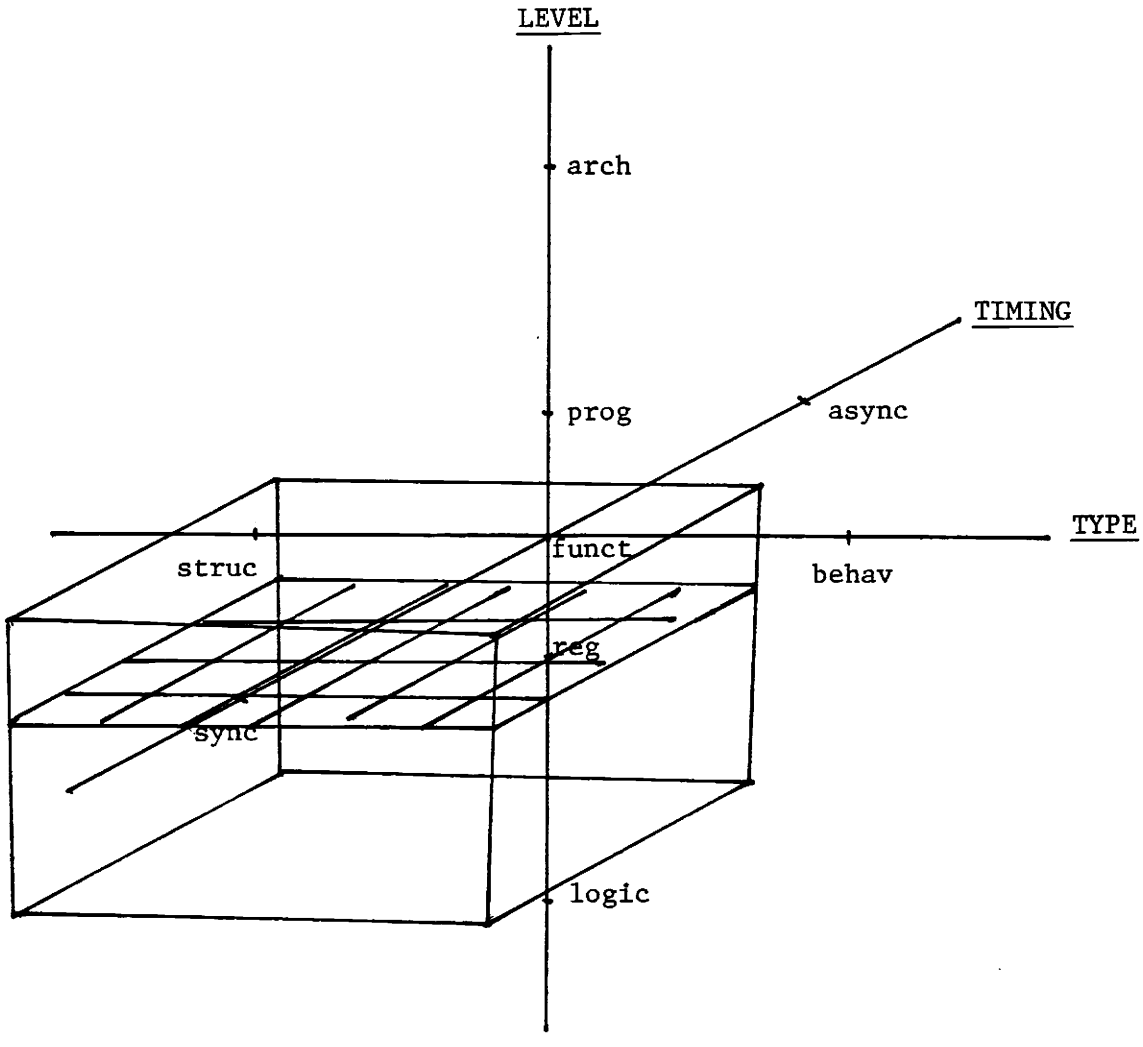
A FLOWWARE description of a serial parity bit generator is given as an example in Figure 16. A CHDL-space representation of this language is shown in Figure 17.

ISPS

ISPS (Instruction Set Processor Specifications) [Ba81a,Ba79, BB78] is a very popular CHDL that was developed to describe the behavior of digital computers at the programming (instruction) level.



FLOWWARE Description of a Serial Parity Bit Generator
Figure 16
(taken from example presented in Ching, 1977)



FLOWWARE Representation in CHDL-Space
Figure 17

The programming level is one level higher in hardware abstraction than the register transfer level as it deals with instructions and the rules for interpreting those instructions. The basic elements of this level are data operations, instructions and instruction interpreters [Ba75]. ISPS can be used to describe systems at the register transfer level (see Figure 18) but it was primarily designed to describe digital systems at the programming level.

> "It is important to emphasize that ISPS supports a wide range of applications, rather than a wide range of design levels" [Ba81a].

ISPS is the second attempt to implement the ISP notation as a computer language. The ISP notation first appeared in 1970 [BN70, BB72,Si74] and was the result of trying to develop a uniform notational scheme for a book on computer structures [BN71,SB82]. The idea behind the development of the ISP notation was that the logical behavior of a processor could be specified by knowing the type and sequence of its operations. This sequence is determined by the bit patterns stored in a system's memory and by a set of rules for interpreting those bit patterns. Thus if the type of operations and the rules for interpretation are specified then the behavior of a processor depends only on the program stored in memory. ISP is a notation to provide these specifications. The first attempt to incorporate the ISP notation into a hardware description language resulted in ISPL [Ba76]. Five years later, in 1978, the ISPS language was introduced [BB78].

A fixed notational format is usually used for ISPS descriptions. This format consists of a declaration section followed by an

```
mult (a<7:0>, b<7:0>) <15:0>:=
    BEGIN
    mult -0  NEXT
    temp <15:0>  "00 @ b NEXT
    COUNT -8 NEXT
    run:=
       BEGIN
       IF a <0> =>
           mult-mult + temp NEXT
       a-a SRO 1 NEXT
       temp-temp SLO 1 NEXT
       COUNT-count -1 NEXT
       IF (count GTR 0) =
           RESTART run
       END
END
```

REGISTER TRANSFER LEVEL VIEW

```
mult (a <7:0>, b <7:0>) <15:0>:=
     BEGIN
     mult a*b
     END
```

PROGRAMMING LEVEL VIEW

ISPS Descriptions of 8-bit Multiplier
Figure 18
(taken from example used in Northcutt, 1980)

activity section. The declaration section is used to define memory, registers, allowed data operations and the format of the machine's instructions. The activity section is used to define both the instruction interpreter and the processors' instruction set. The interpreter is the mechanism that fetches, decodes and executes an instruction. The instruction set include all of the instructions a particular processor can execute. An instruction is described by a condition and associated action sequences.

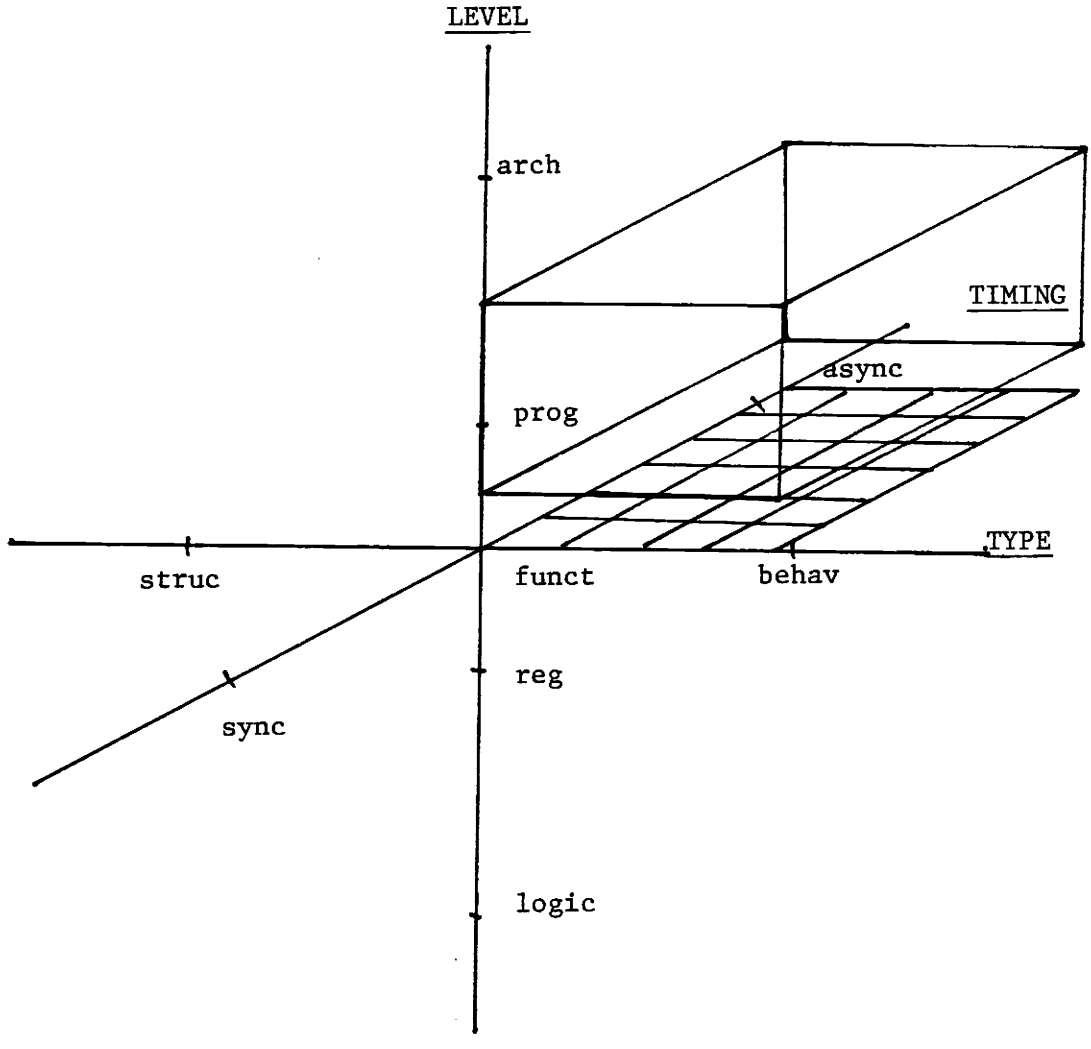
ISPS descriptions use a block structure to provide for hardware modularity. Function and MACRO statements are used to develop hierarchically detailed descriptions. An ISPS description is asynchronous and since no specific control signals are used for sequencing it is classified as a procedural language. Typically, ISPS is not used for precise timing studies, due to the lack of an explicit clock declaration. Timing information can be produced if it is carefully built into the description [Ba81b].

ISPS is currently the basis for a multi-level design automation system being used at Carnegie-Mellon University [Ba80,BS75]. Besides the simulation and synthesis of hardware, software generation [Bs75], programming verification and architecture evaluation [BS77] have all been performed with the ISPS language [Ba81].

As an example of an ISPS description, the 1948 Manchester Mark 1 computer has been described in ISPS notation in Figure 19. Other examples of ISPS descriptions can be found in the following two sources [SN82,BM78]. A CHDL-space representation of ISPS is given in Figure 20.

```
MARK 1:=
   begin
** MP. State**
   M[0:8191]<0:31>
**PC.State**
  P1\Present.Instruction<0:15>,
      f\function <0:2>:= P1<0:2>,
      s\address<0:12> :=P1<3:15>,
  CR\Control.Register <0:12>,
  ACC\Accumulator <0:31>,
**Instruction Execution**
  icycle\instruction.cycle {main} :=
     begin
     REPEAT
        begin
        PI←— M CR <0:15> next
        DECODE f = >
           begin
           #0:=CR=M[S],                 !JMP S (JUMP TO S)
           #1:=CR=CR+M[S]               !JRP S (RELATIVE INDIRECT JUMP)
           #2:=ACC=-M[S]                !LDN S (LOAD NEGATIVE S)
           #3:=M[S]-ACC                 !STO S (STORE S)
        $4:#5:=ACC=ACC-M[S]             !SUB S (SUBTRACT S)
           #6:=IF ACC lss 0 =>CR=CR+1 !CMP   (COMPARE AGAINST ZERO)
           #7:=STOP(  )                 !STOP
           end next
        CR=CR=1
        end
     end
  end
```

ISPS Description of Manchester Mark 1 Computer
Figure 19
(taken from example presented in Sieworek, 1982)



ISPS Representation in CHDL-Space
Figure 20

KARL

KARL (the Karlsruhe Architectural and Register Transfer Language) [Ha77,Hv79] is intended for design documentation and use as a modern design approach. KARL combines a graphical notation and a textual register transfer language to create a description approach suitable for use at the register transfer and instruction levels. The language is best known in Europe where it has been used as a pedagogical tool in a textbook on hardware design [Ha77].

KARL was developed to provide a descriptive aid for use in cases where both a textual CHDL description and a graphical block diagram representation were needed to supply information for a design. A modified version of CDL, called CDL/KA (Karlsruhe), is used as a textual documentation language. A graphical language, ABL (A Block Diagram Language), is used to obtain equivalent block diagrams of the CDL/KA description. The combination of both language descriptions into one common format is called KARL.

The CDL/KA language used in a KARL description adds several extensions to the original CDL language. Primitive storage elements such as registers and memories remain the same but memoryless elements have been extended with the addition of an ENCODER statement. The peripheral elements SWITCH and LIGHT are now accompanied by UNIT and EXTERNAL device types. Control statements have also been extended with the introduction of the following three statement types: [Ha77]

AT...UNTIL...KEEP...ELSE...

WHILE...KEEP...ELSE...

AT...DO...

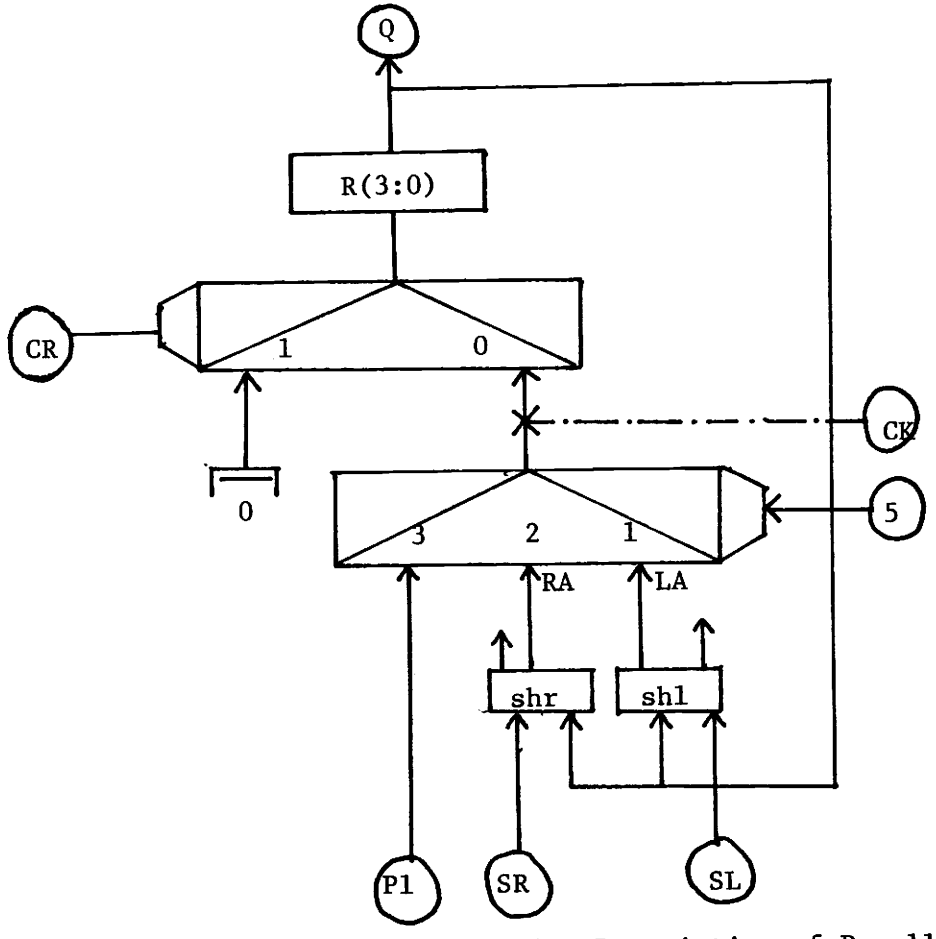
The first two statements are used with asynchronous activities while the third is used for synchronous operations.

