NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

ALGORITHMIC HARDWARE DESIGN LANGUAGE (AHDL)

Allan Kang Ying Wong

A Thesis

in

The Faculty

of

Engineering

Presented in Partial Fulfillment of the Requirements
for the degree of Master of Computer Science
Department of Computer Science
Concordia University
Montreal, Quebec, Canada

September 1978

ABSTRACT

ALGORITHMIC HARDWARE DESIGN LANGUAGE (AHDL)

Allan Kang Ying Wong

A textual consensus multi-level language, intended specifically for the description, simulation, and design of computing hardware systems, is presented in this thesis. The language is AHDL (Algorithmic Hardware Design Language). Its structure contains a fluid hierarchy of three levels : behavioral level, functional level, and structural level. These levels can be either applied individually for their designated purposes, or interacted in accordance with the rules described in the text. The language AHDL is capable of expressing concurrent and parallel operations, which are the fundamental properties of hardware systems. It also accommodates the application of standard logic design techniques and available IC technologies.

The language AHDL is described on a theoretical level. This research was concentrated mainly on the validity of the language design. The implementation of AHDL was outside the scope of the work concerned. The structure of AHDL was, however, thoroughly tested on many different examples. Some of these examples are provided in the appendix, while others are presented in appropriate places throughout the body of this thesis.

PREFACE

The establishment of a " universal " language applicable to the simulation, description, and design of computing hardware systems, has recently been the subject of different researches. Such a language would require a consensus multi-level modular structure. It would support the formulation of algorithms at any desired level of detail. The aim of this project is to design a textual linguistic structure, which fulfills the requirements of the language concept stated above.

Hardware design and description languages, principally Register Transfer (RT) languages, were first designed early in the sixties. Since that time, they have been refined, and new ones have been developed, in attempts to keep pace with the evolution of IC technology. The RT languages presently in existence together cover the continuum of digital applications. Each RT language, however, has certain limitations on its application because of the framework of its structure. In general, a RT language is applicable only to a particular level of detail.

AHDL (Algorithmic Hardware Design Language) developed in this research is a textual consensus multi-level modular language. It is unique in that it potentially covers the complete range of contemporary digital design. It was derived by the technique of language tuning, which has been responsible for the emergence of most of the existing textual RT languages.

This thesis work was undertaken as a result of the ideas provided by Dr. Terill Fancott.

I wish to express my full gratitude to Dr. T. Fancott for his supervision and guidance during the research and development of the language AHDL.

TABLE OF CONTENTS

CHAPTER TWO : BEHAVIORAL LEVEL OF AHDL

CHAPTER THREE : FUNCTIONAL LEVEL OF AHDL

CHAPTER FOUR : STRUCTURAL LEVEL OF AHDL

CHAPTER ONE

GENERAL INTRODUCTION

RATIONALE OF DIGITAL HARDWARE SYSTEMS

As pointed out by Dietmeyer (Dietmeyer-01) and Peatman (Peatman-02), the hardware organisation of any digital system can be simply represented in the model illustrated by Figure 1-1. The actual implementation of any digital system, however, may take a variety of forms because of the two criteria : cost-effectiveness, and designated application (Fairchild-03). In Figure 1-1, the rational hardware organisation for any digital system would include two major networks interacting together. These two major networks are the control network and the data processing network. The data processing network transforms system inputs to meaningful outputs. All the operations in the data processing network are sequenced by the control network. The control network itself may be subdivided into the timing circuitry and the mode circuitry. Operations which are functions of time only are sequenced by the timing circuitry, and operations which are functions of both time and data are sequenced by the mode circuitry. The mode circuitry is a decision-making structure, and the decisions made are the functions of timing issues from the timing circuitry and data from the data processing network. Generally, the control network activates only the appropriate portion of the data processing network, for the specific data transformation.

The constituents of any digital system are discrete hardware components collectively known as structural primitives (Gardner-04). These basic building blocks may bear different levels of meaning and complexity. The lowest level has been generally accepted as gates and flip-flops at the SSI level. The higher levels may involve MSI and LSI.

Concurrency and parallelism are the fundamental properties of hardware systems. The harmonious co-operation of the structural primitives within a digital system must be ensured by synchronizing mechanisms. Hardware synchronization is achieved through enabling and disabling electrical signals. These binary signals are usually generated according to algorithms.

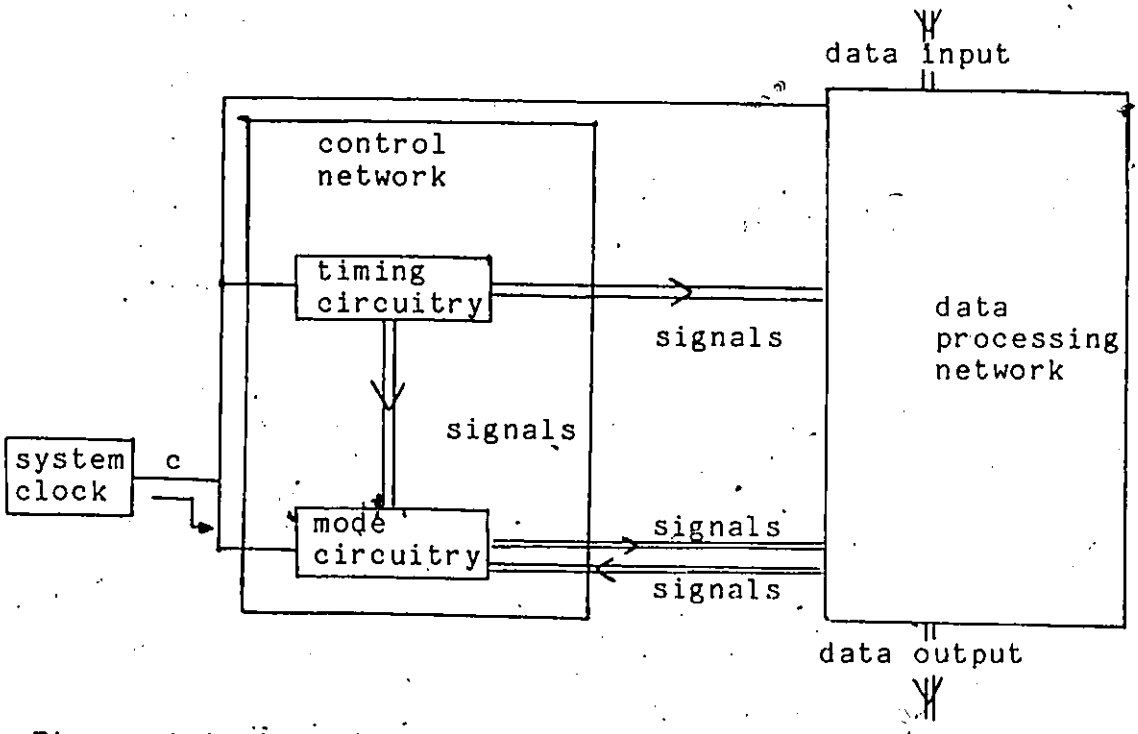


Figure 1-1 A rational hardware configuration for any digital system.

THE PHILOSOPHY OF DIGITAL-SYSTEM DESIGN AND DESCRIPTION

Digital system design is usually a top-down multi-level process of algorithmic formulations and the assembly of appropriate structural primitives for the execution of the algorithms formulated. Each level in a design process represents a stage in the design cycle to provide rational feedback to the designer. Similiar to the philosophy of a design process, description of a digital system may also be a multi-level process. The purpose of any descriptive process is to express the desired level of detail of a system, ranging from simple behavioral information to subtle physical details.

Multi-level description is a method of making a digital system understandable by documenting information on an appropriate set of levels. It allows the different people, who take part in the different areas of a system, to communicate among themselves. The tool for communication, however, is a language, powerful and universal enough, for expressing different levels of ideas, but using a common syntax.

A top-down multi-level design methodology is a basic engineering principle. If there is a language, which can accommodate the different stages within a top-down design cycle and support the expressions of various design techniques, available in a design process, the ease of the design will be greatly enhanced.

DIGITAL SYSTEM DESIGN OBJECTIVE AND TECHNIQUES

" The ultimate objective of any digital system design is to achieve a given performance for the lowest total system cost. " (Fairchild-03). In the past decade , system design, component selection and logic design were to a large degree independent of one another. Logic designers were concerned with designing with the minimum number of gates and flip-flops by using the conventional techniques such as Boolean algebra or Karnaugh Maps. The evolution of IC technologies has brought abundant choices of more cost-effective MSI and LSI logic components. The impact of these sophisticated logic components has changed radically the techniques of designing digital systems by making system design, logic design, and component selection heavily interdependant. To-day, designing by minimizing the number of gates and flip-flops is no longer sufficient. Modularization (Ellis-05, Franklin-06, Clark-07), which is the partition of a system into subsystems or even sub-subsystems called modules, is far more important. Each module, which is a bounded system itself and is functionally complete, then lends itself to design with increasingly sophisticated and cost-effective ICs. In some cases, a module is just a single IC logic component. A module is a bounded system because it cannot be affected by other systems except through the input interconnections. It is functionally complete because it has been isolated for designated performance to the specifications of a designer.

In order to cope with the rapid evolution of IC

technologies, a new design methodology for digital systems is required to include system design, logic design, and component selection as an integral part in any design process. In particular, such a design methodology must be hierarchical in order to do the following :

1. To formulate the algorithms for a system architecture at the schematic level.
2. To partition a system into modules so that each module can be individually and completely developed; then the control structure to make the modules interact harmoniously can be designed.
3. To provide a framework which can accommodate different IC technologies.
4. To accommodate conventional logic design techniques whenever needed.

Before a design technique can be applied at will to do the four things stated, it must contain enough basic elements which can be selectively assembled for any particular application during a design process.

MOTIVATION FOR A HIGH-LEVEL UNIVERSAL LANGUAGE

The motivation for a high-level universal language is to provide a versatile notation which can be used equally well as a descriptive , simulating and design tool, applicable to any wanted level of detail. The notation required is a formal mechanism to generate and transmit the right amount of

information needed. It must maintain proper and unambiguous communication among different phases and areas of system development.

The first criterion of a high-level universal language is to establish an orderly framework capable of accommodating the development of numerous and yet distinctive dialects for communications as pointed by Iverson and Chu :

Iverson (1-Iverson-08) :

> " ...., in the design of a data processing system, it is imperative to maintain close communication between the programmers (i.e. the ultimate users), the software designers, and the hardware designers.."

Chu (1-Chu-10) :

> " Software designers , practically system programmers, need to know how the hardware system works without knowing electronics. In fact those who are in selling, servcing , operating, teaching, fabricating, installing , and testing computers all need to know how a modern stored-program computer works rather than how the hardware is interconnected. "

The second criterion of a high-level universal language is to provide a design tool that keeps up with technology evolution, but not the specific application of any technology as stated by Barbacci and Siewiorek (Siewiorek-12) :

" Because of the rapid technology evolution, there is

a great need to attain independence from any specific technology in the design process. "

REQUIREMENTS OF A HIGH-LEVEL UNIVERSAL LANGUAGE

A high-level universal language must fulfill three basic requirements. First, it must be based on a multi-level hierarchical modular philosophy in order to be compatible with the hierarchical nature of digital systems. The hierarchical nature of digital systems was clearly explained by Barbacci (Barbacci-13), and the essence of a multi-level hierarchical modular philosophy was presented by Su (Su-15). Second, it must be consensus. As pointed out by Lipovski (Lipovski-18), a consensus hardware descriptive and design language contains a common set of linguistic elements applicable to any level of a digital design or descriptive process. These linguistic elements can then be assembled according to needs into the corresponding and appropriate constructs. Third, to permit a rapid top-down approach for any designated application, it must have a high degree of extensibility incorporated into its structure. The meaning and the advantages of extensibility in the structure of a language were well-explained by Schuman (Schuman-19) .