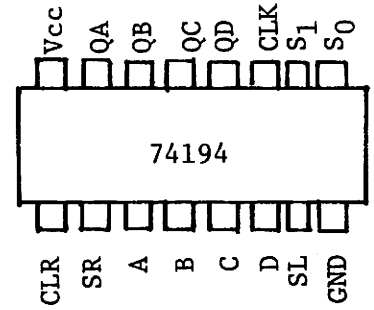
With the use of ABL descriptions, a complete KARL description of a digital design can become quite large. For this reason a shorter KARL description of a 74194 TTL logic chip will be used as an example in Figure 21. A CHDL-space representation of this language is given in Figure 22.

PMS

PMS (the Processor Memory Switch) [BN70,BK72,SB82] notation was developed to describe the structure of a digital system at the highest level of design. This level, the architectural or PMS level, describes systems in terms of processing units, memory components, peripheral units and switching networks. The PMS notation is an excellent tool for such things as system performance analysis and bottleneck detection [SB82]. This notation was originally used in a textbook to describe the physical structure of computers and computer networks [BN71]. The PMS notation has been implemented as a CHDL called PMSL [Kn73].

PMS is primarily a graphical notation (PMSL is in a textual form), that presents system structure information in a block diagram format. There are seven basic elements used for descriptions at this level, these are: Memories, Processors, Links, Controllers, Switches, Transducers, and Data Operations. Components from the

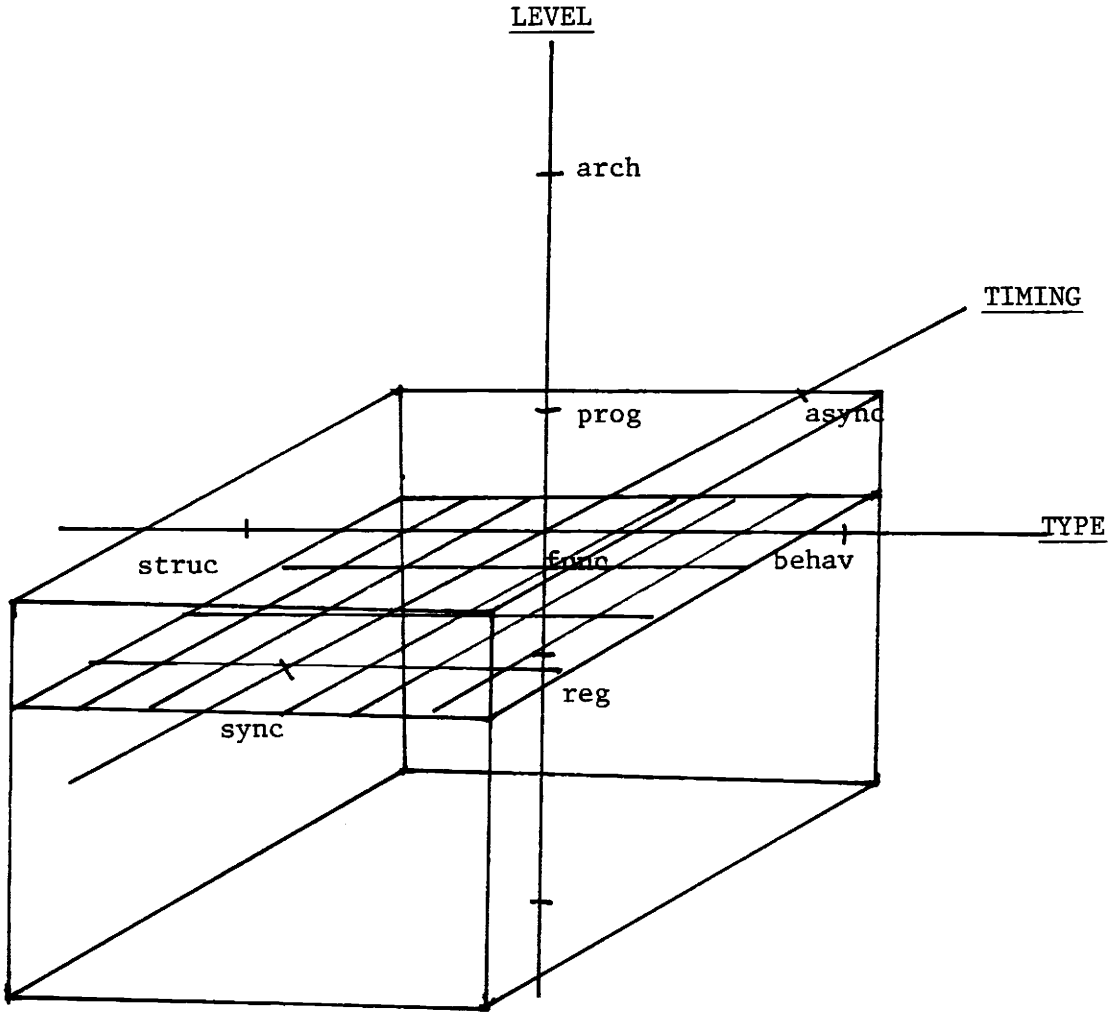




Symbolic CDL/KA Description

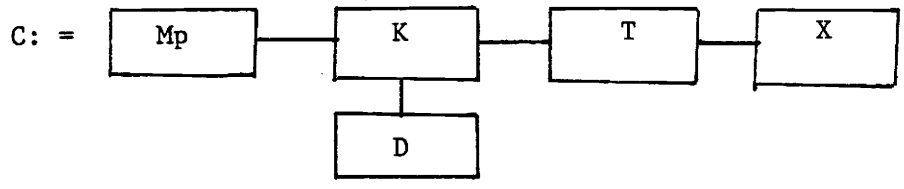
```
Unit TTL 194 (P1(3:6),CR,trigger(CK),SR,SL,
              S(1:0);Q(3:0));

register Q(3:0);

while CR keep Q=0 else
      at CK do case S of
      (0:  )
      (1: Q:(  ):=shl Q:SL)
      (2:(  ):Q:=shr SR:Q)
      (3:Q:=P1)

END TTL 194
```

KARL Description of Parallel Shift Register
Figure 21
(taken from example presented in Hartenstein, 1977)



KARL Representation in CHDL-Space
Figure 22

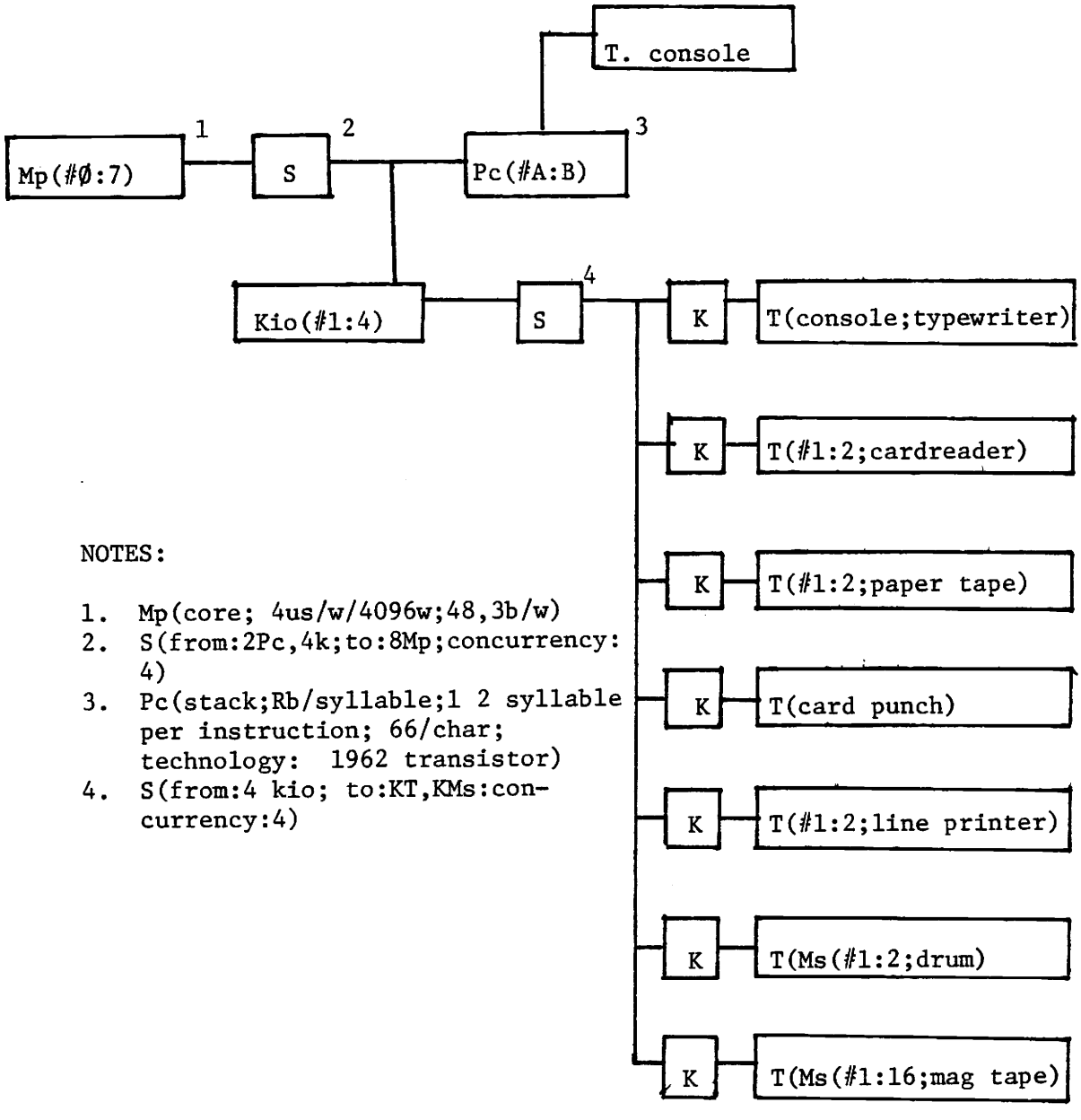
seven categories are connected to make stored program computers, which become an eighth element in the PMS descriptions. The classic structure of a simple computer consisting of a memory, a processor (with control and arithmetic portions), a transducer and an external device is given in PMS notation below [BN70].



In this example Mp is the primary memory, K is the processor control unit, D is the ALU, T is the transducer and X is some external device. The structure of a system is implied by the inclusion of an element in a PMS description and by the interconnection of these elements. The functions of the elements are either implied by the element label or can be explicitly listed by means of an attribute list [BK72]. PMS provides for little functional information in its notation. Hierarchical descriptions can be obtained by decomposing large blocks into subcomponents with more specific functions. This can be seen in the PMS example of a Burroughs B5000 computer system in Figure 23. In this example the transducer block of the generalized computer structure (shown above) has been expanded. A CHDL-space representation of PMS is given in Figure 24.

SDL

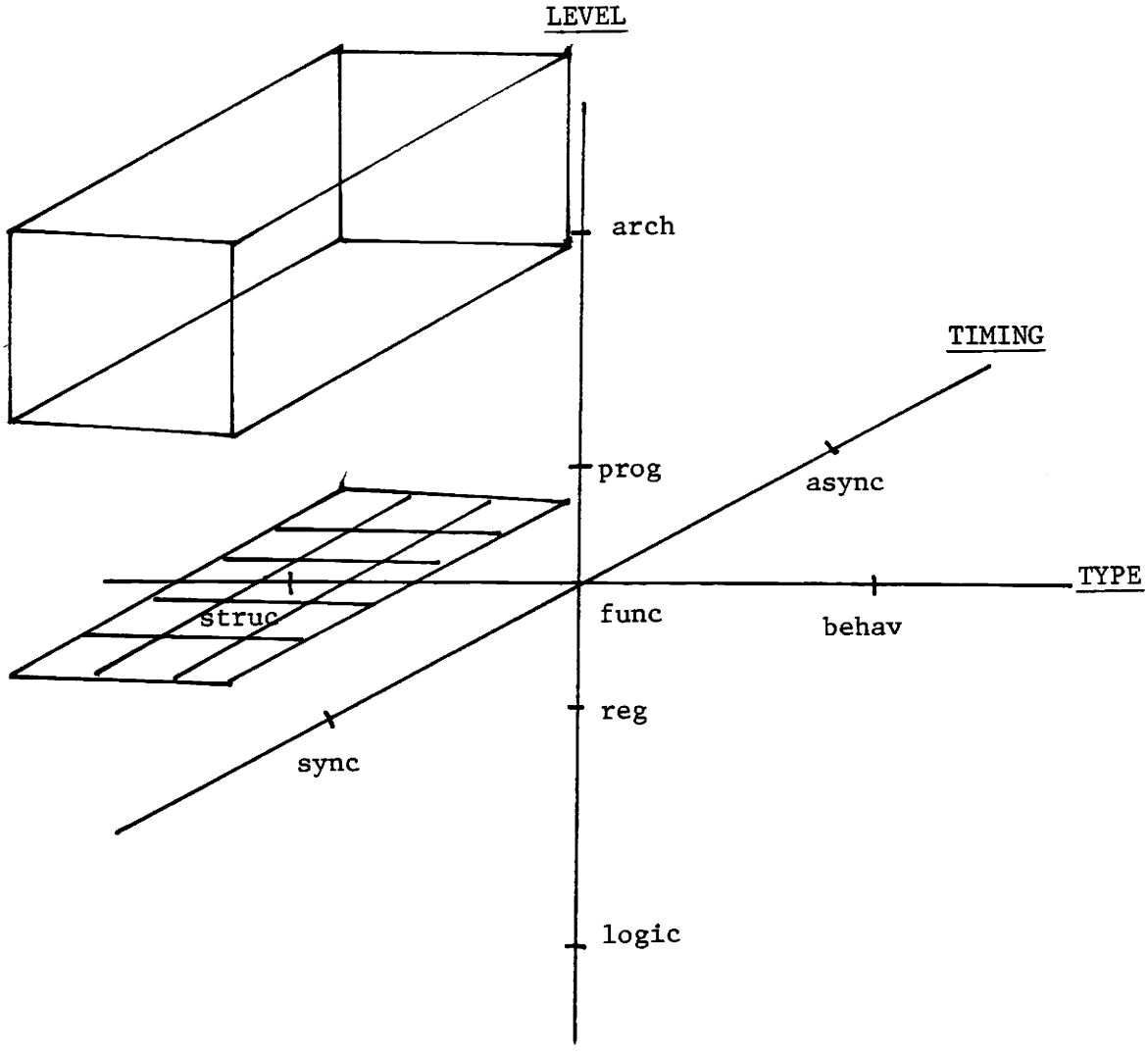
SDL (the Structural Design Language) [Va77,Co81,va77b] was developed for describing only the structural properties of a digital system. The language is intended as a complement to existing CHDLs



NOTES:

1. Mp(core; 4us/w/4096w;48,3b/w)
2. S(from:2Pc,4k;to:8Mp;concurrency: 4)
3. Pc(stack;Rb/syllable;1 2 syllable per instruction; 66/char; technology: 1962 transistor)
4. S(from:4 kio; to:KT,KMs:concurrency:4)

PMS Description of B5000 Computer
Figure 23
(taken from example presented in Siewiorek, 1982)



PMS Representation in CHDL-Space
Figure 24

that emphasize only the behavioral aspects of a computer description [va77b]. SDL can be used to represent structure at all levels of the design hierarchy. This language has been combined with ADLIB [Hi79] to form the SABLE multi-level digital system simulator [Hi79].

SDL is unique in that it provides only the structural information of a design. There are four objectives the language was developed to meet; these are: [va77]

1. To provide accurate representation of structural information.
2. To be useful over all levels of the design process.
3. To be applicable to the different purposes a designer may have during the design process.
4. To be able to perform designer-controlled mapping of higher-level hardware primitives into lower-level ones.

To achieve these objectives a language describing how systems are interconnected was developed.

SDL descriptions consist of an individual block connected to its outside world by (EXTERNAL) connections. These external connections can also be divided into OUTPUTS and INPUTS if that information is necessary. Some external connections may be logically equivalent, this can be taken into account with an EQUIVALENCE statement. When using SDL it is necessary to declare the types of all components used in a description. For each TYPE of component names must be given to all components of that "type." Finally, all component interconnections must be defined. In this language an interconnection is called a "net." Several other statements are

allowed with SDL including: MACRO, PURPOSE, and LEVEL.

The future use of this language seems to lie in the area of a design automation system, such as SABLE, and in design verification studies.

An SDL description of a 16-bit shift register built from two 8-bit shift registers is given in Figure 25. A CHDL-space representation of this language is given in Figure 26.

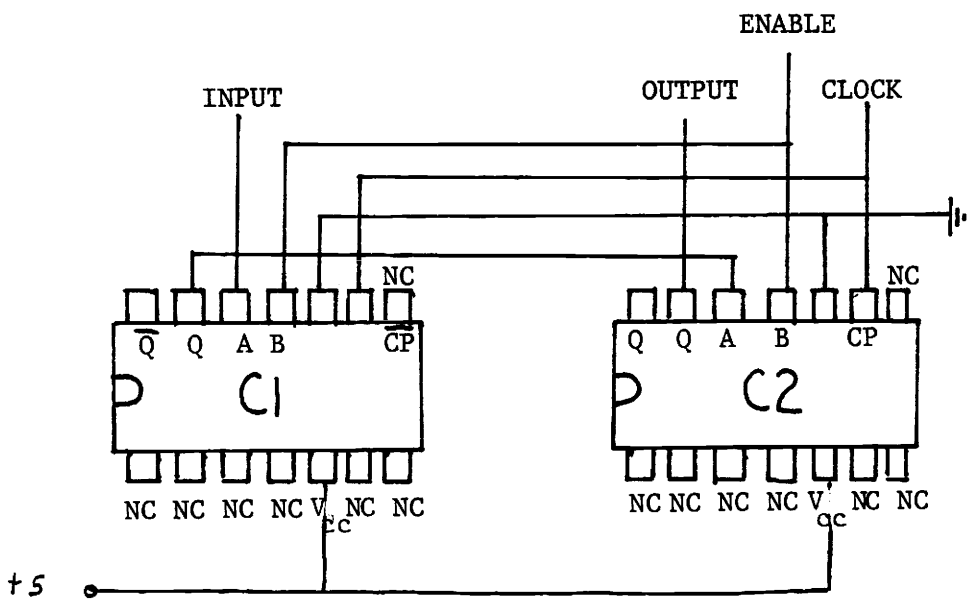
SLIDE

SLIDE (Structured Language for Interface Description and Evaluation) [PW81,PA80] was developed to describe the behavior of input/output interfaces and interconnected digital systems. This language is used to describe asynchronous activities at the register transfer level. SLIDE is an updated version of an early I/O description language called GLIDE [PW77].

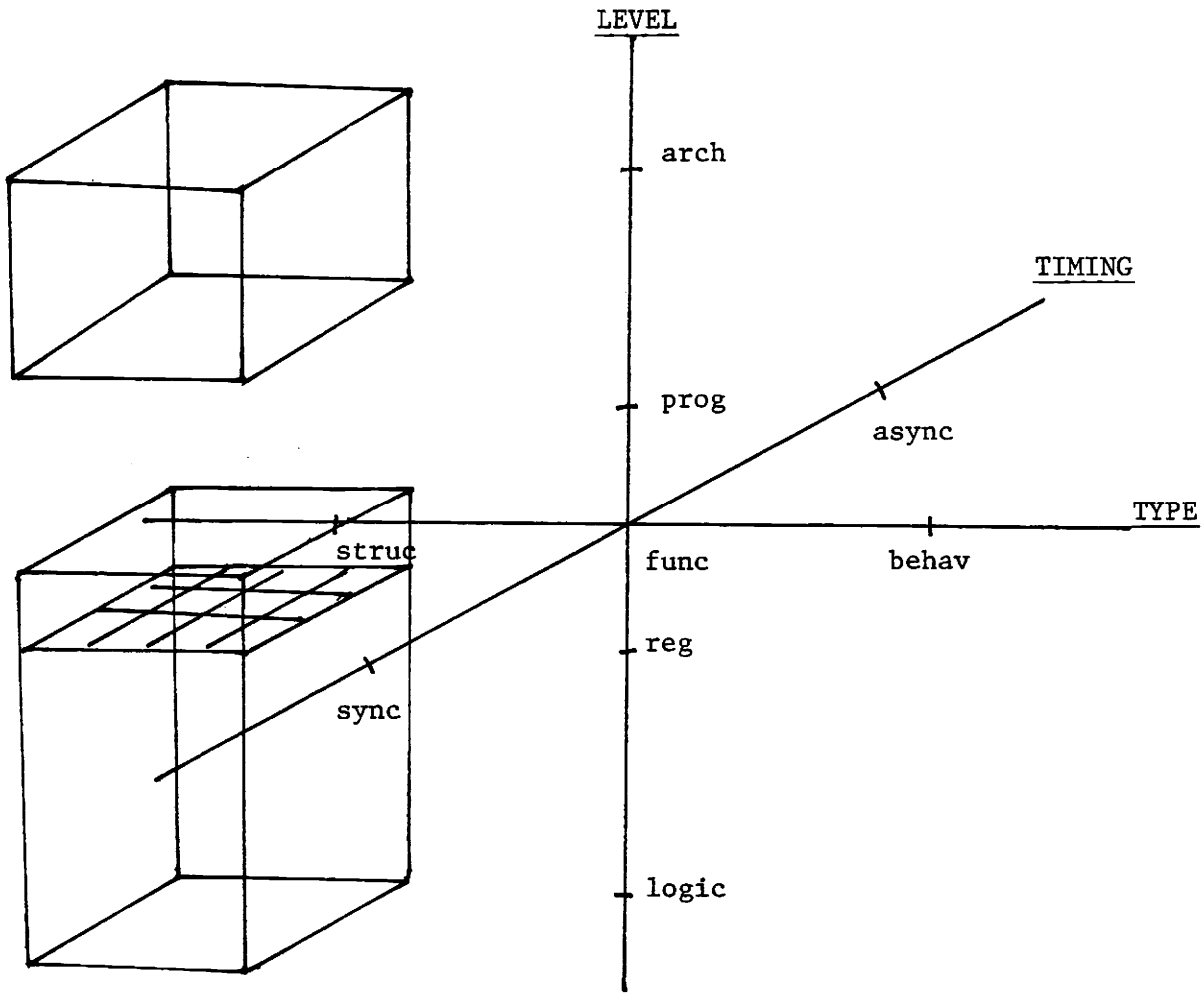
The purposes behind developing an I/O description language are unique to this class of CHDL. First, a nonprocedural environment is needed so that processes will be inhibited until some conditions become true. Secondly, a timeout construct is needed to prevent "fatal" timing problems in simulation runs. Finally, a set of I/O related primitives is needed. These are such things as synchronous line declarations, transition of logic levels (rather than at edges), and FIFO declarations.

A SLIDE description is constructed by developing a number of processes and placing them into one main process block. A process is a hardware unit that is executing independently from any other hardware unit. Within each process local variables (registers and

```
NAME:     SHIFTREG:

PURPOSE:  LOGSIM, PCBGEN, CKTANALYSIS:

LEVEL:  TTLPACK:

TYPES:  SN7491;

EXT::DATA, ENABLE, CLOCK, Q;

SN7491:C1, C2;

NET 1=FROM (.CLOCK) TO (CL. CLOCK, C2 CLOCK);

NET 2=FROM (.ENABLE) TO (CL. ENABLE, C2.ENABLE);

NET 3=FROM (.DATA) TO (CL.DATA);

NET 4=FROM (C1.Q) TO (C2.DATA);

NET 5=FROM (C2.Q TO (Q));

END;
```



SDL Description of 16-bit Shift Register
Figure 25
(taken from example presented in vanCleemput, 1981)



SDL Representation in CHDL-Space
Figure 26

lines) and other processes (subprocesses) can be declared. These processes communicate through variables declared in the main process (which are global variables) and executed synchronously. Process execution is dependent on its assigned priority in the description. A set of three rules must be satisfied to schedule execution of a process. These are:

1. If the process is a subprocess of one currently existing.
2. If all initialization conditions are true.
3. If no process with a higher priority at the same subprocess layer is executing.

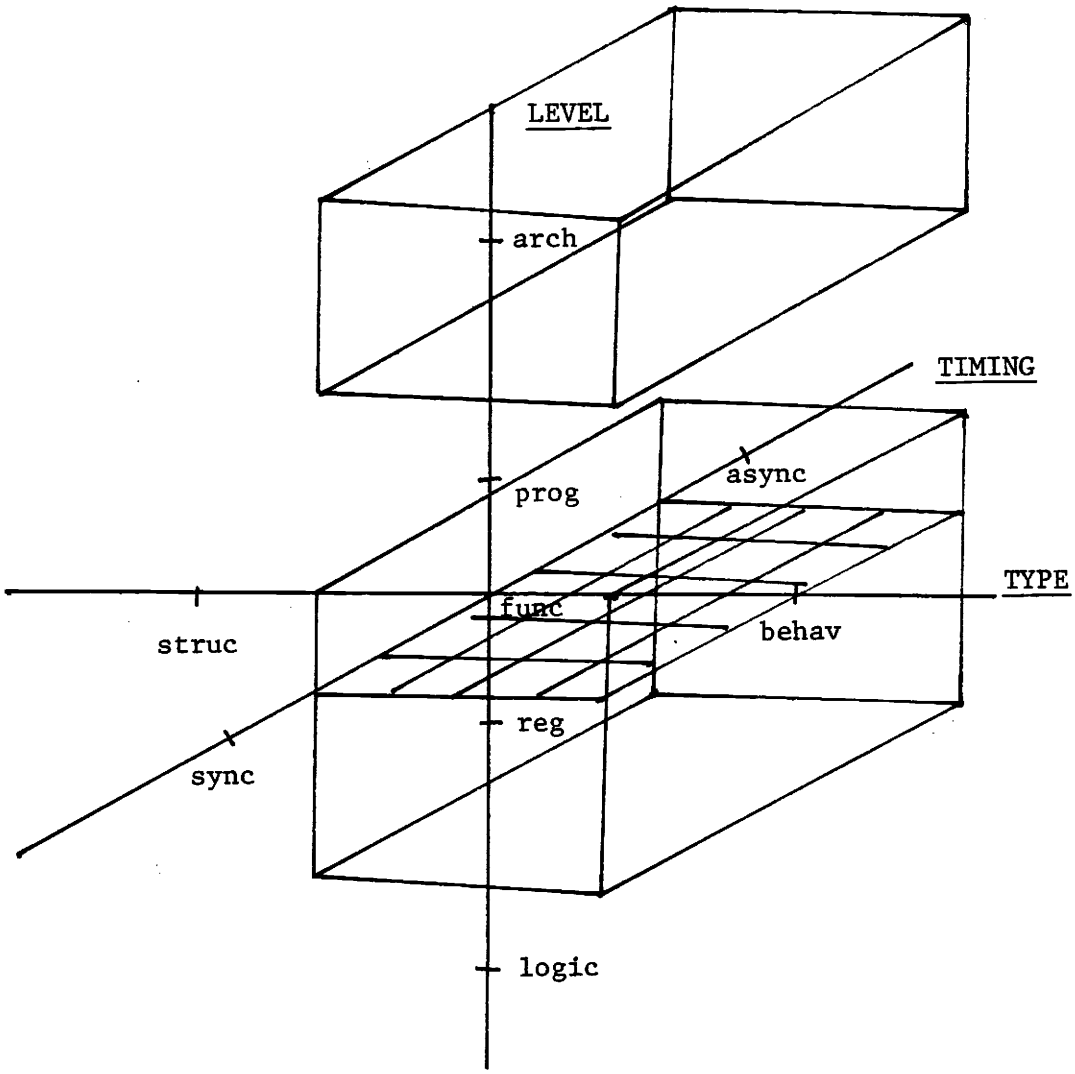
Pascal-like BEGIN...END blocks are used to allow for concurrent execution within a process.

A SLIDE simulator exists that allows the simulation of a SLIDE description. A hardware description is first written in SLIDE, then compiled into SIMULA-67 code, which is finally run on an interactive simulator. Both UNIBUS and D-bus have been simulated with this language [PA80]. A partial outline of a SLIDE description of the UNIBUS is given in Figure 27 [PW81]. A CHDL-space representation of SLIDE is given in Figure 28.

The following descriptions of some of the less publicized CHDLs are intended for the benefit of the reader who is interested in further studying this case of HDLs. References will be supplied in order to make this paper a good starting point for further research. APDL (A Processor Description Language) [Da68,Da69] is an ALGOL based language for the description of processors at the register transfer level. This language uses the block structure provided by

```
MAIN PROCESS UNIBUS;
   global bus declarations
   INIT I arbiter:0  WHEN initline EQL/;
   INIT arbiter:1  WHEN TRUE:
   INIT D reset:0 WHEN initline EQL/;
   INIT D transfer:1 WHEN dataready EQL/;
   INIT A reset:0    WHEN initline EQL/;
   INIT A transfer:1  WHEN dataready EQL/;
       .
       .
       .
     Other device processes and initiated similarly
   PROCESS arbiter:
       EXT REGISTER
           psw <15:0>;              !program status word!
       COMB pri <2:0>:=psw <7:5>:  !processor priority!
   BEGIN
       WHILE TRUE DO
       BEGIN
           IF npr THEN
               .
               .
               .
           ELSE IF (      ) THEN
               .
               .
               .
       NEXT
       .
       .
       .
       END    ! end of while true statement!
              ! end of arbiter!
       .
       .
       .
```

Partial SLIDE Description of UNIBUS
Figure 27
(taken from example presented in Parker, 1981)



SLIDE Representation in CHDL-Space
Figure 28

the ALGOL constructs to organize hardware descriptions into hierarchical levels. ALGOL extensions are used to handle timing and register activities. APDL descriptions must be translated into ALGOL before simulation can occur.

CASD (Computer-Aided System Design) [Cr70] is a collection of computer programs to aid in the design of computers. The CASD design language is based on an extended version of PL/1. Control and data operations are specified with the use of microinstruction sequences.

CASL (Computer Architecture Specification Language) [MO79] was designed to provide hardware descriptions at the register transfer level. CASL is best suited for describing machines that can be easily split into asynchronous modules [ST81]. CASL has many features that distinguish it from other CHDLs. These include abstract data representations and structure-oriented control mechanisms (FSMs).