PROMISES OF A TEXTUAL LINGUISTIC STRUCTURE

A textual linguistic structure is more advantageous for a high-level universal language for four reasons. First, the descriptive power of conventional programming languages to

express algorithms in terms of language statements has implied that textual languages can better express subtleties and variations than can graphical languages through shapes and sizes of figures. Second, the control structures in the conventional programming languages have already benefitted from extensive work over the past two decades. The merits of these control structures can be extracted and incorporated into a high-level universal language. Third, extensibility can be easily incorporated into textual language statements, and extensibility is the basic requirement for the generation of a multi-level hierarchical language structure. Fourth, the existing compilation techniques for compiling conventional programming languages are potentially applicable for a high-level textual universal language.

IS THERE ANY TEXTUAL HIGH-LEVEL UNIVERSAL LANGUAGE ?

1974, Su (Su-17) discussed different textual hardware description languages in the field. He pointed out that these languages, intended for the design and description of digital systems, were developed on the basis of a hierarchical top-down methodology. Therefore the developments of these languages were basically directed towards the concept of a textual high-level universal language. These languages are RT (Register Transfer) languages because they are operating at the register level. They have covered all the needs in both the design and descriptive processes for digital systems. However, each RT language is applicable to only a particular level of practice because its

applicability is limited by the language structure.

Up to the present, there is no textual HDL (Hardware Description Language) that can fulfill all the requirements of a textual high-level universal language, undoubtedly because there are always reasonable limitations for the application of any existing textual HDL. These limitations were explored by Lipovski (Lipovski-18) and Jordan (Jordan-20). Similar information can also be found in Barbacci-14.

A TEXTUAL HIGH-LEVEL UNIVERSAL LANGUAGE TO DESIGN AUTOMATION

Design automation using computers as design aids is a new and important subject in digital design methodology . The context of design automation is to provide a computerized hierarchical top-down design process from simulation to synthesis for any proposed digital design. It aims at a rapid top-down iterative design scheme by allowing efficient optimization of the many possible design tradeoffs which are part of the process of system or logic design. The book by Breuer (1-Breuer-21) gives a clear concept of design automation, mentioning different fundamentals and requirements.

1972, design automation had already received a great deal of attention as illustrated by Breuer in his survey (2-Breuer-22). Since then, the number of active researchers in this area has been ever increasing, as clearly indicated by the proceedings presented at the 1975 International Symposium on

C.H.D.L. and Applications (Proceedings-23). Design automation systems of variable degree of success were reported , and the better known projects providing an overview of the subject are presented in Siewiorek-12, and Stewart-26. All these researchers adopted somewhat different techniques and approaches for the developments of their design automation systems, but the quality and the feasibility of the languages used were always their primary considerations. It was not unusual that existing textual RT languages were tuned to their needs through appropriate modifications . This underlines the fact that a more universal textual high-level language would be desirable in the industry of design automation.

In order to derive a comprehensive automated design, the quality and the feasibility of the language adopted is very crucial because of three reasons. First, the top-down hierarchical nature of any automated design needs a compatible hierarchical design language. Second, the language used is needed to convey accurately the design problem to the computer and vice versa. Third, the language used is required to support any design stage in the cycle of an automated design by accommodating the pertinent and necessary design techniques. Therefore, a textual high-level universal language based on a multi-level hierarchical modular philosophy is potentially applicable to design automation. Its textuality allows a clear communication between the computer and the designer in an automated design process. The issue is however, how much latitude the language has in its structure to accommodate the

necessary information and the variety of design techniques possibly engaged.

SURVEY ON TEXTUAL HDLs IN THE FIELD

Hardware description and design languages developed so far are RT languages (Jordan-20) because they always operate at the register level. However, no RT language can completely assume the service of a textual high-level universal language because any RT language is good for only a particular level of application. Since these RT languages have covered a wide range of applications, several of them appropriate to different levels of applications may be occasionally combined to fulfill the task of a textual high-level universal language. Somehow, the application of different RT languages to the different levels of a particular commitment may be hindered by the problem of ambiguity. This ambiguity is due to the use of a set of common symbols to express different concepts in the different languages.

Operations at the register level are fully reflected in the assignment statements generalised in the syntax :

condition ⟶ carrier ⟵ data operation expression

The second part of the assignment statement :

carrier ⟵ data operation expression

represents the source and destination of a data transfer. It can

be broken up to have the following meanings :

1. The left arrow (⟵) is the transmit or assign operator which transmits or assigns what is on the right side to the destination on the left side.
2. The " data operation expression " represents any data transformation, whereas only the result will be transmitted or assigned. In special cases, it represents only a value to be transmitted or assigned but no data transformation at all.
3. The " carrier " is a memory entity acting as a destination for the corresponding data transfer. As pointed by Barbacci (Barbacci-14), carriers can be registers, or terminals of a transient natures. The latter represents wires coming out of combinational networks.

The first part of the assignment statement :

condition ⟶

represents the control which invokes the corresponding data transfer. The right arrow (⟶) is the operator which transfers control at a particular time interval from what is on the left to what is on the right. The " condition " abstracts the control signal generated by a " test network " or a " monitor ". The existence of the condition implies that dynamic data transfer is a rule rather than an exception. A more detailed description of the characterization of the RT level based on different RT languages was made by Barbacci (Barbacci-14).

As a conclusion drawn from the survey on different RT languages : ISP (Bell-27), DDL (Dietmeyer-01), CDL (2-Chu-11), APL (2-Iverson-09) and the language by Vogel (Vogel-28); and from the suggestions by Barbacci (Barbacci-14) and Jordan (Jordan-20); any RT language is, or can be completely defined by five basic abstract data types : register, terminal, variable, declarator, and operator. They are listed and explained in Table 1-1. They are called abstract data types because they abstract either the structures or the behaviors of the physical building blocks, as suggested by Flon (Flon-29).

The power and scope of the different RT languages range from stating a problem at the schematic level to representing structural details of physical objects. For example, ISP is good for any schematic-level application, DDL is good for descriptions of larger modular transfer, and CDL is good for the gate and flip-flop level. Though there are indeed overlapped capabilities among the different RT languages, one language may not be as suitable as another for a particular level of application.

In fact, the applicability of a RT language is determined by the available abstract data types it contains, the transformations which can be applied to these abstract data types, and the characteristics of the language structure. In ISP, the absence of the " terminal " abstract data type makes explicit terminal wiring description impossible. The control labels in CDL specify timing to an accuracy of one clock pulse

making CDL applicable for description at the structural level. The explicit dynamic multiplexed terminal assignment statements in DDL with the syntax :

<ST> condition : terminal identifier = expression

make the language more suitable for modular descriptions.

The procedural RT languages usually impose more rigid formats for their applications because the execution of statements is based on their sequential ordering. The execution of a statement is basically conditioned by the completion of the preceding one. In addition, the consideration of timing in procedural RT languages is generally crude. Therefore the structures and the timing considerations in the procedural RT languages can rarely cope with the subtleties and the precision needed respectively in the gate and flip-flop level. The non-procedural RT languages impose less rigidity in formatting because lexigraphical ordering of statements has no meaning. The execution of statements is based on a scheme of " dynamic selection " for a statement to be executed, and timing consideration is highly refined or even in terms of single descrete clock pulses. Therefore the non-procedural RT languages are more suitable for the level of gates and flip-flops. Barbacci (Barbacci-14) and Jordan (Jordan-20) compared the structures of different RT languages. They concluded that the limitations of these RT language were due to their particular linguistic structures.

Recently, moderate effort has been put into different research work, driving towards the development of a textual high-level universal language. Well-known researchers in terms of their efforts and ideas are Lipovski (Lipovski-18) and Su (Su-16). Lipovski is now working on the development of a framework which can accommodate the different powerful RT languages in a consensus way. Thus, when several RT languages of different levels of applications are combined together, they can fulfill all the requirements of a textual high-level universal language. Su developed LALSD which has been claimed to be a language for automated logic and system design. LALSD has a multi-level hierarchical structure and reasonable MACRO facilities, and it is potentially capable of describing, documenting, simulating, and synthesizing digital systems. The applicability of the language LALSD is, however, still at the experimental stage.

Table 1-1 Abstract data types of RT languages

abstract data type	declarator
structure	names or labels made up of natural language elements
function	identify the existence of different structures or declare the properties of different structures
extension	declarators are generally freely choosable but rules may exist to determine the ways to choose
special remark	a system of nomenclature
abstract data type	register
structure	one-dimensional array of binary nature
function	store information over a period of time
extension	forming two dimensional arrays (memory) or other structures by means of indexing
special remark	registers are identified by declarations and they are candidates for the destinations of data transfer

---

abstract data type : variable
structure : identifiers made up of a single character or several characters
function : variables have no inherent meaning, but they can take up either Boolean values or numerical values
extension : subscripted or non-subscripted
special remark : basic units of control formulations or computations; operated on by the appropriate operators; no physical correspondence

---

abstract data type : terminal
structure : identifiers made up of a single character or several characters
function : each terminal represents the output of a " circuit node " by taking on the current logical value of the node over a period of time or in a transient manner
extension : terminals can be dimensioned
special remark : terminals are identified by declarations

---

abstract data type : operator
structure : symbols
function : data transformations, and control formulations
extension : forming expressions for data transformations and control structures
special remark : choice based on convention

THIS THESIS

This thesis proposes a linguistic structure which theoretically encompasses all the essential features of a textual high-level universal language. The language which has been developed is called AHDL (Algorithmic Hardware Design Language). It is based on a consensus multi-level modular philosophy. The objective of the research work was concentrated mainly on the validity of the language proposal. Implementation of AHDL was not within the scope of the research.

The technique used for the development of AHDL is " language tuning " which is the technique responsible for the emergence of most of the existing RT languages. The basic principle of language tuning was implicitly justified by Barbacci (Barbacci-13) :

" In fact, conventional programming languages have been used. The issue is, however, how much they must be changed to reflect parallelism, timing and the structure of the object being represented. "

The power of language tuning was explicitly justified by Lipovski (Lipovski-18) saying :

" Specifically, what is more commonly referred to as hardware description language is a variation of a programming language tuned to the overall needs of describing hardware. "

Overall, the tuning process derives the syntax and semantics of a RT language from the merits of conventional programming languages with the additional inclusion of linguistic elements able to express parallelism and concurrency. The rule for establishing these elements for AHDL is abstractive power and familiarity .

RELATIONSHIP TO OTHER WORK

Conventional procedure-oriented programming languages developed for serial computations are powerful and well

developed, but they all have serious limitations for solving problems involving parallelism, Opler (Opler-65). There is, however, a high degree of correspondence between software programming techniques and hardware operations. It is clearly indicated by the philosophy and techniques of micro-programming (2-Chu-11). In fact, some conventional programming languages have tuned by different researchers to describe hardware which has the inherent property of parallel and concurrent operations. The tuning process has produced different powerful hardware descriptive and design RT languages like CDL, ISP and many others. It follows that language tuning, as learned from the past experience, is an efficient technique for the work of this project.

The intended applications and structures of textual RT languages have been continually evolving. Informal work on textual RT languages was started early in the sixties by Bartee (Bartee-31), and Schorr (Schorr-32). Then came the more formal and powerful textual RT languages like CDL, ISP, DDL, among others. Since the backgrounds and the approaches for the development of the different textual RT languages in the field were different , the applicability of these RT languages vary. They cover a cintinuum of levels of description and design of digital systems.

Nowadays, the complexity of digital systems, the rapid evolution of IC technologies, and the trend of design automation necessitate a textual high-level universal language to

communicate with, to simulate with, to design with, and to drive comprehensive automated designs. However, no existing textual RT language can individually and sufficiently support the goals stated. As a result, different researchers are looking for a language which possesses a linguistic structure functionally compatible to the " genuineness " of a textual high-level universal language. This is also the main objective of the research for the development of AHDL.

INTRODUCTION TO AHDL

AHDL, introduced by this thesis, is a consensus multi-level modular hardware design and descriptive language. It is capable of accommodating the three levels of detail of any digital system : the top or the behavioral level, the middle or the functional level, and the bottom or the structural level. The levels reflected in the structure of AHDL can be interacted, i.e. in a construct written in any given level, the features of the other two levels are available according to a set of rules described in the body of this thesis. A high degree of extensibility was incorporated into AHDL to ensure the smooth extension from the behavioral level through the functional level to the structural level.

The behavioral level, which concerns algorithmic formulation at the schematic level, is characterised by the register transfers, the procedural structures, and the basic set of operators. The structure of the behavioral level is basically

defined by " time blocks ". The generalised representation of a time block is

condition ⟶ execution of a " single action " or
a " block of single actions "

The structure of the behavioral level is procedurally similar to conventional programming languages, except for the possible inclusion of concurrent single actions in the block of single actions of the time blocks. Any single action may represent a basic unit of data operation, or a time block. The " condition " represents an active flow of control at a particular time interval.

The functional level represents the principles of the mechanisms contributing to the system dynamics. It is characterised structurally by Boolean equations and terminal variables. This level stresses the use of control variables as terminal functions. Modularization at the functional level becomes explicit. Modules may be specifically designed on this level in terms of input-output relationships.

The structural level considers timing in terms of single clock pulses. It is basically a level of logic design in terms of gates and flip-flops. This level is characterised by discrete modular descriptions and the interconnection of modules through input and output terminals.

The extensibility of AHDL partially comes from the

hierarchical nature of the set of operators it possesses. The operators situated at a higher level can always be defined by operators at the lower level so as to provide more detailed information.

Since the procedural structure of the behavioral level can not accommodate all the detailed information at the structural level, a non-procedural structure, which imposes less restrictions on the expression of digital functions, was adopted for this level. However, the time block characteristic prevails in the structures of the three levels, except that any time block in the structural level contains solely concurrent single actions in its block of single actions. Generally speaking, the procedural structure at the behavioral level fades into the non-procedural structure at the structural level through the compatibility of the time block construct to both structures.

The definition of a time block and a single action in its block of single actions can be recursive, because such a single action may be another time block itself. The procedural structure of the behavioral level and the non-procedural structure of the structural level defined in terms of time blocks are presented in the following :

1. The procedural structure outlined in BNF (Backus Naur Form) for the behavioral level is

<single action> ::= <simple action > | < time block>

<time block> ::= < condition> <execution>

<execution> ::= <single action> | <execution> <separator> <single action>

<condition> ::= <state of control> <→>

<separator> ::= , | ;

" , denotes concurrent behavior "

" ; denotes sequential behavior "

The most important feature of the procedural structure is the <execution> . It signifies that sequential ordering of single actions carrys a significant meaning, because the execution of a single action may be conditioned by the completion of the preceding one.

2. The non-procedural structure outlined in BNF for the structural level is

<single action> ::= < simple action > | < time block>

<time block> ::= <condition> <execution>

<execution> ::= <single action> | <execution> , <single action>

<condition> ::= <state of control> < >

In order to make the characteristics and the applicability of each level in AHDL more understandable, each level is compared in Table 1-2 with a well-known RT language which has the compatible level of behaviors and application.

Table 1-2 Comparison of each level of AHDL with a consistent RT language

LN :	CDL	DDL	ISP
RE :	2-Chu-11	Dietmeyer-01	Bell-27
ST :	non-procedural	non-procedural	procedural
AP :	gate and F-F level of application	transfer of data between automata or large modules	schematic-level formulating tool
CT :	in terms of single clock pulses	state of control memory medium	state of control memory medium
TB :	containing only concurrent activities	containing only concurrent activities	containing both sequential and concurrent activities
CM :	structural level of application	functional level but biased to the structural level	behavioral level of application

Remark : 1. The compatibility of the different RT languages with the respective levels in AHDL, as shown in the above comparisons, serves only as a guide-line to make the structure and applicability of AHDL more understandable. The actual differences between AHDL and the three languages are not conveyed.

2. The symbols used in this table are explained as in the following :

- LN = language
- RE = resource
- ST = structure
- AP = application
- F-F = flip-flop
- CT = control
- TB = time block
- CM = comparison with AHDL

CHAPTER TWO

BEHAVIORAL LEVEL OF AHDL

INTRODUCTION

The behavioral level of AHDL is designed to support the algorithmic formulations for digital systems at the schematic level. It can represent the pertinent sequential and parallel operations clearly. Specifically, this level is applicable to describing and simulating the behavior of systems, without going into the details of construction.

All objects in the behavioral level are defined by the set of " basic language elements " : separator, letter, digit, logical value, and abstract data types.

The separators are symbols chosen to do the following :

1. Represent ordinary punctuation marks to enhance the readability of any construct at the behavioral level.
2. Indicate timing by marking sequential and concurrent relationship among the linguistic structures.
3. Indicate the functional relationship among the different linguistic entities.

Letters and digits respectively do not have any inherent meaning. They serve to form the other linguistic structures.

Logical values are "true" and "false " as usually encountered. There are four abstract data types in the behavioral level of AHDL, namely : register, variable, declarator, and operator. They are the chief building blocks for the formation of any construct in the behavioral level.

The five basic language elements are combined together according to rules to form single actions and declarations . The declarations form a system to do the following :

1. They declare the existence of linguistic structures. In this case the declarators are simply " labels " without any intrinsic meaning.
2. They declare the properties of different objects. In this case the declarators are " reserved " words rather than freely choosable labels as in the previous case.

The single actions are the basic units of algorithmic formulation for the solution of design and descriptive problems. They are expressed by the three types of statements available in the behavioral level, namely : assignment statement , designational statement, and conditional statement. A statement is considered as an operational event occurring in a particular period of time.

Single actions which are acting as basic units of computations are expressed in assignment statements. Assignment

statements which characterise the RT level of operations are data transfer statements with the generalised syntax :

carrier ⟵ data operation expression

The operator (⟵) transfers the computed result by the data operation expression to the designated " carrier ". Any carrier which is assumed to have the ability to hold data information is either a register or a variable. The data operation expression represents either a rule to compute a value or a rule to present a value before a transfer. Any computation, which is always a " simple data transformation ", is based on either normal arithmetic composition or Boolean algebraic operation. It is always a simple data transformation because its formation involves only either just a single unary operator on a single operand or a single binary operator on two operands. The operands are themselves variables. The operators are symbols selected to abstract the designated data transforming mechanisms. In certain " simple data transfer ", the data operation expression contains no operator at all, but just a single variable representing the value for the transfer.

The single actions, which constitute the basic structural units of control sequencing, are designational and conditional statements.

A designational statement creates a control path between itself and a particular part in a construct. Control will then be transferred to that part for its execution. When execution is

completed, control will be returned to either the designational statement itself , or the single action right after the designational statement. There are, therefore, two types of designational statements : sequential designational statement, and dynamic multiplexed designational statement . Sequential designational statements are represented by procedure calls . When a procedure call is executed, a branch to the corresponding procedure for the subsequent execution will be performed. If the execution of the procedure is completed, control will be returned back to the single action right after the procedural-call statement. Associated with any dynamic multiplexed designational statement, the presence of a decoding process is always assumed. This decoding process decodes a particular condition encoded as one of the states in a register. The switching operator in the dynamic multiplexed designational statement, then, transfers control to the corresponding part of the construct for execution. When execution of that part is complete , control is returned back to the dynamic multiplexed designational statement itself.

Conditional statements are built up from assignment statements, designational statements, and even conditional statements themselves. The characteristics of conditional statements are summarised in the following :

1. The basic structure of any conditional statement is generalised in the syntax :

   condition ⟶ ACTION

   It says that the ACTION part will be executed only when the

" condition " of Boolean nature is " true ".

2. The condition of any conditional statement is reflected explicitly by the Boolean value of a Boolean expression or that of a relational expression. The Boolean value of a Boolean expression, which is of ordinary Boolean algebraic composition, is the function of the Boolean expression. The Boolean value of a relational expression, which represents a " test network " to compare the magnitudes of two variables, is the result of the corresponding comparison.
3. The condition of any conditional statement implies a particular time interval during which the ACTION is completely executed.
4. The ACTION part may contain a single " unconditional statement " which is either an assignment statement or a designational statement.
5. The ACTION part may contain a " block " of several unconditional statements.
6. The ACTION part may contain a " block " of unconditional statements and conditional statements intermingled together, i.e. conditional statements can be nested .
7. The execution of the statements within a block can be sequentially dependent , concurrently independent , or sequentially and concurrently intermingled . This timing relationship among the statements is marked by appropriate separators.

The control structure of the behavioral-level of AHDL is

based on the concept of a time block . This is a condition and execution relationship working on the principle of a conditional statement, i.e. a time block is defined by the syntax :

condition ⟶ a single action or a block of single actions

A conditional statement, however, is only one of the ways to represent a time block. The condition of a time block can be represented in three possible ways. First, it may be represented explicitly as the condition of a conditional statement. Second, it may be encoded as one of the states in a register. In this case, the recognition of the condition is performed by decoding. Third, its existence may not be explicitly defined by any structure. Instead, it may be assumed by default at the " entry point " where execution begins. For example, in a construct containing several statements in sequential ordering, the execution of a statement is conditioned by the completion of the preceding one.

The inclusion of concurrent single actions in the block of single actions of a time block is a rule rather than an exception. Furthermore, any single action in a block of single actions may represent another time block. Therefore the definition of a time block and a single action can be recursive.

Single actions can always be grouped together to be identified by unique labels. These groups are named blocks. The representational effect of a block is the same as a single action. The way of partitioning a block of single actions into

uniquely named subblocks, or sub-subblocks is referred to as the " blocking mechanism ".