DTMS (Descriptive Techniques for Modules and Systems) [TS80,Si81] was developed to provide mixed-level descriptions of systems or subsystems in a procedural, nonprocedural or mixed timing format. DTMS is also capable of describing the interconnections of hardware modules. It also provides constructs to describe parallel and concurrent operations called processes.

FST (Functional Simulator and Translator) [Fr67,B162,B171,FM68] is a set of computer programs that perform functional design and simulation of small scale synchronous digital systems. The FST system consists of four major facilities: 1) A language to describe the process flowchart of a digital system. 2) A compiler to convert

the flowchart descriptions into a list structured description of the system. 3) A simulator to verify that the system will perform as expected. 4) A translator to generate State and Control Unit Tables. This design process is best suited for synchronous systems.

FTL (Functional and Timing Specification Language) [Sc79] was developed for precise modeling of complex digital systems. This language is particularly well suited for timing analysis. An FTL description is intended to be used as an input for simulation of a design.

HILOMK2 [FM81] is an attempt to provide both structural and behavioral information about a digital system. The HILOMK2 language produces very accurate structural descriptions. Both synchronous and asynchronous operations can be described. A simulator written in BCPL and driven by a waveform description is available.

LCD [EG76,EG77] (a Language for Computer Design) permits the description of both the structural and functional aspects of a digital system. LCD descriptions can be made at very high architectural levels, at the logic gate level or at any level in between. This language requires the development of two alternate descriptions of the same design. The first description is a high level model giving the system specifications of the design. The second description is an implementation model that describes the dataflow or control operations at a particular design level.

LOGAL [St77,Lu73] (Logical Algorithmic Language) was developed to perform logic design at the register transfer level. This language adds several new features to Reed's RTL language. LADS (Logic Algorithm Design System) is used with LOGAL and provides software

support and processes a LOGAL description to allow simulations of digital systems.

MODLAN [PJ81,Pa76] (A Hardware Module Description Language) is used to describe digital systems at the gate, register transfer and instruction levels. This language uses PASCAL-like constructs to develop very modular hardware descriptions. A MODLAN description consists of a hierarchical block structure of modules and submodules. A module can be one of three types: 1) Structural module. 2) Functional module or 3) A controller module. Communication between modules is carried out via links and terminals.

MIMOLA [Zi79] is a procedural language used as the primary component for the "top-down" MIMOLA design system. The MIMOLA language is suitable for use as a CHDL of the RT level and as a HLL to describe algorithms. The algorithmic descriptions are similar to the instruction level of the design hierarchy. Both synchronous and asynchronous operations and concurrent activities can be described by this language. Support software for this language has been implemented in PASCAL.

PHPL [AF79] (Parallel Hardware Processing Language) was developed for the description and simulation of hardware at the gate, register transfer and architectural levels. It is intended for use by logic designers and computer architects for design verification, debugging and documentation [ST81]. Descriptions can be made of synchronous or asynchronous operations. Statement execution depends on explicit control event labels. A control event may depend on logic levels, periods of stability or high or low transitions.

S_A* [Da81] is a language for describing computer architectures. This language deals solely with the PMS level of a digital design. S_A* breaks a system's architecture up into two levels. The first level is called the exo-architecture, it is concerned with the logical structure and capabilities of the system as visible to the user. The second level is called the endo-architecture, which includes the functional capabilities of a machine's physical hardware. This second level is equivalent to the PMS level of design. Both sequential and asynchronous systems can be described with S_A*.

LALSD [SB75,BS71,So73] (A Language for Automated Logic and System Design) allows a system to be decomposed into subsystems that can be independently designed. Descriptions can be made at the gate, register transfer and architectural levels. This language has been upgraded by the introduction of the LALSDII language which is useful for multi-level descriptions and simulations [SH81,SF81]. All of the CHDLs that have been discussed in this section are summarized in Table 3.

NAME	REFERENCES	LEVEL OF DETAIL				B/F/S[1]	P/NP/SNP[2]	S/A[3]	USED IN DESIGN SYS.	ADAPTED FROM	LANGUAGE IMPLEMENTED IN	MACHINES IMPLEMENTED ON	DATE FIRST INTRODUCED
		GATE	RT	PROG.	ARCH.								
ADLIB	(Hi79,Hi80, C981)	✓	✓		✓	B	NP	S/A		PASCAL		SABLE	1979
AHPL	(HP73,Hi74)	✓	✓			F	SNP	S		APL	SNOBOL FORTRAN	CYBER 175 DEC-10	1971
CDL	(Ch65,Ch70, Ch72)	✓	✓			F	SNP	S		ALGOL	FORTRAN-IV	IMB 370 B7000	1965
DDL	(Dv67,DD68)	✓	✓		✓	F	NP	S/A			FORTRAN	UNIVAC 1108	1967
FLOWWARE	(Om73,CT77)		✓			B,F	P	S		CDL FLOWCHARTS	NOVA ASSM. PL/1	NOVA 800 IBM 360/50	1973
ISPS	(BB78,Ba79, Ba81)			✓		B,F	P	A			BLISS	PDP-11/70	1978
KARL	(Ha77,Hv79)	✓	✓			F,S	NP	S/A		CDL ABL			1977
PMS	(BN70,BK72, Ku73)				✓	B,S	P	A			SNOBOL	PDP-10	1973
SDL	(va77,Co81)	✓	✓		✓	S	P					SABLE	1977
SLIDE	(PA80,PW81)	✓	✓		✓	B,F	NP	S/A		FLD GLIDE	SIMULA-67		1980

NOTES:
1. Behavioral/Functional/Structure descriptions emphasized
2. Procedural/Non-Procedural/Strongly Non-Procedural Sequencing Mechanism
3. Syncronous/Asynchronous timing sepcified.

CHDL Summary Table
Table 3

NAME	REFERENCES	LEVEL OF DETAIL: GATE	LEVEL OF DETAIL: RT	LEVEL OF DETAIL: PROG.	LEVEL OF DETAIL: ARCH.	B/F/S[1]	P/NP/SNP[2]	S/A[3]	USED IN DESIGN SYS.	ADAPTED FROM	LANGUAGE IMPLEMENTED IN	MACHINES IMPLEMENTED ON	DATE FIRST INTRODUCED
APDL	(Da68,Da69)		✓			B,F	P	S/A		ALGOL	ALGOL-60	CDC-620	1968
CASD	(Cr70)		✓		✓	F	NP	S/A		PL/1	P/L1	IBM 360	1970
CASL	(M079)		✓		✓	F,S	NP	A					1979
DTMS	(TS80,Si81)	✓	✓			B,F	P/NP	S/A					1980
FST	(Fr67,B171, B172)	✓	✓			F	P	S			FORTRAN-IV	IBM 360	1967
FTL	(Sc79)	✓	✓			F	SNP	S/A		CDL			1979
HILO MK2	(FM81)	✓	✓		✓	F,S	P	S/A		HILO	BCPL	SEL-68	1980
LALSD	(SB75,SH81	✓	✓	✓	✓	B,F	NP	S/A		PL/1	PL/1 SNOBOL	IBM360/91 CDC 6400	1975
LCD	(EG76,EG77)		✓		✓	F,S	SNP	S		RTL			1976
LOGAL	(Lu73,St77)		✓			F	NP	S			FORTRAN-IV	UNIVAC 1108	1973

CHDL Summary Table
Table 3 (con't)

NAME	REFERENCES	LEVEL OF DETAIL				B/F/S[1]	P/NP/SNP[2]	S/A[3]	USED IN DESING SYS.	ADAPTED FROM	LANGUAGE IMPLEMENTED IN	MACHINES IMPLEMENTED ON	DATE FIRST INTRODUCED
		GATE	RT	PROG.	ARCH.								
MIMOLA	(Z:79)		✓	✓		F,S	NP	S/A			PASCAL	IBM 370	1978
MODLAN	(Pa76,Pa81)	✓	✓	✓		F,S	P/NP	S/A		FDL	FORTRAN-IV	ICL 1900	1980
PHPL	(AF79)	✓	✓		✓	F	SNP	S/A					1979
S_A*	(Da81)				✓	B,S	P	S/A		S*			1980

CHDL Summary Table
Table 3 (con't)

4. AN HDL SELECTION APPROACH

At this point, it should be apparent to the reader that a large number of languages (HLLs, GPSLs and CHDLs) are available for the description and simulation of digital hardware. The selection of a single language that is well suited for a particular design environment can be a difficult and unrewarding task if an organized decision making approach is not used. Picking an HDL at random, studying its features and constructs and then making a determination as to whether or not the language should be used in a design project is not an efficient selection approach. Clearly, some type of comparison process must be used. This chapter presents an HDL selection approach that relies on two distinct levels of comparisons.

An important preliminary step in any decision making process is to insure that a decision does indeed need to be made. In the case of selecting an HDL this involves considering if the use of an HDL-based design approach would be advantageous in a particular design environment. The advantages and disadvantages of using HDLs for design purposes are reviewed in Section 1 of this chapter. These will be presented in a brief form as it is assumed that the reader has already considered these points and is now interested in selecting an HDL.

If it is desired to use an HDL in design work then a search process to find an appropriate language can begin. The selection approach presented in this chapter is a two step process. The first step is to select one of the three classes of HDLs as being best suited to meet the goals of a design system. This portion of the

selection approach is discussed in Section 2. Step two is to choose an individual HDL from this class of languages. This second step is itself split into two groups of selection considerations. The first group consists of "technical" considerations and is used to determine a small pool of appropriate languages. These considerations are discussed in detail in Section 4.3. The second group involves considerations of a "practical" nature, and is used to make the selection of a single language from the pool of applicable HDLs. Practical considerations for the selection of an HDL are presented in Section 4.4.

Throughout this chapter an example depicting the selection of a language for use in an academic environment is carried out. At the end of each section the results of the section are applied to the example thus arriving at the selection of a single HDL at the end of Section 4.

The type of HDL needed for an academic environment is very similar to what might be required for an industry environment, although cost is more of a consideration for a university. The HDL should be capable of describing and simulating hardware at the register transfer and architectural levels. It should emphasize functional and behavioral information rather than the structural aspects of a design. The HDL should use a modular description format, so groups of students can work on individual modules with relative independence. It would also be beneficial if the sequencing mechanism was not dependent on the location of these modules in a design description. Other requirements of this

language will be discussed as they become applicable.

4.1. Should an HDL be Used?

The use of a specialized notation to describe hardware has been a practice of the electronics industry since 1939 when Shannon started his work on digital switching circuits. These initial descriptions were carried out at the circuit, and later at the logic levels. As VLSI designs became the reality of everyday design, there was an increasing need to describe hardware at a design level that was higher than the logic level. This higher level is the register transfer level, which deals with data transfers between system registers according to a specified set of rules. It can also be useful to carry out descriptions at even higher levels of the design hierarchy, such as the instruction and architectural (PMS) levels. Several classes of computer languages are capable of describing a digital system at one or more of these levels.

> "A computer designer can benefit from using a design language at a higher level just as a computer user can benefit from a higher level programming language" [Ch72b].

Among the many advantages for using an HDL at the register transfer level (or at a higher level) are:

1. A precise, yet concise design description is provided.
2. A standard format of documentation is developed.
3. Communication between interested parties is improved.
4. Hardware descriptions can be simulated on a computer.
5. Descriptions can be used as inputs to automatic design systems.

6. Design costs are reduced.

7. A designer's creativity is enhanced.

8. The HDL is useful in an educational environment.

The combination of top-down design techniques with complex VLSI design projects has created a real need for precise hardware descriptions that are simple enough to be easily comprehended by designers. Typically on this type of design project the overall system will be split into several semi-independent hardware modules. Each member of a design team will then be responsible for the design of at least one of these blocks. In this situation it is important that all members of the team have a compatible picture of the overall system requirements and precise descriptions of the functions their subunits should provide.

An HDL description at an architectural level can provide the needed system information and abstract the unnecessary detail that can lead to misunderstanding. The ability to suppress information about the lower design levels is especially useful in the development of design specifications where the details have not yet been designed. Once the design team fully understands the functions their individual subunits are to provide, the team members can work at their individual pace.

Without having a specific HDL to use for documentation, the members of the design team are likely to use their own notations to document their design work. This can make communication between the various designers difficult and confusing. If a common HDL were used by all of the designers, communication between them would

be encouraged and easily accomplished. A consistent notational scheme could also prove beneficial when it becomes time to connect the individual subunits together and complete the overall design.

The final documentation of a design would also be made much simpler if a multilevel HDL description of the entire system was made. With the use of a standard HDL notation, users' guides and technical manuals could be developed by people less familiar with the design. There have been some studies done in the area of automating this documentation process based on a hardware description [Br72,BS75].

Another important aspect of the design process is to insure that the completed design works as expected. This can be done by constructing a prototype of the system and observing its behavior. This process can be expensive; especially if design changes cause more than one prototype to be built. Another drawback to this approach is that a designer must often wait until the entire system has been assembled to test his particular subunit. Using an HDL that has simulation capabilities can alleviate many of these testing problems.

Most current HDLs have developed a simulator that allows the usage of the hardware description as an input. Simulation of digital designs can save money by reducing the number of hardware prototypes that must be constructed. The most valuable attribute gained by digital simulation is that design feedback can be carried out at a much earlier stage in the design. For example, the early architectural view of the system can be simulated to observe system

performance and any possible bottlenecks. Subunits can be simulated in a detailed fashion while the rest of the system is specified very abstractly and cast in a supporting role. Simulation can be a valuable design tool that should not be underestimated.

In addition to simulation, many other design-related services can be performed based on a complete hardware description. The automatic translation of a description into logic level components, wiring lists, layout diagrams and IC mask specifications are all features of some of the available computer-aided design (CAD) systems that are based on HDLs. The development of these CAD systems is another popular research area [Th81,Hv79,Su73,Br72].

Using an HDL for any of the stated reasons will increase the efficiency of a design process. This efficiency will manifest itself as increased design throughout, reduced man-hours per design and cost reductions. If an entire automatic design system is implemented, the savings in design costs can be dramatic (of course a CAD system is not free).

The use of an HDL notational scheme can enhance a designer's natural creativity by allowing experiments with design ideas that would not usually be tried due to a lack of time. With the aid of an HDL these ideas can be expressed in a form easily read and understood by co-workers, possibly stimulating their creativity as well. It is difficult to put a dollar value to this feature but it is obviously a positive aspect of HDL use.

Another important area in which an HDL can play an effective role is in the academic environment. Very few schools can afford

to have students actually build the hardware systems they have designed in class. If a simulation facility existed, the students could still perform the required design work, then describe their design in terms of an HDL and simulate it to verify its correctness. This system would teach students more about design during their time at college and give industry better-prepared designers.

Thus it seems that the use of an HDL to describe digital hardware at the register transfer level is the answer to many of the design industry's woes. Many companies have already accepted this fact and are currently doing in-house research to develop a useful HDL-based design system. However there are some negative aspects that should be considered before jumping on the HDL bandwagon. Many of these points are related to a design system's cost, they are:

1. Implementing a new design system can be expensive.
2. Designers must spend a significant amount of time learning a new language and design system.
3. System bugs must be discovered and fixed, often with a trial-and-error approach.
4. A company's typical design project may be too small to be cost effective on an HDL-based system.
5. The abstraction of design information may cause problems with design implementation.

The first three disadvantages are self-explanatory, while the last two deserve further consideration.

If a design company typically works with design projects that require a large portion of the system descriptions to be made at the

logic level, then the use of a register transfer level HDL will not be cost effective. Logic level simulation languages might be a better choice. Design projects should be computer system related to receive the most advantages from HDLs.