The basic principle of control transfer is summarised in two steps. The first step is the interpretation of the condition of a time block. The second step is the activation and execution of the corresponding single actions. There are three modes of control transfer as described in the following :

1. When a condition is encoded as one of the states in a register, the control transfer is a dynamic and selective process. It involves decoding the condition and then interpreting the result for a particular path of control transfer. Special operators are incorporated for this mode of control transfers.
2. If a condition is represented by either a Boolean expression or a relational expression, the interpretation of the condition is directly assumed. Then control transfer to the corresponding block of single actions or the corresponding single action starts only when the condition is true .
3. In any construct which contains single actions or blocks of single actions in sequential ordering, control will pass down from a single action or a block to another sequentially and respectively. In other words, the execution of a single action or a block is conditioned by the completion of the preceding one. If a time-slot of concurrent single actions or blocks does exist between two sequential entities, the whole time-slot will receive the same control transfer as if it were a single entity. This sequential mode of control

transfer in any case can only be violated by procedure calls which imply " jumps " to skip a certain number of single actions or blocks for the execution of the corresponding procedures.

The structure of the behavioral level is " procedural " because sequential ordering of statements has a significant meaning. It implies the consideration of consecutive timing issues. This level is for formulating the framework or architecture of any digital system. Formulations in the behavioral level do not consider the intrinsic properties of the basic building blocks. For example, a register is declared as a storage element irrespective to its ultimate principle of operation. However, " synchronous " and " asynchronous " operations can be reflected by the indicator " ' " which will be described later in this chapter.

BASIC LANGUAGE ELEMENTS AND SOME IMPORTANT LINGUISTIC STRUCTURES

Letters do not have individual meanings, but they are used to form " identifiers " and " information strings ". The letter set can be extended arbitrarily with any distinctive characters. The basic set provided is

<letter> ::=

A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|

U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|
o|p|q|r|s|t|u|v|w|x|y|z

Identifiers must start with a letter, but individually they do not have any inherent meaning. They serve to identify the existence of different entities in a construct. Within a construct, the identifiers must be " unique ". The syntax of the identifiers is

<identifier> ::= <letter> | <identifier> <letter> |
<identifier> <digit>

The information strings are used for different purposes, but their principal purpose is to represent " number systems " in terms of bits, in the behavioral level. The syntax of information strings is

<information string> ::= nXr
" n denotes numerical representation "
" X denotes number-system designator "
" r denotes bit-length of n "

The symbols n, X and r are left to the choice of the users for convenience. Any information string is binary in nature because it is defined by bits. In the above syntax, X specifies the number system, r specifies explicitly the bit length of the string in terms of a " decimal number ", and n is the value

represented particular to the number system. For example, if B designates " binary ", O designates " octal ", and D designates " decimal ", then the value of " nine " may have the representation of " 001001B6 ", " 1106 ", or " 9D6 " respectively.

Digits are used to form identifiers, information strings, and " numbers ". The basic digit set provided is

<digit> ::= 0|1|2|3|4|5|6|7|8|9

Numbers have conventional structures and meanings. Real numbers are expressed in the form of

A"B

The symbol " separates the places of decimals specified by B from A, and the example is the real number : 15"824 .

Logical variables may assume one of the two binary values : true or false . They are represented by appropriate identifiers.

The standard functions and properties of the separators, in the basic set at the behavioral level, are listed in Table 2-1.

Table 2-1 Standard functions and properties of the separators in the basic set.

Separator	Explanation
,	indicates concurrent activities

;	indicates sequential activities
"	separates the places of decimals in real numbers
:	separates declarator on the left and the structure on the right in any declaration
:=	separates the switching operator from the list of labels in any dynamic selective designational statement
-	separates the upper limit from the lower limit by having the meaning of " from---to "
$	denotes the beginning of a comment
/	separates and defines the equivalence of two different representations containing the same number of entities; for example see Remark 3 at the end of this table
//	separates and associates two parallel entities on both sides respectively; for example see Remark 2 at the end of this table
begin end	brackets to delimit the definition of an algorithm
( )	brackets to contain the parameter list of any procedure call, and to be used freely to enhance readability
* *	brackets to contain label list for sequencing control
< >	brackets to contain subscripts for one-dimensional arrays
[ ]	brackets to contain the subscripts for the " rows " of the two-dimensional arrays
{ }	brackets to be used freely to enhance readability

Remark :

1. " Indexing " in the brackets of " < > " , " [ ] ", " * * "

always follows the convention of " from left to right ". When the separator " ; " is used in the brackets of " < > " and " [ ] ", it acts like the separator " ~ " .

2. The two parallel entities separated and associated by the separator " // " are subject to the convention of indexing. The matching of " entity1 // entity2 " to the corresponding counterparts must be one to one correspondence. The principle of the matching is generalised as : entity1 // entity2 " matching mechanism " part1 //part2 . Therefore entity1 is matched to part1, and so is entity2 to part2. If entity1 contains two subentities as " x,y ", and part1 contains two subentities as " p,q ", then the simultaneous matching is " x to p " and " y to q ", i.e. corresponding to the position indexed. The matching mechanism is generally an appropriate operator. In the example of

(Addition//Subtraction//Division)⟶(A//S//D)

the results of the parallel operations Addition, Subtraction, and Division are transferred to the registers A, S, and D respectively and parallelly. The symbol " " is the transfer operator.

3. Indexing is also applied to the both sides of the separator " / ". For example, in the construct of

A,B,C,D / a,b,c,d

A and a define each other, and so do B and b, C and c, and D and d.

The abstract data types : declarator, register, variable, and operator are the principal building blocks for any construct in the behavioral level. They are explained in detail in the next four sections.

DECLARATOR

Declarators are the basic entities in the system of nomenclature (which will be discussed in more detail in one of the latter sections). They serve to define the properties of identifiers and to declare the existence of different co-existing individual linguistic structures within a construct. In the behavioral level, some declarators are tuned to have inherent meanings. They are called the " reserved declarators ". The basic set of reserved declarators contains " Register ", "Subregister ", " Memory ", " Operator ", " Integer ", "Real", "Boolean ", " Operation ", " Sequence ", " Bus ", and " Procedure ". This set can be extended at will by "predefinition " to compensate for any deficiency that may arise. The purpose of the reserved declarators is to define the properties of different identifiers. The existence of different individual linguistic structures, which may be identified for the sake of clarity when structuring a behavioral-level construct, can be declared by non-reserved and unique labels. The syntax of any declaration is

declarator : structure

For example, the declaration of

Register : A

means that the identifier A possesses the property of a register, and the declaration of

L : A ←— Y

means the corresponding assignment statement to be declared by the label L.

REGISTER

Registers are identifiers declared as data carriers. They may appear in the form of " simple identifiers " without any subscript, for example :

Register : A

Sometimes, registers represented by unsubscripted identifiers do not provide the necessary information, because they do not show the binary nature and the word-length of them. The provision of subscripts makes registers into one-dimensional arrays. The brackets " < > " are particular to the one-dimensional subscript specifications.. The incorporation of indexing in these brackets makes the binary nature of a register clearer. In the first example :

Register : A <0;4>

the subscripts of 0 and 4 are respectively the lower and upper limits for the word-length of the register A. In the second example :

Register : A <0,1,2,3,4>

each bit is clearly indicated positionally from left to right.

Registers can be decomposed into subregisters by the appropriate " subregister declarations " of the syntax :

Subregister : identifiers of subregisters

The decomposition process is typified by the two examples presented in the following :

1. Example of decomposing a register into subregister :

Register : A<0;4>

Subregister : A <0;2>

A <3;4>

$ The register A <0;4> has been decomposed into subregister A <0;2> and subregister A <3;4>.

2. Example of decomposing a register into bits :

Register : A<0;4>

Subregister : A<0>

A<1>

A<2>

A<3>

A<4>

$ The register A<0;4> has been decomposed into bits which have been declared as subregisters.

The concatenation operator " @ " allows the formation of a compound register by cascading several different registers positionally together, i.e. the opposite process to the decomposition a register. The compound register thus formed will

assume all the characteristics of the component registers which each individually, positionally, and uniquely contributes to the overall register specification. The compound register is represented by a new identifier. (Assignment of an identifier to a newly formed compound register is part of the " renaming mechanism " which will be discussed in more detail in one of the latter sections.) The formation of a compound register is illustrated in the following example :

```
Register : B<0;3>
           C<0;3>
           BC<0;7> # B<0;3> @ C<0;3>

    $ The compound register formed by cascading
      the registers B and C has been renamed by the
      operator " # " to be BC
```

Obviously from the above example, the operator " @ " must override the operator " # " in the precedence of operations.

In fact, the decomposition of a register into subregisters or even into bits can also take place within the subscript by the aid of the concatenation operator. The subscript is then decomposed into different portions to be identified by unique identifiers. These identifiers are subsequently dimensioned by the brackets " < > " with the subscripts indexed corresponding to the original position in the " source register ". For example, if E<0;10> is the register to be decomposed deliberately into E<op @ adr>, then op and adr can be

dimensioned and indexed to have the designated format of E<op<0;4> @ adr<5;10>>. Therefore the two potential subregisters can be isolated as E<op> and E<adr>, or as op<0;4> and adr<5;10>, to be renamed and declared. The stacking of registers of the same word-length together forms a memory which is a two-dimensional array. The dimension of the stacking is contained in the brackets " [ ] ". The syntax of a memory is

MR c;d]<a;b>

MR is the identifier of the memory to be declared. c and d tells how many registers are there, and a and b denote the word-length of each register. The decomposing and renaming mechanisms, which produce distinctive subregisters or bits, can also be applied to decompose a memory into memory blocks or single memory cells (individual registers) . Such mechanisms, however, will be applied to the content contained in the brackets " [ ] " instead of that in the brackets " < > ".

VARIABLE

A variable is an identifier to which either a Boolean value, a real value, or an integer value is assigned. It can be used as an operand in any " computation " or " control formulation ". Any computation is defined by the data operation expression of an assignment statement, and any control formulation is regarded as the process of defining the condition for an execution . In the behavioral level, variables can be identified in three possible ways. First, identifiers can be

explicitly declared as either Boolean, integer, or real variables. Once a variable has been declared, it possesses the property as declared permanently, and it must be operated on by the same type (Boolean, arithmetic) of operators. Second, declared data carriers like registers, subregisters, and identified bits can be taken as variables, but the properties of these variables are determined by the nature of the operators acting on them. Third, variables can be freely defined as " catalytic identifiers " to represent the required intermediate stages. Then, their nature depends on the nature of the corresponding assignment statements. The example of a catalytic identifier will be given in the section of CONTROL STRUCTURE later..

OPERATOR

The operators in the behavioral level serve three purposes : computation, control formulation, and the formation of syntactic structures. Operators for computations are referred to as " computational operators ", operators for control formulations are called " control operators ", and operators for the formations of syntactic structures are called " syntactic operators". Any operator is classified as computational operator if its operation has to be defined by an " data operation expression ". Some of the operators primarily designated for control formulations operate as if they are computational operators. Therefore they are included in the class of computational operators. Since some of the Boolean operators may

be used for computations as well as control formulations, they are included in both the classes of computational operators and control operators. Table 2-2 shows the complete set of computational operators subdivided into different groups according to functions. Table 2-3 illustrates the unary operations of the computational operators on the register A. Table 2-4 illustrates the binary operations of the computational operators on the registers A and B , and the constant C (i.e. integer variable). Table 2-5 shows the complete set of control operators subdivided into different groups according to functions. Table 2-6 illustrates the unary and binary operations of the control operators on the Boolean variables D and E, and the integer variable K. Table 2-7 shows the complete set of syntactic operators. It should be noted that the classification of the basic set of operators into the three different types of operators is made only to provide a clearer concept.

Although the basic set of operators is sufficient to meet most of the general requirements, occasions may arise, where special operators are needed to be defined to describe either non-standard, or higher-level functions. The fact that an operator actually abstracts a modular logic network implies that a process to define a special operator corresponds to the definition of a particular module. The definition of any special operator can be worked out by either one of the two ways : simple declaration, and procedure call.

The simple declarations are having the syntax :

Operator : OPERATOR IDENTIFIER begin ALGORITHM end

For example, the operator which represents the operation of a A-to-D converter is declared :

```
Operator : convert

        begin

        Integer : A   $ A represents an analog signal
                        changing respective to time

       Register : D

                  (A>D) ——→ D ←—— count-up D

                  (A<D) ——→ D ←—— count-down D

        end
```

```
$ The capability of the operator " convert "
$ is assumed by the declaration. Then, it can
$ be used like other standard operators
$ available in the behavioral level of AHDL.
$ What is delimited in the brackets " begin
$ end " serves to depict the basic property
$ of this special operator, and it has the
$ same meaning as the assignment statement
$ of : D ←— convert A .
```

" convert " is the operator with the working algorithm defined within the separators (actually brackets) " begin " and " end " . In this case the operator " convert " serves more for documentary purpose. It specifies the fact that an analog value represented by the integer variable A is converted into a binary value to be stored in the register D. The actual conversion-mechanism is not specified here, but its existence is implied.

The detailed working algorithm of any special operator can be theoretically described in an appropriate procedure . The application of the special operator then becomes a procedure call. The procedural algorithm may provide the framework for actual implementation. Therefore procedures to represent special operators are not only for documentary purposes, but they may represent the simple and ultimate drafts in the early phase of a design process. Procedures will be discussed in more details in one of the latter sections.

Table 2-2 The complete set of computational operators

Operator Group	Symbol	Explanation
arithmetic	+	conventional addition
	-	conventional subtraction
	x	conventional multiplication
	÷	conventional division
logical	¬	logical NOT
	v	logical OR
	^	logical AND
	⊕	logical EXCLUSIVE-OR
vectoral	shift	conventional right shift
	rotate	circular left shift
	exchange	swap the contents of two registers
special	count-up	count up by one
	count-down	count down by one
	decode	select one out of " 2 to the power n " possibilities

transfer ⟵	transfer the computed result to the designated carrier

Table 2-3 Unary operations of the computational operators on register A.

Expression of Unary Data Operation	Explanation
vA	logical OR all bits of A
^A	logical AND all bits of A
⊕A	logical EXCLUSIVE OR all bits of A
.A	logical COMPLEMENT all bits of A
rotate A	shift A one bit to the left with the content of the leftmost bit inserted into the position of the right-most bit
shift A	shift A one bit to the right with the rightmost bit dropped off
count-up A	increase the content of A by one
count-down A	decrease the content of A by one
decode A	select one out of "2 to the power n" possibilities, and n is the number of bits in the register A

Table 2-4 Binary operations of the computational operators on the registers A and B, and the constant C.

Expression of Binary Data Operation	Explanation
A+B	add the content of A to that of B
A-B	subtract the content of B from that of A

AxB	multiply the content of A to that of B
A÷B	divide the content of A by that of B
AvB	logical OR the corresponding bits of A and B
A^B	logical AND the corresponding bits of A and B
A⊕B	logical EXCLUSIVE OR the corresponding bits of A and B

The complement operator " . " is not a binary operator.

C shift A	shift A by C bits to the right
C rotate A	left circular-shift A by C bits
A exchange B	swap the contents of A and B
C count-up A	count up A by C times
C count-down A	count down A by C times
A←B	transfer the content of B to register A

The operator " decode " can not be used as a binary operator, because the standard format of a decoding operation is :

K ← decode AB

Table 2-5 The complete set of control operators.

Operator Group	Representation	Explanation
logical	.,v,^,⊕	same as used for computations
relational	>,<,<=,>=,=,.=	greater than,less than,less than or equal to, greater than or equal to,equal to,and not equal to respectively
transfer	⟶	transfer of control on a condition
switching	switch	functionally similar to the FORTRAN " computed GOTO " for " selective control transfer "

sequential	call	sequential control transfer, but the return of control is also implied; used only in procedure-call statements

Table 2-6 Unary and binary operations of control operators.

Operator Group	Nature of Operation	Example
logical	unary and binary	refer to Table 2-3 and Table 2-4
relational	binary	D=E means comparing D and E for equality
transfer	unary	activates the counterpart of a condition
switching	unary	the example is : switch K := * x,y,z * ; " switch " works on the integer variable K to transfer control to one of the identifiers in the brackets " * * ".
sequential	special	e.g. do COSINE; calling procedure COSINE

Table 2-7 The complete set of syntactic operators and their operations.

Operator	Symbol	Explanation
concatenation	@	The operator cascades different objects of the same properties together to form a complex structure, e.g. A@B, whereas a compound register is formed from registers A and B.
renaming	#	The operator renames a syntactic

structure. For example, C # A@B has the compound structure renamed C.

Remark : The operator " @ " overrides the operator " # " in the precedence of operations.

BLOCKING MECHANISM

The purpose of the blocking mechanism is to decompose the block of single actions of a time block into subblocks or even sub-subblocks to be identified by unique labels. The blocking mechanism is part of the system of nomenclature for any construct, and it has the syntax :

<blocking mechanism> ::= * <label list> *

<label list> ::= <label> | <label> <separator> <label list>

<label> ::= <identifier>

The " separator ", which is either a separator to denote concurrent occurrence of labels or a separator to denote sequential occurrence of labels, can be inserted repetitively according to needs. The " label list " is contained in the brackets " * * ". For example, the time block :

condition ⟶ * A1; A2; A3 *

is the result of the application of the corresponding blocking mechanism. The label list contains the labels A1,A2,and A3. The labels which each identifies a subblock are separated from one another by the separator " ; ". This separator denotes that the execution of each subblock is sequentially performed from left to right. Each label can further be decomposed by the very same blocking mechanism to produce sub-subblocks.

RENAMING MECHANISM

The renaming mechanism is part of the system of nomenclature to rename previously defined structures with new names. The new name then takes up all the characteristics of the old name, except that the syntactic representation of the new may be different from the old. The operator for renaming is " # " . For example, the process of forming a compound register from the registers : A<0;5>, and B<0;5> is

AB<0;11> # A<0;5> @ B<0;5>

The compound register formed has been renamed to be AB with different subscripts from that of registers A and B.

SYSTEM OF NOMENCLATURE

The system of nomenclature for any construct contains three basic mechanisms : declaration, blocking, and renaming. Declarations are ultimate to any construct at the behavioral level. The other two mechanisms cope with various unusual

situations. The system of nomenclature is important for any construct because, it enhances the understandability, readability, extensibility, changeability, and workability of it.

PROCEDURE CALL

Procedure call is the mechanism to ensure programming hierarchy which is a modular approach to solve problems. A procedure itself is a " program module " in AHDL. The syntax of any procedure call is

```
      <procedure call> ::= call <procedure identifier>
<procedure identifier> ::= <identifier> <subscript list>
   <subscript list >   ::=  (  <list of parameters>  )  |
                            (  <empty>  )
<list of parameters>   ::= <list of identifiers>
```

The procedure calls must be sequenced by the operator " call ". The parameters are then passed down from the procedure call to the corresponding procedure . The current values of the parameters are only defined in the procedure-call statement. The syntax of a procedure is

Procedure : procedure body

The procedure body is a block of single actions starting with a " procedure header " defined below :

<procedure header> ::= <procedure identifier> <formal parameter list>

<formal parameter list> ::= (<formal parameters>)

The number of parameters in the list of parameters in any procedure call statement must be equal to the number of " formal parameters " in the respective procedure header. The parameters are linked by one to one positional correspondence. After a procedure has been executed, control is passed back to the single action which follows the corresponding procedure-call statement. A procedure call operation is illustrated by the following example :

```
                ----
                call COSINE (A,B,C)   $ procedure-call statement
                ---
   Procedure  : COSINE (a,b,c)        $ procedure header
                ----
```

In the above example, A,B and C are the actual parameters, and a,b,and c are the formal parameters.

PROCEDURE CALLS AS AN INTERACTING TOOL

Procedure calls at the behavioral level are not restricted to calling procedures structured in terms of the behavioral

level . They can be used as a tool to interact the behavioral level with the other two levels of AHDL. More precisely, behavioral-level procedure calls can call procedures constructed in terms of the functional level or the structural level. Such interaction of levels allows programming hierarchy to match with the required " information hierarchy ". The latter is defined as a process of providing more detailed information in each successive level. The three levels of AHDL represent such an information hierarchy.

CONTROL STRUCTURE

Sequencing of the execution of single actions or blocks of single actions is basically performed by either dynamic and selective control, or sequential control. The principle of dynamic and selective control is the interpretation of the condition of a time block, followed by the execution of the corresponding block of single actions. It is dynamic because all the time blocks, at the same level of operations within a construct, have the same equal chance to be selected. It is selective because one and only one condition will become true at a particular time interval. The fundamental principle of sequential control is that the execution of a single action is conditioned by the completion of the preceding one.

Dynamic and selective control transfer are explicitly expressed by the help of the operator " decode " and the operator " switch ", the separator " := ", and a " label list "

derived from the blocking mechanism. Such a dynamic and selective control transfer is examplified by the following example :

```
K ←— decode A ,
switch K := * A1, A2, A3 *
```

The register A is decoded by the operator " decode " to produce an integer value to be assigned to the variable K. Simultaneously, the current value of K is matched positionally to the corresponding label in the label list. The interpretation and the matching of the value of K is performed by the operator " switch ". The label list contains solely " concurrent labels " as indicated by the separator " , ". The variable K is a catalytic identifier. It is defined as a matter of convience. Its identity is not previously declared.

Sequential control transfers may be assisted by the separator " ; ", and sometimes a label list derived by the blocking mechanism. Special provisions for sequential control transfers are procedure calls.

EXPRESSION AND STATEMENT

Any expression in the behavioral level of AHDL is defined as a quantity having either a single variable, one or more operators acting on one or more associated variables respectively, or a procedure identifier . The use of appropriate

separators in an expression is always possible. Expressions are the basic components from which statements are structured. There are three types of statements in the behavioral level : assignment statement, designational statement, and conditional statement.

Assignment statements specify the way to assign computed values to the corresponding carriers. The type of value to be assigned is determined by the expression part of any assignment statement, terminologically called the " data operation expression ". This kind of expression is basically computational. The carrier which receives the value assigned assumes the property of the value. The syntax of an assignment statement is

carrier ⟵ data operation expression

Designational statements specify the way to sequence the execution of single actions or blocks of single actions. There are two types of designational statements provided : " switch statement " and " call statement ". The switch statement for dynamic and selective sequencing control is having the syntax, in terms of an expression, as the following :

switch VARIABLE := EXPRESSION

The VARIABLE is always the " integer type ". The EXPRESSION is, in fact, a label list derived from the blocking mechanism. For example, in the switch statement :

switch K := * A1, A2, A3, A4 *

control will be transferred to A1, A2, A3, and A4 correspondingly one at a time if the value of K is equal to one, two, three, or four respectively at that particular time interval. The call statement is for sequential control. It has the syntax in terms of an expression :

call EXPRESSION

The EXPRESSION is represented by a procedure identifier.