Another problem with using HDLs is that too much information may be lost by working at levels higher than the level where actual implementations are made. For example, an IF-THEN-ELSE statement in an HDL might be implemented as a particular AND-OR logic circuit. However, it could also be implemented as a multiplexer or decoder chip. As long as a designer realizes that this situation can occur then this disadvantage can be easily lived with.

After discussing the benefits and drawbacks of using an HDL, it seems most designers choose to use an HDL. All in all, the opportunity to simulate computer designs at the register transfer level is not usually rejected by most designers.

4.2. Selecting a Class of HDL

The first step of selecting an HDL is to determine which class of language will best meet the goals of a design system. The selection of an HDL class is not a particularly difficult decision as there are only three classes to choose from. However, the selection of a class is important as it focuses a designer's attention on the languages that will work best in his design environment.

Making a class selection involves weighing the advantages and disadvantages of each of the three classes and selecting the one

most beneficial for the design environment. The advantages and disadvantages of HLLs, GPSLs and CHDLs have been previously presented in Chapter 3 and will only be stated in a summary form here.

High-level languages can be useful for hardware description and simulation applications as they provide most of the language constructs necessary to describe digital hardware. Using an HLL to describe digital systems is beneficial for the following reasons:

1. An HLL description can be simulated with the use of an HLL compiler.
2. Implementing an HLL design system is relatively inexpensive as HLL software costs are lower.
3. HLLs are very popular and frequently found in industry.

On the other hand, there are some serious drawbacks to using HLLs to describe hardware, these are:

1. HLLs lack the essential language constructs that are necessary to provide a precise hardware description.
2. HLL descriptions are long and often confusing.
3. The structure of an HLL description does not represent the structure of the hardware it represents.

An HLL can be best used to describe digital hardware if cost is a major concern or it is the only type of language available for use. However, the description itself will be both lengthy and imprecise, a combination not recommended for most design environments.

General Purpose Simulation Languages can also be useful for hardware simulation. These languages are especially useful at the architectural level of design, where advantage can be taken of

their statistical nature. The benefits to using a GPSL are:

1. The languages specialize in simulating nondeterministic systems, such as a computer system modeled at an architectural level.
2. GPSLs provide a statistics package that allows the occurrence of random events and the collection and evaluation of statistical data.
3. The languages are well publicized and documented (although often for non-engineering applications).

GPSLs, like the HLLs, also have serious drawbacks that usually preclude the selection of a GPSL for design purposes. These disadvantages are:

1. Very little structural information can be represented, even at an architectural level.
2. These languages have few facilities for dealing with deterministic systems, such as a computer system modeled at the register transfer level.

The GPSLs are ideal tools for the simulation of an architectural level description if only performance information is hoped to be gained from the simulation. The use of a GPSL for any other hardware description related task is recommended only if no other alternatives exist. The use of a GPSL that can also be used as a high-level programming language, such as Simscript II [Ki81], will facilitate the development of register transfer level descriptions. If a GPSL must be used for multi-level description purposes, it should be one from this more flexible subclass.

Computer Hardware Description Languages have been developed expressly for the purpose of describing and simulating digital hardware. As such, they are by far the best suited for this type of design work. The most important benefits to using a CHDL as a design aid are:

1. They permit a precise, yet concise hardware description.
2. They provide an efficient means of communicating and documenting designs.
3. They are amenable to use for design simulation purposes.

The disadvantages of using a CHDL are quite similar to those for using any HDL. The most important of these is their relative cost and the abstraction of too much information to provide an efficient design structure. In spite of these drawbacks the CHDL class is usually found to be the best suited for hardware design tasks. For these reasons the selection of an individual CHDL will be emphasized in the remaining sections of this chapter. The following statements are made as a summary for the selection of an HDL class.

1. HLLs are recommended if low cost is important, or if they are the only choice (i.e., some HLL simulation is better than no simulation).
2. GPSLs are recommended for performance studies of algorithmic level design descriptions.
3. CHDLs are the best choice for virtually all other digital design environments at or above the logic level.

Selection Example (Part A.)

The CHDL class has been chosen in this selection process example as it is the only class that meets both the multi-level of description and description modularity requirements.

4.3. Selection of an Individual HDL: Technical Considerations

After a particular class of HDL has been chosen as the most suitable for use in a design system, an individual language from that class must be selected for implementation. Before making a choice of a specific language there are several language-related points that should be considered. This section will list and discuss the nature of these considerations. This discussion will not be an evaluation of individual languages but rather it will present key concepts that should be kept in mind when making a language selection.

The considerations raised here have been selected to allow an engineer to make a fitting choice of a description language while having to evaluate only a small number of criteria. Basing a selection process on a set of criteria is a common and efficient approach to making decisions. Specific criteria for each separate design system will vary with the scope and type of design work to be performed. The considerations raised in this section are general in nature and should be used as a guideline for developing a set of specific criteria.

The group of considerations discussed in this section are called "technical" considerations as they deal with universal characteristics

of a language, such as its level of description and its sequencing mechanism. These technical considerations are aimed at making a CHDL selection rather than at choosing an HLL or GPSL. This has been done for two reasons: first, CHDLs seem to be the best suited for most hardware description tasks; and second, when a specific HLL or GPSL is selected it is usually because it is already available on the job site.

The considerations presented in this section are split into primary and secondary groups. The primary technical considerations are the four universal characteristics of CHDLs used to create the four-dimensional CHDL space discussed in Chapter 3. Once again these four considerations are: the level of description, the type of description, the sequencing mechanism and the timing specification. Each of these is discussed in regard to the use as a basis for the development of design system criteria. The four secondary considerations are also important but are deemed to be dependent on the "values" of the first four considerations/dimensions and are not discussed in as much detail.

Both groups of consideration are useful aids for selecting an individual CHDL. Toward this end the following procedure is recommended:

1. Read each of the eight considerations presented below.
2. Using the four primary considerations develop a set of criteria (i.e., standards on which a judgement or decision can be made [Wo77]) that represents the goals of the proposed CHDL-based design system.

3. Convert the set of criteria into ranges of acceptable CHDL features that can be represented as "lines" on the respective axes of the CHDL-space. From these lines a four-dimensional subspace can be defined. Any of the points within this subspace (figure) would successfully meet all four of the design criteria.
4. Overlay the CHDL-space figures representing an individual language's range of features. If there is any intersection of the four-dimensional figures, then that particular language could be used to satisfy the design system criteria. The larger the intersection the better the individual CHDL will meet the design goals.
5. Using the above method find several languages that can satisfy the design criteria. This will form a pool of suitable languages available for further consideration.
6. Then use the four "secondary" considerations in a case-by-case examination of how each language in the pool matches up with criteria based on these considerations. Any language that is no longer appropriate for use should be removed from the pool. Evaluations should be "tough" enough that only a small number (2-4) of languages remain in the pool.

At the end of this process only the languages that can best describe digital hardware in the manner required will remain in the pool. The use of this process is present in Part B. of the selection example.

Level of Description

The first dimension of the CHDL-space is based on the previously discussed hierarchical design levels. CHDLs were initially developed to raise the level of abstraction in a design model to the register transfer level. In recent years some CHDLs have been developed that specialize in describing digital systems at levels other than the register transfer level. These other levels include the logic, instruction and architectural levels. The level at which most design work will take place is an important criterion to use in the selection of a CHDL. If most design work will be carried out at an architectural level, then a language that emphasizes architectural descriptions should be selected. A language that provides descriptions only at a single design level would be shown as a point in this dimension at the appropriate level of detail.

There is currently a move toward using mixed-level descriptions [SH81,Zi79,Hi79,Hi80]. This could involve describing sections of a design at a detailed register transfer level while the remainder of the design is described in an abstract architectural fashion. This type of description can be useful in many design situations. One example of this might be in simulation where only a single module needs to be tested in a detailed fashion. The remaining sections of the design are only needed to provide stimulus and data for the module to be tested and can be specified at the architectural level. The same situation could be formed with any combination of design levels. A language that is suitable for mixed-level studies should be capable of describing digital systems at several levels of

detail, although the reverse is not necessarily true. A suitable language should also provide convenient interfacing between modules at different levels. A language that can support hardware descriptions at several levels of detail would be represented as a range of values or a "line" in this dimension.

Type of Description

The second dimension of the CHDL-space is the type of description emphasized by a language. Some CHDLs have been developed specifically to provide structural information (i.e. SDL) and have no facilities for indicating behavioral information. Just the opposite is true for several other languages (i.e., ADLIB and ISPs). The majority of CHDLs fall somewhere in between these two extremes and are capable of describing both structural and behavioral information in varying degrees. Included in this large middle ground are functional descriptions.

Structural descriptions are used to represent a system in terms of actual hardware components. The elements of this level are typically user-defined hardware units such as logic gates, flip-flops and IC packages. Interconnection of the structural elements is stressed in this type of description. Structural descriptions are the closest representations of the actual hardware.

Functional descriptions are a middle ground between structural and behavioral views. At this level of abstraction, low-level structural details are suppressed so that a designer deals with registers and logic networks rather than individual flip-flops and logic gates. The behavior of a system is described as an algorithm

involving the register transfer level components of the system. The actual data transformations, both content and order, are stressed in this type of description. Timing and concurrency studies can be carried out to a fine degree of accuracy at this level. Most CHDLs offer a range of functional descriptions by using various degrees of structural, behavioral and temporal abstraction.

Behavioral descriptions offer a very high level of abstraction. Descriptions are algorithms that express the activities occurring in a system. Complex expressions and operators are used to simplify behavioral descriptions and often have no direct correlation to the actual hardware. This type of description stresses the order in which activities occur but not necessarily the data involved with those activities. As an example of this consider a CHDL statement that represents the contents of two storage locations being added together. While an adder is implicitly declared by an addition operator no specific details about it are implied (such as 1's complement, carry-look-ahead, etc.). Timing details are not usually dealt with at this level of description.

It should be decided before a particular language is selected which of these three types of descriptions is required by the design goals. CHDLs that represent only one type of description are themselves represented as a point in this dimension. Languages that span either a range of description types or a range within one particular type, as frequently is the case for a language possessing functional description capabilities, are indicated as a range of points in this dimension.

Sequencing Mechanism

The third dimension of the CHDL-space is the type of sequencing mechanism used by a particular language. The sequencing mechanism of a language is that process which allows execution of language statements in the correct order. CHDLs use either a non-procedural or a procedural sequencing mechanism.

Non-procedural languages require the use of control variables to describe the order of execution for a system description. These control variables are used to enable or inhibit the execution of a statement. Sequencing is performed by modifying selected control variables, enabling the next statement to be executed. The modifications of these control variables can be made in an explicit manner and included in the operations performed by a statement (as in AHPL) or in an implicit manner and generated by an independent clock (CDL) or finite state machine (DDL). In non-procedural languages the relative position of statements is not important because the control variables are used to assure that the proper order of execution is achieved [Ba75].

Procedural languages are similar to programming languages and their method of sequencing. In this type of description, statements are placed in the exact order they are to be executed [Sh79]. By grouping statements that are to be executed at the same time into special syntactic blocks, parallel actions can be described. Sequential actions are then expressed as lists of these blocks placed in the order they are to be executed. This type of language suppresses details of control hardware because that information is

not needed to execute (or simulate) a procedural description. For example, the procedural language ISPS has no facilities to explicitly declare any sort of clock [Ba81].

Some CHDLs that utilize a block-structured description format allow a hybrid of these two sequencing mechanisms. The total description is treated as a non-procedural unit and blocks may be executed in any order. Internal to the blocks a procedural sequencing mechanism is used to simplify the description. In this situation the relative ordering of statements within a block is important but the ordering of the blocks themselves is not of concern. This type of description will be referred to as a non-procedural description and the case where every statement is activated via a control variable is now called "strongly" non-procedural [ST81].

The sequencing mechanism used by a CHDL can be of great importance when deciding on the suitability of a particular language. Strongly non-procedural languages can represent detailed control and timing information. Thus they are useful for modeling control circuitry and performing timing verification studies. Procedural languages are more useful for dealing with new designs where specific timing and control details are of no concern to designers. Non-procedural languages provide a happy medium between these two extremes. As before, a CHDL can be represented as either a point or a line in this dimension.

Timing Specification

The fourth dimension of the CHDL-space is the type of timing specification or mode a language uses. The timing mode refers to

whether a CHDL can describe synchronous or asynchronous circuits. Some languages have been developed that allow both types of circuits to be described (i.e., SLIDE).

A synchronous language uses a signal from a clock structure to activate the statements of a description. A clock is used to specify a standard amount of time allowed for the action called for by each statement. The clock period must be longer than the time it takes for the longest instruction to execute. Several parameters can be specified in this type of language to allow very detailed timing studies. Some of these parameters include a clock's phase, pulse width, duty cycle and period, propagation and setup delays and trailing edge, leading edge and level triggering specifications.

Asynchronous languages use a signal from a preceding activity to activate the next statement for execution. Asynchronous systems do not require a clock for statement activation. Asynchronous descriptions are usually more complex as they must describe concurrent actions that neither start nor stop at the same times. Detailed timing studies can be carried out with this group of CHDLs but some special control structures are needed [De70,PD72]. These structures include such things as ready and acknowledge signals and facilities for combining multiple ready signals or sending multiple acknowledgements. Some asynchronous languages suppress the timing details completely and are grouped as an asynchronous language only because the activation of description statements is dependent on previously executed statements rather than on a clock. ISPS is a prime example of this type of

asynchronous language.

Both types of timing modes are useful for digital design work. At the register transfer level synchronous operations are the most common. At the architectural level asynchronous operations are more frequently found. One CHDL, SLIDE, specializes in timing studies of I/O related hardware using both of these timing modes. A language that can describe both synchronous and asynchronous circuits or a range within a single mode is represented as a line in this dimension.

Secondary Considerations

There are also four secondary considerations about CHDLs that should be briefly discussed. These considerations are dependent on one or more of the four CHDL-space dimensions. While the previous four dimensions characterize CHDLs using discrete levels, these new considerations provide a continuous spectrum of relative values. The presence and type of these considerations are dependent on a language's location in the CHDL-space. Four of this type of CHDL characteristics have been selected as providing the most supplemental information to be combined with the earlier considerations thus allowing the best technical selection. These four new considerations are:

Secondary Consideration	Primary Considerations Dependent Upon
1. amount of timing information	(level, type, sequence, time)
2. amount of block structuring	(level, type, sequence)
3. abstraction of data types	(level, type)
4. complexity of operators	(level, type)

The amount of timing deals with how much timing detail is included in a system description. CHDLs can vary from providing very little timing information to providing very extensive models of timing characteristics. AHPL is a procedural, synchronous CHDL that allows very little explicit timing information. On the other hand FTL is a strongly non-procedural, synchronous (and asynchronous) language that allows for greater detail of timing specifications than is possible with most other CHDLs [Sc79].

Another CHDL characteristic that presents a spectrum of values is the amount of block structuring in a language. The addition of a block structure to a language is a straightforward way to describe the existence of several levels of detail in a design. Early CHDLs used a single block structure where all system facilities were declared at the same level of detail (i.e., CDL). More recent languages use an ALGOL or PASCAL-like hierarchical block structure. The ability to work with designs using blocks at different levels of detail is dependent on the language's ability to describe systems at several levels of detail. Using a block structured language is very advantageous for mixed-level simulation studies [SH81].

A third CHDL characteristic that provides a range of choices is the degree of abstraction of data components by a language. Abstract data types allow a design to describe a hardware system without being concerned with the actual structure of the data types. Some CHDLs (ISPS) provide only a few generalized data types that suppress most specific implementation details. Other languages use a large number of explicit data types that imply a significant

amount of physical structure. A CHDL that provides abstract data types can be used more easily at different levels of abstraction which results in a better model for simulation.