Conditional statements which represent time blocks explicitly have the syntax :

condition a single action or a block of single actions

Since a single action may represent any statement (or a pause of no operation in exceptional cases), conditional statements are structured from different kinds of statements.

ACTIVE-TRANSITION INDICATOR

The indicator " ' " indicates the " active-edge behaviors " of any timing pulse which is identified as a variable. For example, if K is the timing pulse, then 'K indicates a leading-edge transition , and K' indicates a trailing-edge transition . The example which illustrates how to use the indicator " ' " is the following :

```
$ Illustration of the use of the indicator " ' "

   Register : A<0;7>

  Algorithm : 'K ⟶ A ⟵ count-up A

$ In the above example, the " Algorithm " says that
$ at the leading-edge transition of K, the content
$ of the register A is increased by one.
```

In the above example, the indicator " ' " indicates that at the leading-edge transition of K , all the flipflops in the register A react simultaneously . Thus all the flipflops are synchronized to the timing pulse K.

SYNCHRONOUS AND ASYNCHRONOUS DATA TRANSFERS

The control structure based on the concept of a time block can be symbolised by Figure 2-1.

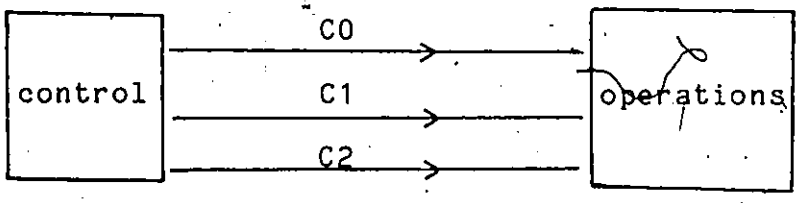


Figure 2-1 Symbolism of the time-block concept

C0, C1, and C2 are the conditions of three different time blocks. Theoretically, data transfers, in the forms of

assignment statements, with respect to a particular condition can be synchronized by the condition. For example, in the construct :

```
'C0 ⟶ data transfer 1 ,
      data transfer 2 ,
      data transfer 3 ,
```

the three data transfers are synchronized by C0 at the leading-edge transition. If " C0 " is used for the condition instead of " 'C0 ", then the three data transfers become concurrent instead of being " synchronous ". It is no way to know when each of the three data transfers is executed in the duration of the condition " C0 ". Therefore concurrent data transfers are considered as " asynchronous ". Under certain circumstances, it is easier to write down data transfers sequentially related corresponding to a particular condition, i.e. data transfers in sequential ordering. For example, in the construct :

```
C0 ⟶ data transfer 1 ;
      data transfer 2 ;
      data transfer 3 ;
```

the execution of " data transfer 3 " depends on the completion of " data transfer 2 ", which in turn depends on the completion of " data transfer 1 ". The three data transfers are considered as asynchronous because with respect to C0 , they are not

executed at the " same time point ".

SUBSCRIPT

Subscripts play an important part in " indexing " in AHDL. Subscripts for one-dimensional arrays are enclosed in the brackets " < > ", and subscripts for two-dimensional arrays are enclosed in the brackets " [ ] " . The convention chosen for indexing subscripts in AHDL is from left to right.

It is not always necessary that subscripts be presented in the form of integer numbers. Labels, arithmetic expressions, and appropriate separators are always allowed to be used to structure appropriate subscripts. In the example :

```
Subregister : A<10>
              B<a,b,c>
              C<I+1>      $ I is an integer variable
```

10, a, b, c, and "I+1" are subscripts.

SUMMARY AND COMMENTS

The behavioral level of AHDL is designed principally for schematic formulation. This chapter has discussed the basic linguistic elements and characteristics of the behavioral level. The other two levels presented in the following chapters are

designed to provide the tools for detailed representation of circuit modules and elements. It should be noted, however, that the power and versatility of AHDL is due not only to the distinctive formulations available at each of the three levels, but also to the possible interaction of the three levels through " procedural co-existence " in any construct. Therefore AHDL has a fluid hierarchy of three distinctive levels.

The context of procedural co-existence is to call procedures constructed in terms of one level of AHDL by procedure calls expressed in terms of another level of AHDL. It is a problem of practical application of AHDL, and a methodology to deal with the need of appropriate and desirable information for different parts of a digital system. The interaction of levels in AHDL is based on the three criteria : programming hierarchy , information hierarchy , and " information hiding ". They are explained in the following :

1. Programming hierarchy is defined as a modular approach to solve problems.
2. Information hierarchy is defined as a structure to provide more detailed information as going down from one level to another lower level. The three levels of AHDL may represent an information hierarchy.
3. Information hiding is defined as a process to avoid unnecessary details during any algorithmic formulation.

EXAMPLE

Two more examples of formulations at the behavioral level of AHDL are presented in the APPENDIX section. The only example attached here as an illustration is the behavioral description of the PIA (MOTOROLA MC6820).

Example :

```
$ The behavioral-level description of the PIA (MOTOROLA MC6820)
$ looks at the PIA like a black box. It illustrates the basic
$ principle of the HANDSHAKE algorithm for the PIA in a concise
$ AHDL representation. Detailed descriptions of the internal
$ organisation, physical working principles of the different
$ components, and fabricating information will not be given
$ at this level. The block diagram of the PIA is illustrated
$ by Figure 2-2.
```

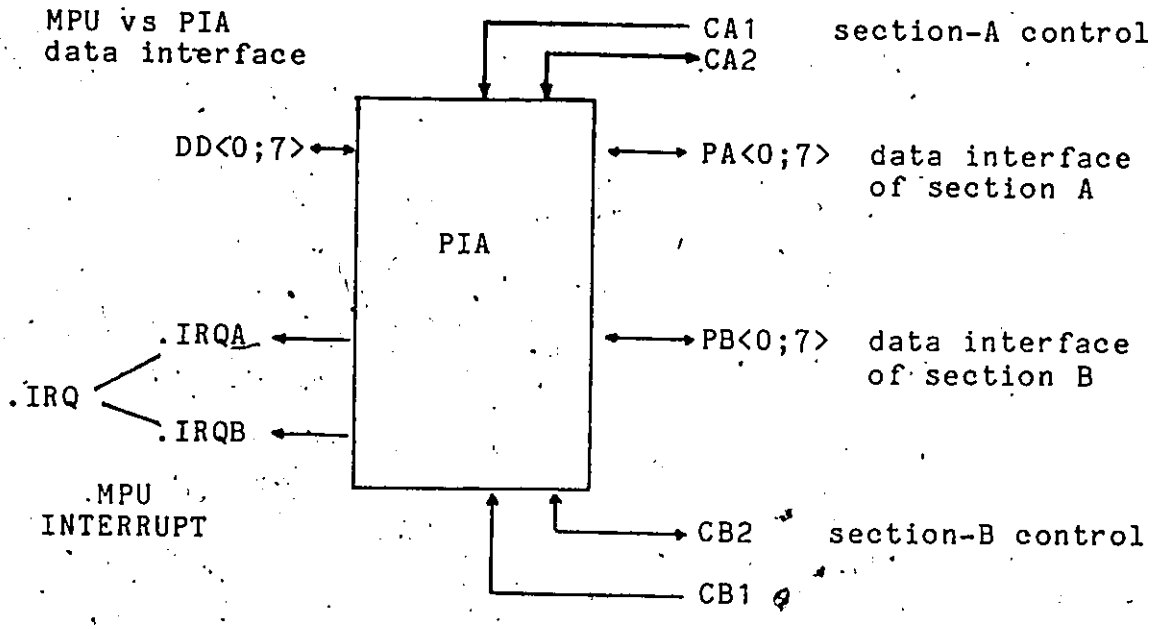


Figure 2-1 Block diagram of PIA (MOTOROLA MC6820)

```
$ In order to make the HANDSHAKE description clearer, Boolean
```

```
$ variables and registers are symbolically defined. The ACTIVE
$ TRANSITION of the pertinent control signals is specified by
$ the indicator "'". For example, 'K represents the transition
$ at the leading edge of K, and K' for the trailing-edge
$ transition. The " SEQUENTIAL CONTROL " for the execution of
$ sequential statements is indicated by the separator " ; ".
$ The "BLOCKING MECHANISM", in the form of : * label list * ,
$ eases the formulation of the behavioral-level description.
$ All the AHDL elements used in this description can be
$ found in the appropriate places in the text of Chapter Two.

$ The behavioral-level description of the PIA begins here

    Bus : DD<0;7>   $ bidirectional data bus between MPU and
                      PIA
          PA<0;7>   $ bidirectional data bus between PIA and
                      peripheral devices (section A)
          PB<0;7>   $ bidirectional data bus between PIA and
                      peripheral devices ( section B )

Boolean : CA1       $ control variable from peripheral devices
                      to PIA( section A)
          CB1       $ control variable from peripheral devices
                      to PIA( section B)
          CA2       $ control variable from PIA to peripheral
                      devices( section A)
          CB2       $ control variable from PIA to peripheral
                      devices( section B)
          IRQA      $ interrupt control variable from PIA to
                      MPU( section A)
          IRQB      $ interrupt control variable from PIA to
                      MPU ( section B)
          DA        $ direction control variable for data
                      transfer, i.e. input or output in
                      section A (catalytic variable ).
          DB        $ direction control variable for data
                      transfer, i.e. input or output in
                      section B (catalytic variable )
          CRA       $ control variable acting as the mediator
                      for handshake in section A
                      (catalytic variable )
          CRB       $ control variable acting as the mediator
                      for handshake in section B
                      (catalytic variable)
          REP       $ control variable abstracting the
                      response from the MPU upon interrupting
                      by PIA (catalytic variable)

Register : BUFA<0;7>  $ buffer register in section A
           BUFB<0;7>  $ buffer register in section B
```

```
Algorithm : * Start ; Data *   $ algorithm begins here and
                                 the condition to activate
                                 the algorithm is assumed
                                 by default; application
                                 of the BLOCKING MECHANISM
                                 is illustrated

   Start : * SecA , SecB *

      SecA : 'CA1⟶CRA⟵1,    $ set CRA at leading-edge
                               transition of CA1

             'CRA⟶IRQA⟵0     $ pull IRQA low

      SecB : 'CB1⟶CRB⟵1,    $ set CRB at leading-edge
                               transition of CB1

             'CRB⟶IRQB⟵0     $ pull IRQB low

$ The block " Start " describes that part of the handshake
$ configuration from the peripheral devices through the PIA
$ to the MPU.

 Data : * Response ; Transfer *   $ Response means the result
                                    of polling the status of
                                    the PIA upon interrupt

   Response : .IRQA ⟶ REP ⟵ 1, $ REP is assumed for clarity

              .IRQB ⟶ REP ⟵ 1

   Transfer : * Section-A , Section-B *

Section-A : * input , output *

     input : REP^.DA ⟶ DD ⟵ PA ;  $ input data from
                                    peripheral devices

                     CRA ⟵ 0  ,  $ reset CRA to low

                     CRA' ⟶ CA2 ⟵ 0   $ input acknowledge

    output : 'REP^DA ⟶ BUFA ⟵ DD ; $ input to buffer
                                     register

                       CRA ⟵ 0  , $ reset CRA to low

                       CRA' ⟶ CA2 ⟵ 0   $ output ready
```

```
Section-B : * in , out *

      in : REP^.DB ---->DD <----PB ; $ input data
                        CRB <---0  , $ reset CRB to low
                        CRB' ---->CB2 <---0 $ input acknowledge

     out : REP^DB ---->PB <----DD ; $ output data
                        CRB <---0  , $ reset CRB to low
                        CRB' ---->CB2 <---0  $ output ready

 $ In order to make the behavioral-level description for
 $ the PIA clearer, a verbal description is provided here.
 $ The input-output processes between the MPU and the
 $ peripheral devices are through the three data buses :
 $ DD, PA, PB. The direction of either input or output
 $ is determined by the catalytic control variables
 $ DA and DB. The algorithm for the bidirectional
 $ communications is defined by the block of single
 $ actions named Algorithm. The algorithm first describes
 $ the appropriate initiations and then the processes of
 $ of data transfer. The PIA is physically divided into
 $ two parts which are symmetrical. The behavioral-level
 $ description provides a glimpse into the working  principle
 $ of the PIA without any regard to its actual application
 $ because initialisation of the appropriate registers
 $ in the PIA is necessary in any realistic designated
 $ application. Besides, the behavioral-level description
 $ was efficiently worked out by the application of the
 $ pertinent blocking mechanisms and the appropriate
 $ recognition of the corresponding time blocks.
```

CHAPTER. THREE

FUNCTIONAL LEVEL OF AHDL

INTRODUCTION

The control structures, the operators, and the various system components abstracted at the behavioral level of AHDL may represent very complex combinatorial and sequential logic networks. If more information, or even the ultimate working principles of these logic networks are required, then descriptions at the functional level of AHDL are necessary.

In the design process of any digital circuit, the first step of algorithmic description at the behavioral level must be followed by the next logical step, i.e. the precise description of the component modules. This step implies an exact specification of the I/O relationship of the component modules within a system, and hence their basic principles of internal operations. The functional level of AHDL corresponds to the second step of the design process. The set of language elements provided by the behavioral level are usually required in this level. However, additional elements are needed for formulating clear and precise functional-level constructs. In addition, the functional-level structure must support the " non-procedural " structures which may be involved in the formulations of the pertinent digital control.

The functional level is characterized by the two abstract data types which are not present in the behavioral level. These two abstract data types are : " state ", and " terminal ". States are expressed in the form of " state variables ", and terminals are expressed in the form of " terminal variables ".

The context of the functional level is summarized in the following objectives :

1. To provide a clear modular approach.
2. To represent the input and output of modular components in terms of terminal variables which are the basic elements for " wiring ".
3. To define the ultimate working principles of the combinatorial logic networks in terms of Boolean operators, terminal variables, and the modes of operations : rippling , and parallel operation .
4. To define sequential logic networks in terms of terminal variables, Boolean operators, state variables, " state transitions ", and the modes of operations : synchronous , and asynchronous .
5. To accommodate different design techniques.
6. To represent more detailed timing issues than in the behavioral level.

TERMINAL

Terminals are input and output logic circuit nodes of modular units, and each terminal represents one and only one logic circuit node. They take on Boolean values as functions of time, and the current values are always assumed. Terminals are the basic elements for wiring . Any terminal, which must be identified by unique identifier called the terminal variable, is validated only by the corresponding " terminal declaration " with the syntax :

Terminal : terminal variables

Terminal variables can be unsubscripted or subscripted identifiers. In other words, terminal variables may be dimensioned and decomposed like registers and subregisters. In the example :

Terminal : A
B<0;2>

A represents a single terminal, and B<0;2> represents three terminals namely : B<0>, B<1>, and B<2>.

Any terminal variable can be specified by a Boolean expression composed of the appropriate Boolean algebraic operation. The syntax of any terminal variable to be specified by the coresponding Boolean expression is

terminal variable :- Boolean expression

The equating operator " :- " associates the terminal variable on the left defined by the Boolean expression on the right. In the example :

Terminal : C :- X ⊕ Y

the function " X ⊕ Y " is associated to the terminal variable C.

Terminal declarations may be associated to structures previously declared. The associations are indicated by the separator " :: " with the syntax :

declarator : IDENTIFIER :: TERMINAL DECLARATION

The TERMINAL DECLARATION then defines what terminals are possessed by the IDENTIFIER. This act makes the IDENTIFIER a distinctive module. The example is

```
Register : RP :: Terminal : U<0;5>    $ six input terminals
                            V<0;5>    $ six output terminals
```

In the above example, RP is the register which possesses six input terminals defined by U<0;5>, and six output terminals defined by V<0;5>.

RIPPLING AND PARALLEL OPERATIONS

The rippling operations and the parallel operations for combinatorial logic networks are represented by special

formalisms in the functional level.

The formalism for parallel operation is the notation :

TR<a~b> :- F(X<c~d>, Y<e~f>, ...)

The identifiers TR, X, Y and others likewise, if necessary, are terminal variables, whereas " F " is any Boolean function: The subscripts from a to b, from c to d, from e to f, and others likewise must be " corresponding ". Concisely, the formalism says that TR<a> is the terminal associated by the operator " :- " to the Boolean function " F " which operates on the variables X<c>, Y<e>, and the others in a similar manner; and so do TR<x> with respect to X<x>, Y<x>, and other terminal variables with the " x " subscript. x represents the subscript at a designated position within the range of the subscripts delimited by the corresponding brackets " < > ". Two examples are given below to illustrate the application of such a formalism for parallel operations.

The first example is the following construct :

```
$ Functional-level description of a 24-bit parallel adder
   Terminal : A<0;23>
              R<0;23>
   Operator : + :: Terminal :
      C<23> :- 0      $ the 23rd carry is always " low "
                        (Boolean)
      C<0~22> :- A<1~23>^R<1~23>vR<1~23>^C<1~23>vC<1~23>^A<1~23>
      sum<0~23> :- A<0~23>⊕R<0~23>⊕C<0~23>
```

In the above construct, the 24-bit parallel adder abstracted in the behavioral level by the operator " + " is completely defined by the terminal declaration. The behavior of the parallel operation has been clearly depicted by the formalism stated earlier.

The second example is the functional-level description of the circuit shown in Figure 3-1.

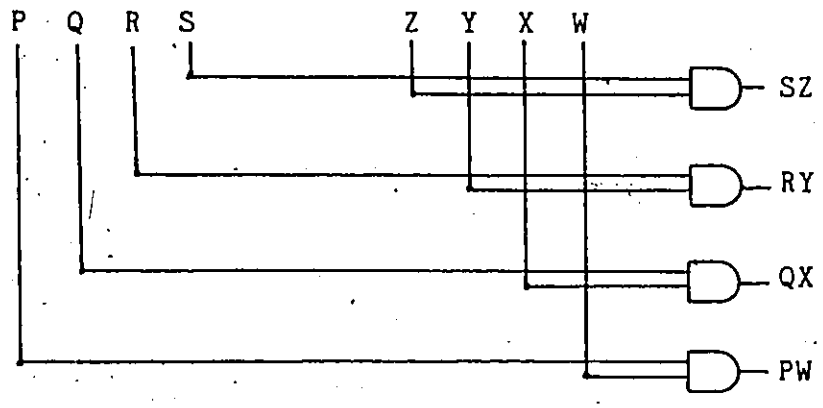


Figure 3-1 Four parallel ANDing operations

The functional-level construct for Figure 3-1 is the following :

```
$ Functional-level construct for Figure 3-1

    Terminal : Ga<P,Q,R,S>
               Gb<W,X,Y,Z>
               Gc<PW,QX,RY,SZ> :- Ga<P,Q,R,S> ^ Gb<W,X,Y,Z>
```

In the above construct, the four parallel ANDing operations are clearly expressed.

The formalism for the rippling operation is the " DO " statement represented below :

DO label RANGE IDENTIFIER=a;b,MODIFIER

The mechanism of this DO statement is different from the classical FORTRAN DO statement, because the variation of the RANGE IDENTIFIER can be " ascending " or " descending ", depending on whether " b is greater than a " or " a is greater than b " respectively. The incorporation of the MODIFIER is an option. It states how the RANGE IDENTIFIER changes. By default, the RANGE IDENTIFIER changes by one. The MODIFIER must be written down as an integer number, for example :

DO add J=1;10,,2

has the MODIFIER equal to " 2 ", and thus the RANGE IDENTIFIER J changes by two. The construct which illustrates the application of the DO statement is the following :

```
$ Functional-level description of a 24-bit ripple adder

Terminal : A<0;23>
           R<0;23>
```

```
Operator : + :: Terminal :
    C<0> :- 0  $ the 0th carry is always " low "
                 (Boolean)
    DO addition  I=0;23      $ iterating in ascending order
    addition : C<I+1> :- A<I>^R<I>vA<I>^C<I>vR<I>^C<I>
               sum<I> :- A<I>⊕R<I>⊕C<I>
```

STATE

For sequential logic networks, terminal variables alone are not enough to represent all the necessary information involved. Another abstract data type called " state " has been defined to compensate for this deficiency. Any state, which represents " a stage of performance " at a particular time interval, is identified by the corresponding " state variable ". State variables are only validated by the corresponding " state declarations " with the syntax :

State : state variables

State variables can be unsubscripted identifiers or subscripted identifiers. More precisely, state variables may be " dimensioned and decomposed like memories ". In the example :

```
State : ST .
        SV[0;2]
```

ST represents a single state, and SV[0;2] represents three states : SV[0], SV[1], and SV[2].

State declarations may be associated to structures previously declared. The corresponding association is indicated by the separator " :: " with the syntax :

declarator : IDENTIFIER :: STATE DECLARATOR

The STATE DECLARATION then defines what states are possessed by the IDENTIFIER. Such an association makes the IDENTIFIER a distinctive module. In the example :

```
Register : R :: State : C[0;2]   $ register R has three states
```

register R possesses three states defined by C[0;2]. C[0;2] can be decomposed into distinctive state variables as C[0], C[1], and C[2].