The complexity of operators used by a language is yet a fourth characteristic that has a wide range of variations among CHDLs. An operator is an entity that is used to transform data according to a pre-defined rule. Operators are often used to suppress structural information in a design description. For example two sequential data transfers from separate registers to a common register could be denoted as follows:

$$R \longleftarrow A,$$

$$R \longleftarrow B,$$

The actual hardware may require using a multiplexer or a bus between the source and destination registers but this is not implied by the language statements. Another example of operator abstraction is the previously mentioned addition operand and its lack of implied detail about the hardware implementation. Thus an operator can range from a specific logic function to a very abstract combinational network. The use of abstract operators at the early (high level) stages of a design is beneficial since hardware implementation is not an issue. At later stages of the design less abstract operators can be used to express more of the actual hardware structure.

While it is conceivable that other "technical" considerations could be used in this process, the first four are clearly fundamental to a design system environment and the second four provide all of the supplemental information needed for most design environments.

Selection Example (Part B.)

This portion of the example for selecting a CHDL suitable for use in an academic environment will follow steps 2 through 6 of the recommended selection approach.

STEP 2: Develop a set of criteria based on the four "primary" considerations.

1) Level of Description - the language should describe systems at the architectural and register transfer levels. Descriptions at the programming level are allowed but logic level facilities are not desired. Due to ranges of allowed capabilities, "high-level" architectural and "low-level" register transfer level descriptions will be allowed.

2) Type of Description - the primary area of simulation for this desired language will be for functional analysis. Structural and behavioral information is tolerated but it is desired to place more emphasis on the behavioral side.

3) Sequencing Mechanism - there are two conflicting goals for this consideration. It is easier to write descriptions in a procedural fashion. However, since students will be working on hardware modules independently there is a need to have sequencing carried out in a "strongly" non-procedural manner. The solution is to compromise and use a non-procedural language that allows

procedural sequencing internal to the modules and non-procedural sequencing external to them. A range of the degree of "non-proceduralness" is biased towards "strongly" non-procedural in order to include more control hardware description.

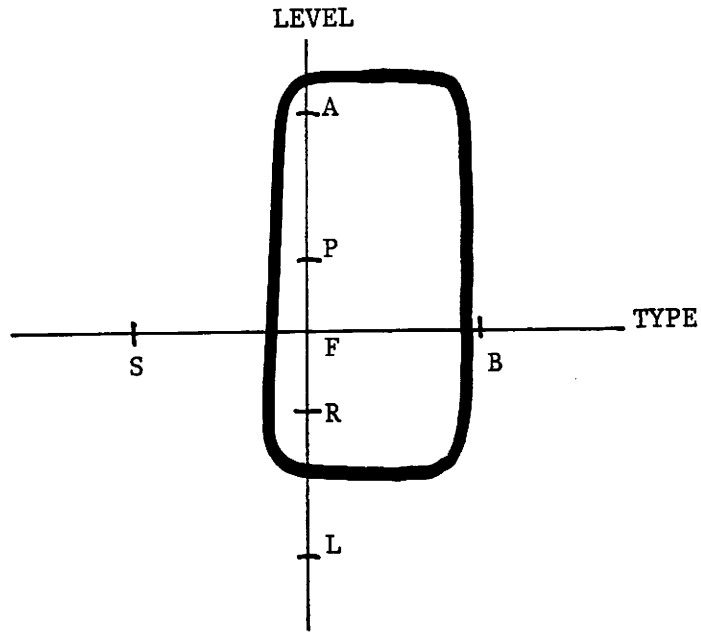
4) Timing Specification - only synchronous timing mechanisms are needed for this implementation, but asynchronous mechanisms can be tolerated.

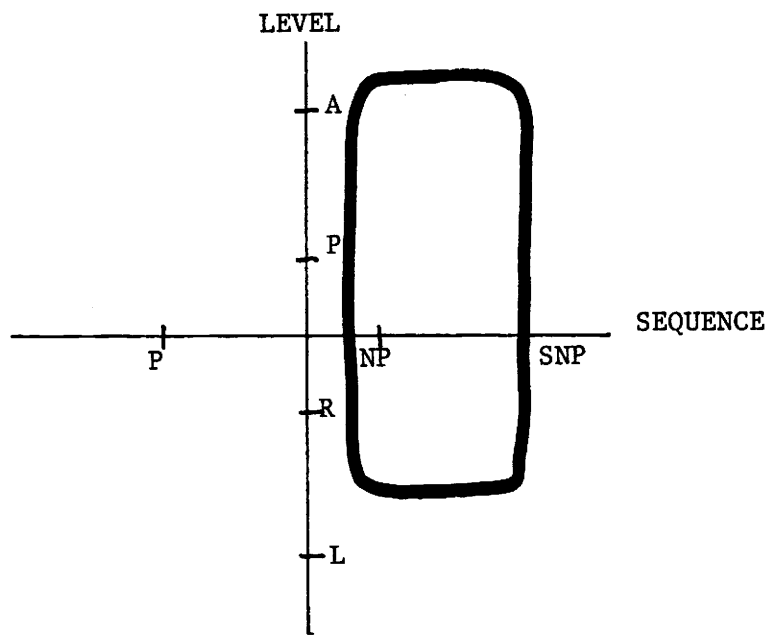
STEP 3: To simplify the graphical views of these criteria two two-dimensional plots will be used rather than the three-dimensional plots presented in Chapter 3. The Timing Specification axis will also be omitted as it is a "don't-care" in this example. The two plots and the representation of the design criteria on them are presented in Figure 29.

STEP 4: In Figures 30, 31 and 32 three separate CHDLs are overlayed with the design criteria. The CHDLs used are CDL, ISPS and DDL.

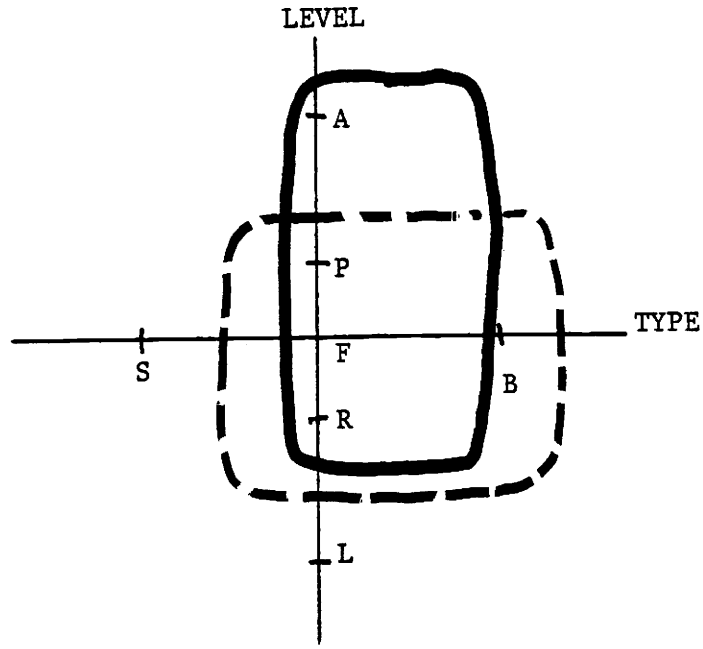
STEP 5: While all three languages match up fairly well in upper plot, only CDL and DDL have all of the features required to satisfy the design criteria.

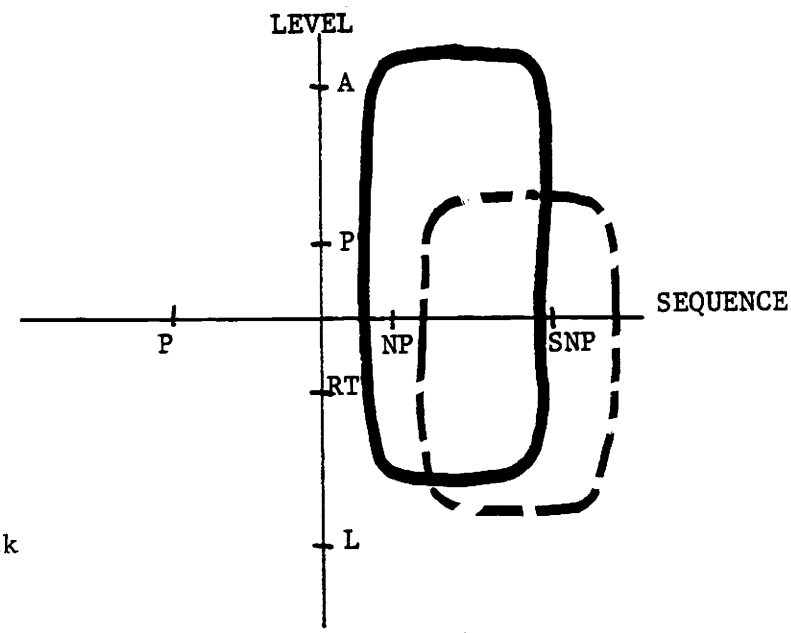
STEP 6: Use of "secondary" considerations to narrow down the number of possible choices. Since there are only two languages remaining in the pool this step will be skipped. If it were not skipped, DDL could get the nod due to the presence of block structuring.





Two-Dimensional Views of Example Design Criteria
Figure 29

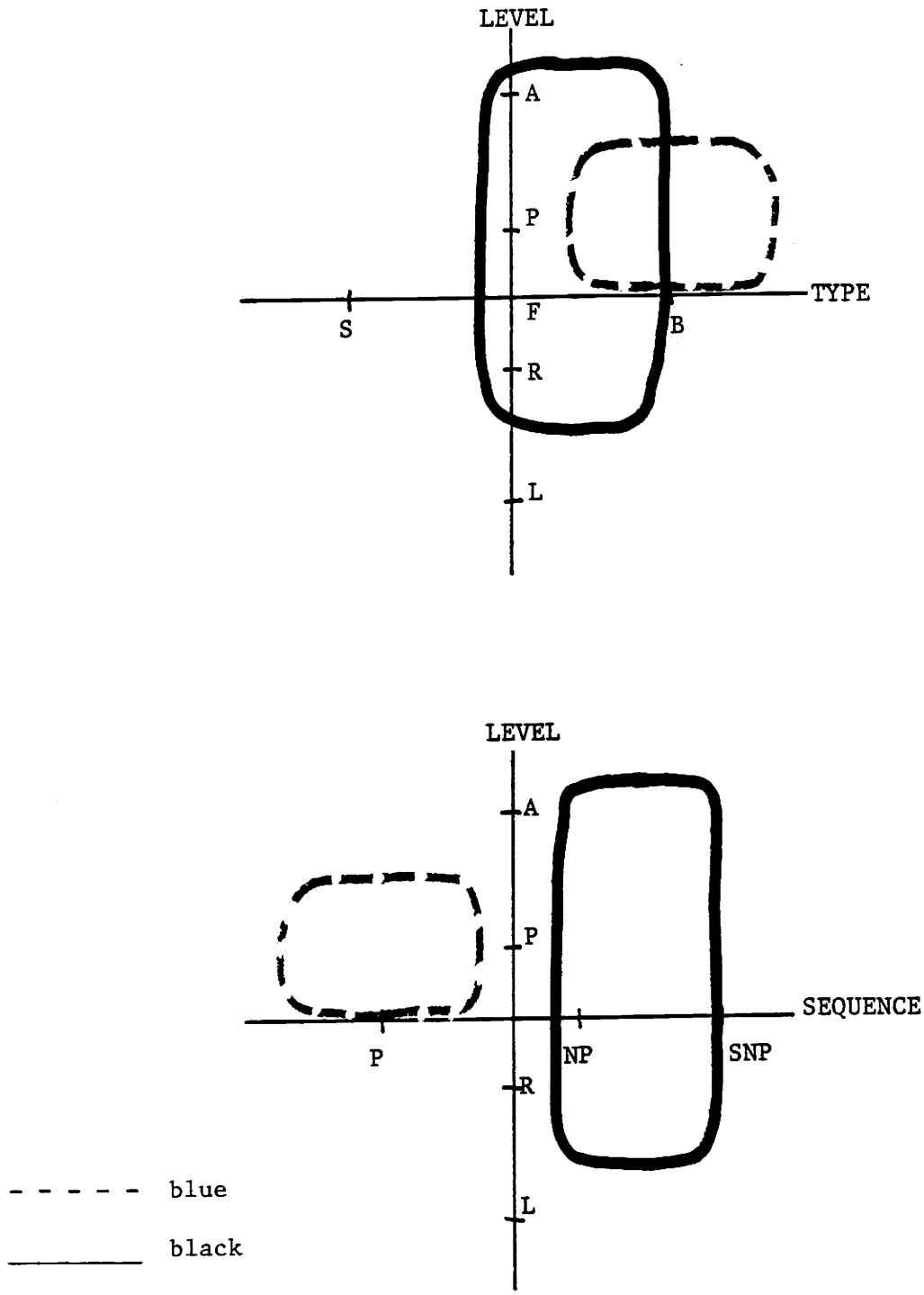




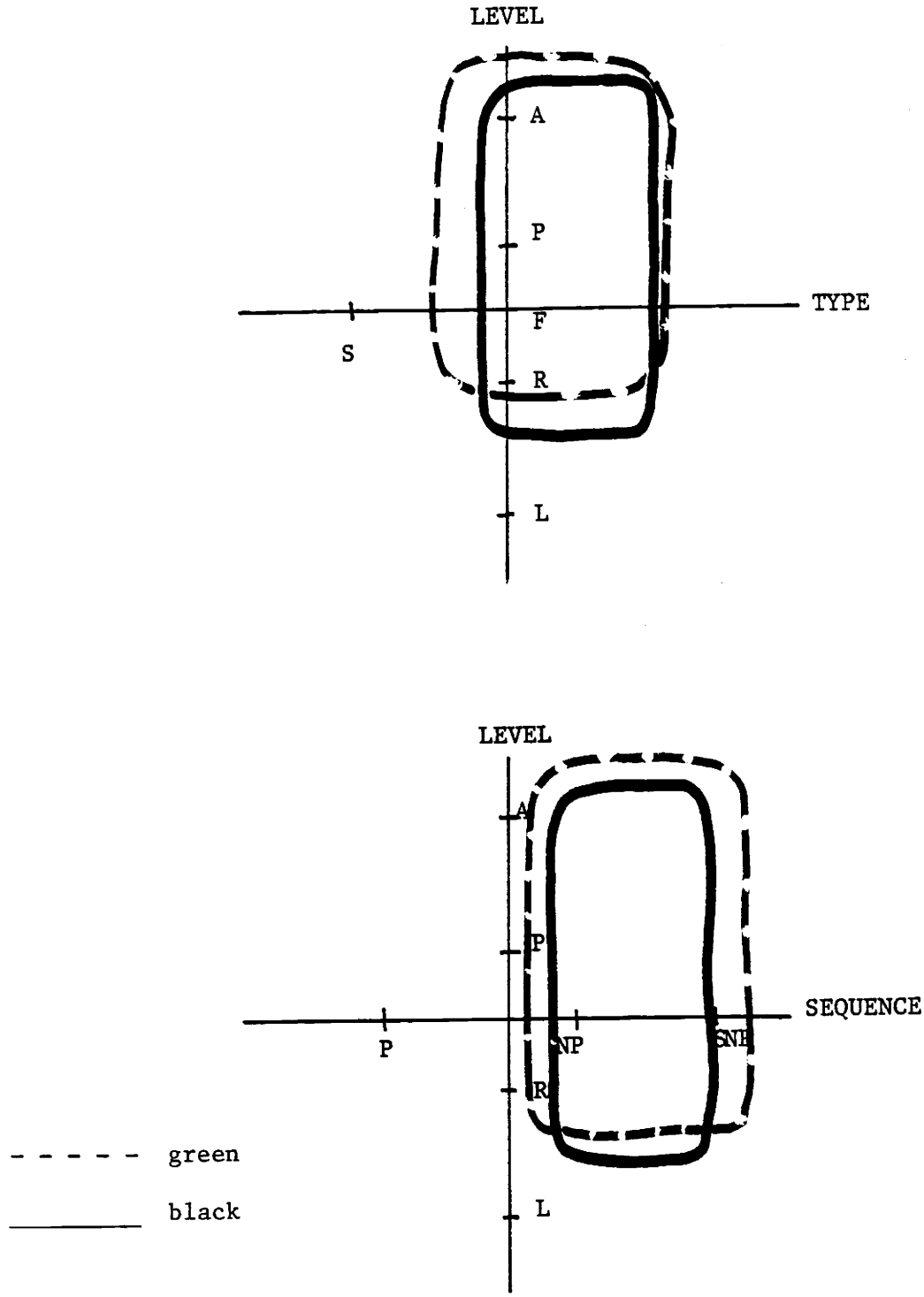
- - - - - red

_____ black

CDL Overlaid With Design Criteria
Figure 30



ISPS Overlaid With Design Criteria
Figure 31



DDL Overlaid With Design Criteria
Figure 32

Now only one part of the selection process remains. This is to reduce the pool of languages by considering some practical aspects of the individual CHDLs.

4.4. Selection of an Individual HDL: Practical Considerations

A small group of CHDLs (2-4) have been discovered to be suitable for use in a design system through an evaluation of technical considerations. Typically only one of these languages can be implemented in a design system. A set of practical implementation "issues" should be used to develop an "order of preference" for the small group of CHDLs. While the technical considerations are used to insure that a CHDL is able to represent digital designs in the formats desired, these practical considerations are used to assure that a language can be efficiently implemented.

The following group of "issues" is used for practical consideration:

1. other usage of the language
2. readability
3. generality
4. simulator availability

These are aspects concerned with making sure that a language can be used by designers in an easy and efficient manner. Each is important to consider when trying to select the best language.

The first consideration involves the industry wide use of a particular language. It is important to select a language that is being used in other design systems to receive the benefits of

already existing software support and any language extensions. Improvements in simulator and translator designs will be facilitated by the mutual interest of several independent groups of users. Languages that have never been implemented in design systems will invariably have many "bugs" that need to be worked out. If a language is selected that has several generations of software support packages, the system "bugs" should be nearly eliminated.

The next consideration, readability, involves the ease of using a language. This includes reading and writing hardware descriptions in a particular language. A description language is used for transferring information both among humans and between humans and machines. Some of the people who read a design description may be only slightly involved with a design project. These people should be able to completely understand the machine being designed just by examining the hardware description. Thus a hardware description should provide all of the information needed about a design. The notation should be very precise and as short as possible. Design information should be extracted from context rather than syntax which can cloud a description. Since a description language is primarily used by designers, it should be simple enough to be written by the designer rather than a computer programmer. This simplicity can be achieved by using common orders of precedence (unlike APL), using popular constructs from design work (logic symbols and related notations) and using only a reasonable number of primitive data types and avoiding special case syntax. Designers will take the time to write out a description only if it is easy to

write descriptions in that language, making writability at least as important as readability.

Another consideration in selecting a language is its generality. A language should not be restricted to any single type of hardware design. More specifically, a description language should be capable of describing different machine organizations, timing mechanisms and actual hardware implementations. This requirement relates to simulation and hardware translation, too. For example, several translators have been developed to translate a hardware description to a NAND-NAND implementation table. This type of system is not as flexible as one which allows a user to define the type of implementation components (i.e., LCD [EG76]) or some future translator that may allow translations to the IC package level.

The final practical consideration is to check that a language simulator exists and is capable of being implemented on the planned host computer. A hardware description language must be combined with a simulator (and a translator) to become a truly powerful design tool. If a simulator does exist, it is of no use to a designer if it can not be used on the type of computer the designer has available for implementation. Studies of the various simulators also need to be performed to determine if the simulation runs are financially feasible. An example of such a study is found in [HC77].

These "practical" considerations are primarily matters of individual design situations (e.g., read, write, generality, ease of use). They are important for making sure an appropriate technical choice is also an appropriate practical choice. These considerations

should be thought of as a fine tuning tool, useful for developing an order of preference among the pool of CHDLs arrived at through technical considerations.

In the selection of an individual HLL or GPSL the technical aspects of the languages need not be dealt with in such fine detail since as a class these languages are relatively similar in these areas. The same practical considerations should again be used to make the final selection of an individual language that is appropriate for use in a given design project.

Selection Example (Part C.)

Two CHDLs have been selected as suitable for design use based on the technical considerations of Section 3. Now practical issues should be considered to develop an order of precedence for the remaining languages.

Both CDL and DDL have been implemented in design systems; CDL in an academic environment [Me68a,Me68b] and DDL in both academic [CD79,Di70] and industrial [Sh79b] settings. However, DDL has received more attention in recent years then has CDL. The two languages are both very readable. DDL is somewhat harder to read and write than CDL as it has a multiple block structure which partially accounts for the extra difficulty. CDL and DDL are both very general and have no dependency on any particular implementation. The CDL translator is written in FORTRAN while the DDL translator has both a FORTRAN and PASCAL version. Simulators for both languages have been developed and improved several times [HC77,KS79].

For the purposes of this example DDL is the preferred language

because of its increased popularity in recent years, its increased notational flexibility due to a multi-block structure and its simulator/translator which has been implemented in two high-level programming languages.

5. SUMMARY

An approach for selecting a language suitable for computer hardware description and simulation was presented. This approach involves selecting one class of Hardware Description Language (HDL) as being best suited for meeting design requirements and then choosing an individual language from that class. The selection of an individual HDL is made based on a set of technical and a set of practical considerations. Technical considerations are used to insure that a prospective language is capable of performing the design tasks for which it will be implemented. Practical considerations are used to aid in selecting a language that can be implemented simply and effectively. The use of the selection approach allows an organized and efficient examination of the languages available for hardware design.

As Computer Hardware Description Languages (CHDLs) were found to be the most suitable for describing hardware, they have been emphasized more than the other two classes of HDLs. A review of CHDLs was presented to acquaint the reader with these languages so that they could be examined on an equal level with the High-Level Languages (HLLs) and the General Purpose Simulation Languages. Included in this review were discussions of "early" CHDLs, a general form of CHDL syntax and semantics and detailed reviews of two example CHDLs (CDL and DDL).

The range of formal languages available for use as hardware description and simulation tools was then discussed. Each of

the three classes of HDLs (HLLs, GPSLs and CHDLs) was presented on the basis of the advantages and disadvantages of using each particular class. Examples of hardware descriptions written in specific languages from each of the three classes of HDLs were also given. CHDLs have again been emphasized by including more example CHDLs and the introduction of a four-dimensional CHDL-space useful for the characterization of languages from this class of HDL.

Finally, the specific details of the HDL selection approach were discussed. This discussion included reviewing the advantages and disadvantages of an HDL-based design system; discussing considerations for selecting a single class of HDL; and then presenting the sets of technical considerations and practical considerations. The selection of an HDL for use in an academic environment was used as a "continuing" example to aid in the presentation of the HDL selection approach.

BIBLIOGRAPHY

[AF79] Anlauff, H., Funk, P. and Meinen, P., "PHPL - A New Computer Hardware Description Language for Modular Descriptions of Logic and Timing," Proc. 4th Intl. Symposium on CHDLs, Oct. 1979.

[AD70] Arndt, R.L. and Dietmeyer, D.L., "DDLSIM - A Digital Design Language Simulator," Proc. of NEC, Vol. 26, pp. 116-118, Dec. 1970.

[BB75] Bara, J. and Born, R., "A CDL Compiler for Designing and Simulating Digital Systems at the Register Transfer Level," Proc. Intl. Conference CHDL's Applications, pp. 69-75, 1975.

[BS71] Baray, M.B. and Su, S.V.H., "A Digital System Modeling and Design Language," Proc. 8th Annual Design Automation Workshop, pp. 1-22, 1971.

[BB78] Barbacci, M.R., Barnes, G.E., Catteu, R.G., and Siewiorek, D.P., "The ISPS Computer Description Language," Dept. of Computer Science, Carnegie-Mellon Univ., March 1978.

[BB72] Barbacci, M.R., Bell, C.G., and Newell, A., "ISP: A Language to Describe Instruction Sets and Other Register Transfer Systems," COMPCON 72, pp. 227-230, 1972.

[BS75] Barbacci, M.R. and Siewiorek, D.P., "Applications of an ISP Compiler in a Design Automation Laboratory," Proc. Intl. Symp. CHDL's Applications, pp. 69-75, 1975.

[BS77] Barbacci, M.R. and Siewiorek, D.P., "Evaluation of the CFA Test Programs via Formal Computer Descriptions," Computer, Vol. 10, No. 10, Oct. 1977.

[Ba75] Barbacci, M.R., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," IEEE Trans. Computers, Vol. C-24, No. 2, pp. 137-150, Feb. 1975.

[Ba76] Barbacci, M.R., "The ISPL Compiler and Simulator User's Manual," Technical Report, Dept. Computer Sci., Carnegie-Mellon Univ., 1976.

[Ba79] Barbacci, M.R., "Instruction Set Processor Specifications for Simulation, Evaluation and Synthesis," Proc. 16th Annual Design Automation Conf., pp. 64-72, June 1979.

[Ba80] Barbacci, M.R., "The Symbolic Manipulation of a Computer Description: An ISPS Simulator," Technical Report, Dept. Computer Sci., Carnegie-Mellon Univ., 1980.

[Ba81a] Barbacci, M.R., "Instruction Set Processor Specification (ISPS): The Notation and Its Applications," IEEE Trans. Computers, Vol. C-30, No. 1, pp. 24-40, Jan. 1981.

[Ba81b] Barbacci, M.R., "Syntax and Semantics of CHDLs," Fifth Intl. Conf. on CHDLs and Their Applications, pp. 305-311, 1981.

[BL62] Bartee, T.C., Lebow, I.L. and Reed, I.S., Theory and Design of Digital Systems, McGraw-Hill Book Co., Inc., New York, N.Y., 1962.

[BK72] Bell, C.G., Knudsen, M. and Sieworek, D., "PMS - A Notation to Describe Computer Structures," COMPCON72, pp. 227-230, 1972.

[BM78] Bell, C.G., Mudge, J.C. and McNamara, J.E., Computer Engineering: A DEC View of Hardware Systems Design, Digital Press, 1978.

[BN70] Bell, C.G. and Newell, A., "The PMS and ISP Description Systems for Computer Structures," Proc. Spring Joint Computer Conf., pp. 351-374, 1970.

[BN71] Bell, C.G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill Book Co., Inc., New York, N.Y., 1971.

[BD73] Birtwistle, G.M., Dahl, O.J., Myhrhaug, B. and Nygard, K., SIMULA BEGIN, Auerbach Publishing, Inc., 1973.

[Bl71] Bliss, F.W., "Computer-Aided Logic Design," Ph.D. Thesis, Case Western Reserve Univ., 1971.

[Bl72] Bliss, F.W., "Computer-Aided Logic Design," Proc. 9th Annual Design Automation Workshop, pp. 259-263, 1972.

[Br66] Breuer, M.A., "General Survey of Design Automation of Digital Computers," Proc. IEEE, Vol. 54, No. 12, pp. 1708-1721, Dec. 1966.

[Br72] Breuer, M.A., "Recent Developments in the Automated Design and Analysis of Digital Systems," Proc. IEEE, Vol. 60, No. 1, pp. 12-27, Jan. 1972.

[Ca83] Calderbank, V.J., FORTRAN, Chapman and Hall Ltd., London, 1983.

[CT77] Ching, S.S. and Tracey, J.H., "An Interactive Computer Graphics Language for the Design and Simulation of Digital Systems," Computer, Vol. 10, No. 6, pp. 35-41, June 1977.

[Ch76a] Ching, S.S., "EXFLOW - An Interactive Graphics Package for the Design and Simulation of Digital Networks," Ph.D. Dissertation, Univ. Missouri-Rolla, 1976.

[Ch76b] Ching, S.S., "EXFLOW User's Manual," Technical Report, Univ. Missouri-Rolla, 1976.

[Ch76c] Ching, S.S., "EXFLOW Implementation Package," Technical Report, Univ. Missouri-Rolla, 1976.

[Ch62] Chu, Y., Digital Computer Design Fundamentals, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1962.

[Ch65] Chu, Y., "An ALGOL-like Computer Design Language," Comm. ACM, Vol. 8, No. 10, pp. 607-615, Oct. 1965.

[Ch69] Chu, Y., "Design Automation by the Computer Design Language," Technical Report 69-86, Dept. Computer Sci., Univ. Maryland, Mar. 1969.

[Ch70] Chu, Y., Introduction to Computer Organization, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1970.

[Ch72a] Chu, Y., Computer Organization and Programming, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

[Ch72b] Chu, Y., "Introducing the Computer Design Language," COMPCON72, pp. 218-222, 1972.

[Ch74a] Chu, Y., "Introducing CDL," Computer, Vol. 7, No. 12, pp. 31-33, Dec. 1974.

[Ch74b] Chu, Y., "Structure of CDL Programs," Technical Note 74-58, Dept. Computer Sci., Univ. Maryland, May 1974.

[CD79a] Cory, W., Duley, J. and vanCleemput, W., "DDL-P Language Manual," Technical Report, Computer Systems Lab., Stanford Univ., 1979.

[CD79b] Cory, W., Duley, J. and vanCleemput, W., "DDL-P Command Language Manual," Technical Report, Computer Systems Lab., Stanford Univ., 1979.

[Co81] Cory, W.E., "Symbolic Simulation for Functional Verification with ADLIB and SDL," Proc. 18th Annual Design Automation Conf., pp. 82-89, 1981.

[CT70] Crall, R.F. and Tracey, J.H., "IDDAP - Interactive Computer Assistance for Creative Digital Design," Proc. National Electronics Conf., pp. 93-98, 1970.

[Cr70] Crocket, E.D., et al., "Computer-Aided System Design," Proc. Fall Joint Computer Conf., pp. 287-296, 1970.

[DM70] Dahl, O., Myhrhaug, B. and Nygaard, K., "Simula Common Base Language," Publication S-22, Norwegian Computing Center, Oslo, Oct. 1970.

[Da68] Darringer, J.A., "A Language for the Description of Digital Computer Processors," Proc. 5th Annual Design Automation Workshop, pp. 15-18, 1968.

[Da69] Darringer, J.A., "The Description, Simulation and Automatic Implementation of Digital Computer Processors," Ph.D. Thesis, Carnegie-Mellon Univ., May 1969.

[Da81] Dasgupta, S., "S_A*: A Language for Describing Computer Architectures," Fifth Intl. Conf. CHDLs and Their Applications, pp. 65-78, 1981.

[De70] Dennis, J.B., "Modular, Asynchronous Control Structures for a High Performance Processor," Conf. Rec. Project MAC, Conf. Concurrent Systems and Parallel Computations, ACM, New York, pp. 55-80, 1970.

[Di70a] Dietmeyer, D.L., "DDLTRN - User's Manual," Dept. Electrical and Computer Engr., Univ. Wisconsin-Madison, 1970.

[Di70b] Dietmeyer, D.L., "DDLSIM - User's Manual," Dept. Electrical and Computer Engr., Univ. Wisconsin-Madison, 1970.

[Di71] Dietmeyer, D.L., Logic Design of Digital Systems, Allyn and Bacon, Inc., Boston, 1971.

[Di74] Dietmeyer, D.L., "Introducing DDL," Computer, Vol. 7, No. 12, pp. 34-38, Dec. 1974.

[DR58] Dinnen, G.P., Reed, I.S. and Lebow, I.L., "The Logical Design of CG24," Proc. Eastern Joint Computer Conf., pp. 91-94, 1958.

[Du67] Duley, J.R., "DDL - A Digital System Design Language," Ph.D. Dissertation, Univ. Wisconsin-Madison, 1967.

[DD68] Duley, J.R. and Dietmeyer, D.L., "A Digital System Design Language (DDL)," IEEE Trans. Computers, Vol. C-17, No. 9, pp. 850-861, Sept. 1968.

[DD69] Duley, J.R. and Dietmeyer, D.L., "Translation of A DDL Digital System Specification to Boolean Equations," IEEE Trans. Computers, Vol. C-18, No. 4, pp. 305-313, Apr. 1969.

[EG76] Evangelisti, C.J., Goertzel, G. and Ofek, H., "LCD - Language for Computer Design (Revised), "IBM Research Report RC-6244, Yorktown Heights, N.Y., 32 pp., Oct. 1976.

[EG77] Evangelisti, C.J., Goertzel, G. and Ofek, H., "Designing with LCD-Language for Computer Design," Proc. 14th Annual Design Automation Conf., pp. 369-376, 1977.

[FM81] Flake, P.L., Moorby, P.R. and Musgrave, G., "HILO MARK2 - Hardware Description Language," Fifth Intl. Conf. CHDLs and Their Applications, pp. 95-101, 1981.

[Fr67] Franke, E.A., "Automated Functional Design of Digital Systems," Ph.D. Thesis, Case Western Reserve Univ., Nov. 1967.

[FM68] Franke, E.A. and Mergler, H.W., "Computer-Aided Functional Design of Digital Systems," SEIEEE Record, pp. 13c1-13c4, Apr. 1968.

[Ge71] Gentry, M., "A Compiler for AHPL Control Sequences, Ph.D. Dissertation, Univ. Arizona, June 1971.

[GA62] Gorman, D.F. and Anderson, J.P., "A Logic Design Translator," Proc. Fall Joint Computer Conf., pp. 251-261, 1962.

[HK82] Hancock, L. and Krieger, M., The C Primer, McGraww-Hill, Inc., New York, 1982.

[HM60] Hannig, W.A. and Mayers, T.L., "Impact of Automation on Digital Computer Design," Proc. Eastern Joint Computer Conf., pp. 211-232, 1960.

[Ha77] Hartenstein, R.W., Fundamentals of Structured Hardware Design, North-Holland Pub. Co., Amsterdam, 1977.

[Hv79] Hartenstein, R.W. and vonPuttkamer, E., "KARL - A Hardware Descriptive Language as part of a CAD Tool for LSI," Proc. 4th Symp. on CHDLs, pp. 151-161, Oct. 1979.

[HC77] Heath, J.R., Carroll, B.D. and Cwik, T.T., "CDL - A Tool for Hardware and Software Development," Proc. 14th Annual Design Automation Conf., pp. 445-449, 1977.

[HH75] Hemming, C.W. and Hemphill, J.M., "Digital Logic Simulation Models and Evolving Technology," Proc. 12th Annual Design Automation Conf., pp. 85-91, 1975.

[He58] Hesse, V.L., "The Advantages of Logical Equation Techniques in Designing Digital Computers," Proc. Winter Joint Computer Conf., pp. 186-188, 1958.

[Hv79b] Hill, D.D. and vanCleemput, W., "SABLE: A Tool for Generating Structured, Multi-Level Simulations," Proc. 16th Annual Design Automation Conf., pp. 272-279, 1979.

[Hi79a] Hill, D.D., "ADLIB: A Modular, Strongly-Typed Computer Design Language," Proc. 4th Symp. on CHDLs, pp. 75-81, Oct. 1979.

[Hi79b] Hill, D.D., "ADLIB-SABLE User's Guide," Technical Report, Computer Systems Lab., Stanford Univ., 1979.

[Hi80] Hill, D.D., "Language and Environment for Multi-Level Simulation," Ph.D. Thesis, Dept. Electrical Engr., 170 pp., March 1980.

[HP73] Hill, F.J. and Peterson, G.R., Digital Systems: Hardware Organization and Design, Wiley and Sons, New York, 1973 (second ed. 1978).

[Hi74] Hill, F.J., "Introducing AHPL," Computer, Vol. 7, No. 12, pp. 28-30, Dec. 1974.

[Hi75] Hill, F.J., "Updating AHPL," Proc. Intl. Symp. CHDLs and Their Applications, pp. 22-29, 1975.

[Iv62a] Iverson, K.E., A Programming Language, Wiley and Sons, Inc., New York, 1962.

[Iv62b] Iverson, K.E., "A Common Language for Hardware, Software and Applications," Proc. Fall Joint Computer Conf., pp. 121-129, 1962.

[JS77] Jordan, H.F. and Smith, B.J., "The Assignment Statements in HDLs," Computer, Vol. 10, No. 6, pp. 43-49, June 1977.

[JS79] Jordan, H.F. and Smith, B.J., "Structure of Digital System Description Languages," Proc. 16th Annual Design Automation Conf., pp. 31-34, 1979.

[KS79] Kawato, N., Saito, T., Maruyama, F. and Vehara, T., "Design and Verification of Large-Scale Computers by Using DDL," Proc. 16th Annual Design Automation Conf., pp. 360-366, 1979.

[KV69] Kiviat, P.J., Villanueva, R. and Markowitz, H., The Simscript II Programming Language, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1969.

[Ki81] Kiwiat, P.J., "Introduction to the Simscript II Programming Language," Proc. Winter Simulation Conf., pp. 32-36, 1981.

[Kn73] Knudsen, M.J., "PMSL: An Interactive Language for System-Level Description and Analysis of Computer Structures," Ph.D. Thesis, Dept. Computer Sci., Carnegie-Mellon Univ., Apr. 1973.

[Le78] LePage, W.R., Applied APL Programming, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[Li77] Lipovski, G.J. "Hardware Description Languages: Voices from the Tower of Babel," Computer, Vol. 10, No. 6, pp. 14-17, June 1977.

[Lu73] Lund, J., "LOGAL - Logic Algorithmic Language," Univac Technical Memo A00317, Roseville, Minn., Mar. 1973.

[Ma80] Maryanski, F., Digital Computer Simulation, Hayden Book Co., Inc., Rochelle Park, N.J., 1980.

[MO79] Maxey, G.E. and Organick, E.I., "CASL - A Language for Automating the Implementation of Computer Architecture," Proc. 4th Intl. Symp. CHDLs, 1979.

[Me68a] Mesztenyi, C.K., "Translator and Simulator for the Computer Design and Simulation Program, Version 1, "Technical Report 67-48, Computer Sci. Center, Univ. Maryland, June 1968.

[Me68b] Mesztenyi, C.K., "Computer Design Language: Simulation and Boolean Translation," Technical Report 68-72, Computer Sci. Center, Univ. Maryland, June 1968.

[Na60] Naur, P., Ed., "Report on the Algorithmic Language ALGOL-60," Comm. ACM3, pp. 299-314, May 1960.

[NS78] Navabi, Z., Swanson, R.E. and Hill, F.J., "User Manual for AHPL Simulator (HPSIM/AHPL Compiler (HPCOM))," Engineering Experiment Sta., Univ. Arizona, June 1978.

[NH79] Navabi, Z. and Hill, F.J., "Efficient Simulation of AHPL," Proc. 16th Annual Design Automation Conf., pp. 255-262, 1979.

[Ni80] Nickerson, R.C., Fundamentals of Fortran Programming, Winthrop Publ., Inc., Cambridge, MA., 1980.

[No80] Northcutt, J.D., "The Design and Implementation of Fault Insertion Capabilities for ISPS," Proc. 17th Annual Design Automation Conf., pp. 197-209, 1980.

[Om73] Omohundro, W.E., "FLOWWARE - A Flow Charting Procedure to Describe Digital Networks," First Annual Symp. Computer Architecture, pp. 91-97, 1973.

[PW78] Parker, A. and Wallace, J., "The Development of GLIDE: A Hardware Description Language for Interface and I/O Port Specifications," Research Report, Dept. Electrical Engr., Carnegie-Mellon Univ., 1978.

[PA80] Parker, A. and Altman, A.H., "The SLIDE Simulation: A Facility for the Design and Analysis of Computer Interconnections," Proc. 17th Annual Design Automation Conf., pp. 148-155, 1980.

[PW81] Parker, A. and Wallace, J., "SLIDE: An I/O Hardware Descriptive Language," IEEE Trans. Computers, Vol. C-30, No. 6, pp. 423-436, June 1981.

[PD72] Patil, S.S. and Dennis, J.B., "The Description and Realization of Digital Systems," COMPCON72, pp. 223-226, 1972.

[Pa76a] Paulman, J., Fundamentals of APL Programming, Wm. C. Brown Pub. Co., 1976.

[Pa76b] Pawlak, A., "FDL - A Language for Digital Systems Modeling on a Functional Level," Third Annual Symp. Computer Architecture, 1976.

[PJ81] Pawlak, A. and Jezewski, J., "MODLAN - A Language for Multi-Level Description and Modeling of Digital Systems," Fifth Intl. Conf. CHDLs and Their Applications, pp. 79-93, 1981.

[Pi80a] Piloty, R., et al., "CONLAN - A Formal Construction Method for Hardware Description Languages: Basic Principles," IFIPS National Computer Conf., pp. 209-217, 1980.

[Pi80b] Piloty, R., et al., "CONLAN - A Formal Construction Method for Hardware Description Languages: Language Derivation," IFIPS National Computer Conf., pp. 219-226, 1980.

[Pi80c] Piloty, R., et al., "CONLAN - A Formal Construction Method for Hardware Description Languages: Language Application," IFIPS National Computer Conf., pp. 229-236, 1980.

[PB82] Piloty, R. and Barrione, D., "The CONLAN Project: Status and Future Plans," Proc. 19th Annual Design Automation Conf., pp. 202-212, 1982.

[Pr64] Proctor, R.M., "A Logic Design Translator Experiment Demonstrating Relationships of Language to Systems and Logic Design," IEEE Trans. Electronic Computers, Vol. EC-13, No. 8, pp. 422-430, Aug. 1964.

[Re52] Reed, I.S., "Symbolic Synthesis of Digital Computers," Proc. ACM, pp. 90-94, Sept. 1952.

[Re53] Reed, I.S., "The Symbolic Design of Digital Computers," Technical Memo, No. 23, MIT Lincoln Lab., Lexington, Mass., 1953.

[Re57] Reed, I.S., "Symbolic Design Techniques Applied to a Generalized Computer," Technical Report No. 141, MIT Lincoln Lab., Lexington, Mass., 1957.

[Ro78] Robinson, I., "The Design and Simulation of a Parallel Processor using HPSIM," M.S. Thesis, Univ. Arizona, Nov. 1978.

[RD83] Robinson, P. and Dion, J., "Programming Languages for Hardware Description," Proc. 20th Annual Design Automation Conf., pp. 12-16, 1983.

[Sc64a] Schlaeppi, H.P., "A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS)," IEEE Trans. Electronic Computers, Vol. EC-13, No. 8, pp. 439-448, Aug. 1964.

[Sc64b] Schorr, H., "Computer-Aided Digital System Design and Analysis using a Register Transfer Language," IEEE Trans. Electronic Computers, Vol. EC-13, No. 12, pp. 730-737, Dec. 1964.

[Sc82] Schriber, T.J., "Introduction to GPSS," Proc. 1982 Winter Simulation Conf., pp. 657-659, 1982.

[Sc79] Schuler, D.M., "A Language for Modeling the Functional and Timing Characteristics of Complex Digital Components for Logic Simulation," Proc. 4th Intl. Symp. CHDLs, 1979.

[Sh79a] Shiva, S.G., "Computer Hardware Description Languages - A Tutorial," Proc. IEEE, Vol. 67, No. 12, pp. 1605-1615, Dec. 1979.

[Sh79b] Shiva, S.G., "Digital Systems Design Language - Design Synthesis of Digital Systems," NASA CR-162032, Marshall Space Flight Center, Al., Oct. 1979.

[Sh80] Shiva, S.G., "Combinational Logic Synthesis From An HDL Description," Proc. 17th Annual Design Automation Conf., pp. 550-555, 1980.

[Sh83] Shiva, S.G., "Automatic Hardware Synthesis," Proc. IEEE, Vol. 71, No. 1, pp. 76-87, Jan. 1983.

[Si74] Siewiorek, D., "Introducing ISP," Computer, Vol. 7, No. 12, pp. 39-40, Dec. 1974.

[SB82] Siewiorek, D., Bell, C.G. and Newell, A., Computer Structures: Principles and Examples, McGraw-Hill, Inc., New York, 1982.

[ST81] Singh, A.K. and Tracey, J.H., "Development of Comaprison Features for Computer Hardware Description Languages," Fifth Intl. Conf. CHDLs and Their Applications, pp. 247-263, 1981.

[Si81] Singh, A.K., "Descriptive Techniques for Digital Systems Containing Complex Hardware Components," Ph.D. Dissertation, Kansas State Univ., Mar. 1981.

[St70] Stabler, E.P., "System Description Languages," IEEE Trans. Computers, Vol. C-19, No. 12, pp. 1160-1173, Dec. 1970.

[St77] Stewart, J.H., "LOGAL: A CHDL for Logic Design and Synthesis of Computers," Computer, Vol. 10, No. 6, pp. 18-26, June 1977.

[SB75] Su, S.Y.H. and Baray, M.B., LALSD - A Language for Automated Logic System Design," Proc. of Intl. Computer Symp., Taipei, Taiwan, pp. 31-42, Aug. 1975.

[SH81] Su, S.Y.H., Huang, C. and Fu, P.Y.K., "A New Multi-Level Hardware Design Language (LALSD II) and Translator," Fifth Intl. Confl CHDLs and Their Applications, pp. 155-169, 1981.

[SF81] Su, S.Y.H. and Fu, P.Y.K., "LALSD II Translator," Report No. 0281, Research Group on Design Automation and Fault-Tolerant Computing, State Univ. N.Y. at Binghamton, 1981.

[Su73a] Su, S.Y.H., "An Interactive Design Automation System," Proc. 10th Annual Design Automation Workshop, pp. 253-261, 1973.

[Su73b] Su, S.Y.H., "A Language for Automated Logic and System Design," Workshop on Computer Descriptive Languages, Rutgers Univ., New Brunswick, N.J., Sept. 1973.

[Su77] Su, S.Y.H., "Hardware Description Language Applications: An Introduction and Prognosis," Computer, Vol. 10, No. 6, pp. 10-13, June 1977.

[SN77] Swanson, R.E., Navabi, Z. and Hill, F.J., "An AHPL Compiler/Simulator System," Proc. Sixth Texas Conf. on Computing Systems, 1977.

[TS80] Tracey, J.H. and Singh, A.K., "Descriptive Techniques for Digital Hardware Descriptions," Workshop on Application of Machine Description Languages, Falmouth, MA., Sept. 1980.

[va77] vanCleemput, W.M., "A Structural Design Language for Computer Aided Design of Digital Systems," Technical Report No. 136, Computer Systems Lab., Stanford Univ., 27 pp., Apr. 1977.

[va79] vanCleemput, W.M., "Computer Hardware Description Languages and their Applications," Proc. 16th Annual Design Automation Conf., pp. 554-560, 1979.

[Wo77] Woolf, H.B., Ed., Webster's New Collegiate Dictionary, G. & C. Merriam Co., Springfield, Mass., 1977.

[Zi79] Zimmermann, G., "The MIMOLA Design System - A Computer Aided Digital Processor Design Method," Proc. 16th Annual Design Automation Conf., pp. 53-58, 1979.