The transition of states from one to another, in any sequential logic network, are declared by the corresponding sequence declaration with a generalised syntax :

Sequence : transitions of states

Any sequence declaration at the functional level has the same format as those in the behavioral level. The sequence declarations at the functional level, however, can provide more precise information. The transition of states is based on the time-block concept, i.e.

condition ⟶ next state

The condition part is always a Boolean expression. Appropriate state variables may be taken as operands for the formation of

such a Boolean expression. The operator " " means that control will be transferred to the next state if the corresponding condition is true. For illustration, an example is provided below :

```
$ Definition of the operation of a 2-bit up counter
$ in terms of state transitions

Counter : CTR :: State : C[n/0;3]   $ assume the declarator
                                    $ " Counter " has been
                                    $ predefined

              :: Sequence :     C[0] ⟶ C[1] ,
                                C[1] ⟶ C[2] ,
                                C[2] ⟶ C[3] ,
                                C[3] ⟶ C[0] ,
```

It should be noted, however, sequence declarations may be presented mathematically. For example, the sequence declaration for CTR in the above construct, can be represented below :

```
Sequence :  (n<3)   C[n]   C[n+1] .,
            (n=3)   C[n]   C[0]
```

REPRESENTATION OF SYNCHRONOUS AND ASYNCHRONOUS OPERATIONS

The appropriate formulation of the " conditions " for state transitions can reflect the " synchronous " and " asynchronous " modes of operations. Such a reflection is achieved by default through the use of the indicator " ' ". The following two examples illustrate how to use the indicator " ' " to reflect

synchronous and asynchronous operations respectively at the functional level :

1. Example to illustrate the use of the indicator " ' " to reflect synchronous operation :

```
$ Definition of the synchronous state
$ transitions of a 2-bit up counter
$ CTR. The state transitions occur
$ at the leading edge of a timing
$ pulse K derived from a clock.
$ All the declarators used are
$ assumed to be already predefined.


        Counter : CTR :: State : C[0;3]

                      :: Sequence : 'K^C[0] ⟶ C[1] ,
                                    'K^C[1] ⟶ C[2] ,
                                    'K^C[2] ⟶ C[3] ,
                                    'K^C[3] ⟶ C[0] ,


 $ In the above construct, the synchronization of the
 $ flip-flops which have encoded the four states is
 $ illustrated.
```

2. Example to illustrate how the indicator " ' " can reflect the asynchronous coupling of two synchronous counters :

```
$ Definition of the operation of two 2-bit binary up counters
$ asynchronously coupled together as shown in Figure 3-2. The
$ transitions occur at the trailing edges of the corresponding
$ timing pulses. K is the timing pulse derived from a clock,
$ and all the declarators are assumed to have been predefined.

    Counter : CT1 :: State : C1[0;3]  $ counter no. 1

                  :: Sequence : K!^C1[0] ⟶ C1[1] ,
                                K'^C1[1] ⟶ C1[2] ,
                                K'^C1[2] ⟶ C1[3] ,
                                K'^C1[3] ⟶ C1[0] ,
```

```
: CT2 :: State : C2[0;3]  $ counter no. 2

      :: Sequence : C1[3]' ^C2[0]⟶C2[1] ,
                    C1[3]' ^C2[1]⟶C2[2] ,
                    C1[3]' ^C2[2]⟶C2[3] ,
                    C1[3]' ^C2[3]⟶C2[0] ,

$ The above example has shown that any state transition
$ in counter CT2 depends on the state transition of
$ C1[3] of the counter CT1.
```

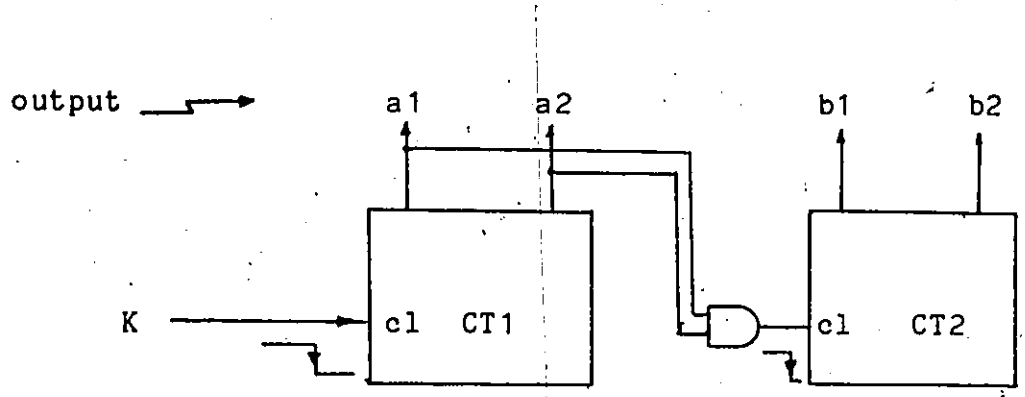


Figure 3-2 Asynchronous coupling of two synchronous 2-bit up counters named CT1 and CT2.

DIRECT REPRESENTATION OF SYNCHRONOUS AND ASYNCHRONOUS OPERATIONS

Though the indicator " ' " can be used indirectly to reflect " synchronous " and " asynchronous " operations, direct representation of such operations are made possible by appropriate declarations. The intent of direct representation is to make the synchronous and asynchronous properties of different components more evident.

The declarators " Synchronous " and " Asynchronous " are

self-explanatary. Since they describe only the properties of different entities, they are better used with other appropriate declarators to form " double declarators " rather than to be used alone. The syntax of double-declarators is

declarator & modifier (declarator)

Therefore the " double declaration " :

Register & Synchronous : AR<0;7>

means that the flipflops in the register AR are operating synchronously. The separator " & " is necessary for any double-declarator formation.

The declarators " Synchronous " and " Asynchronous " can be taken as modifiers because they may change some of the conventions adopted in AHDL. For example, the construct :

```
Counter : CT1 :: State : C1[0;3]
              :: Sequence : 'K^C1[0]⟶C1[1] ,
                            'K^C1[1]⟶C1[2] ,
                            'K^C1[2]⟶C1[3] ,
                            'K^C1[3]⟶C1[0] ,
```

has illustrated the operation of CT1 as synchronous because of the indicator " ' ". It says that at the " leading-edge transition " of K, the flip-flops encoding the states are triggered synchronously. Particularly, in any construct which has the format as in the above example, the convention of synchronous operation is always assumed.

If the declarator " Counter " in the above example is changed to " Counter & Synchronous ", the element " 'K " indicates only the transitional characteristic. The declarator " Synchronous " has already clearly declared the synchronous operation of the counter CT1.

If the declarator " Counter " for CT1 in the above example, however, is changed to " Counter & Asynchronous ", the element " 'K " no longer indicates synchronous operation. Instead, an asynchronous configuration of the counter CT1 is declared. Then, K represents the signal to drive the change of states of the asynchronous counter CT1, at its leading-edge transition.

The declarators " Synchronous " and " Asynchronous " are important to define the characteristics of different components within a digital system directly. In any design process, they express clearly what kind of functional elements are exactly required.

COUPLING

The declarator " Couple " is reserved to cope with unusual situations in any functional-level formulation, declaring the required coupling characteristics of different entities. The use of this declarator is illustrated in the example below :

```
$ Two " synchronous " counters CR1 and CR2 are
$ asynchronously coupled together to form the
$ counter CRR.
```

```
Clock : K

Counter & Asynchronous : CRR<0;3> # CR1<0;1> @ CR2<2;3>

                          :: State : C0[0;15] $ states
                                                of CRR

Subcounter : CR1<0:3> :: State : C1[0;3]
& Synchronous

                         :: Sequence :

                              'k^C1[0] ⟶ C1[1] ,
                              'k^C1[1] ⟶ C1[2] ,
                              'k^C1[2] ⟶ C1[3] ,
                              'k^C1[3] ⟶ C1[0] ,

                 : CR2<0;3> :: State : C2[0;3]

                         :: Sequence :

                              'q^C2[0] ⟶ C2[1] ,
                              'q^C2[1] ⟶ C2[2] ,
                              'q^C2[2] ⟶ C2[3] ,
                              'q^C2[3] ⟶ c2[0] ,

Couple : K/k   $ k and K define one another

         C1[3]/q   $ C1[3] and q define one another
```

In the above example, the coupling characteristic of " C1[3]/q " is equivalent to specifying the state transition for CR2, in the form :

'C1[3]^C2[n]⟶C2[n+1]

Coupling characteristics can exist in a variety of forms. There is no definite rule to exact the ways of specifying them. The only means to make them clear is through the corresponding " couple declaration " in the generalised syntax :

Couple : coupling characteristics

This declaration defines the I/O relationship of the logic functions which perform the coupling.

GENERIC DEFINITION OF COUNTERS

Since counters play a very significant part in digital systems, " generic " definitions of counters are included in the functional level in order to meet the generality . The declarator " Modulo " is to declare how many states which a counter has. The syntax of a "modulo declaration " is either

Modulo : state identifier [a;b % absent states]

or Modulo : state identifier [a;b]

The dimension of [a;b] represents any strict binary count sequence from a to b by default. For example, the modulo declaration :

Modulo : I[0;3]

is the functional-level representation of the symbolism illustrated by Figure 3-3.

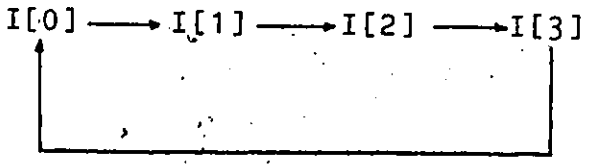


Figure 3-3 Symbolism of the count sequence from state I[0] to state I[3].

The separator " % " separates and indicates the absent states

from a strict count sequence from a to b. For example, the modulo declaration :

Modulo : I[0;5 % 2,3]

has the same meaning as the symbolism illustrated by Figure 3-4.

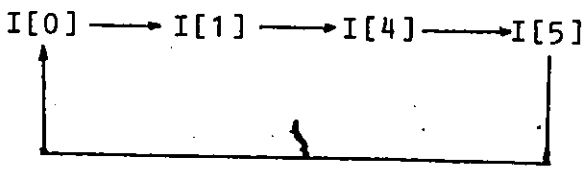


Figure 3-4 Symbolism of a count sequence of four states.

The use of modulo declaration saves the wordiness of writing down the state declaration and sequence declaration for any counter provided that the count sequence is continuous . For example, in the functional-level description of the counter CT1 in Figure 3-2, the whole construct can be simplified as

Counter : CT1 :: Modulo : C1[0;3]

Such a construct implies the same count sequence as previously worked out. If a 4-bit counter CY has a count sequence of the fourteen states sequentially listed :

```
0000P4  ;  $ P stands for binary
1000P4  ;
0100P4  ;
0010P4  ;
1010P4  ;
0110P4  ;
1110P4  ;
0001P4  ;
```

```
    1001P4  ;
    0101P4  ;
    0011P4  ;
    1011P4  ;
    0111P4  ;
    1111P4  ;
  -------------   $ change back to state " 0000P4 "
    0000P4  ;
```

the states of 1100P4 and 1101P4 are absent. The modulo declaration for the counter CY is

```
Counter : CY :: Modulo : CC[0;15 % 3,11]
```

The modulo declarations are compatible with the generic counter descriptions in the field. If the count sequence is not continuous, then the approaches of state and sequence declarations are required.

PROCEDURE CALL

Procedure calls in the functional level are based on the very same philosophy and rules as in the behavioral level. The section " PROCEDURE CALL " in Chapter Two explained already such basic philosophy and rules.

Procedure calls in any functional-level construct can call procedures constructed in terms of either the functional level itself, the behavioral level, or the structural level. The interaction of the functional level with the other two levels of AHDL, therefore, is always possible. Functional-level procedure calls, calling procedures constructed in terms of the structural

level, match programming hierarchy with the required information hierarchy . If they, however, call procedures constructed in terms of the behavioral level, they represent a form of information hiding . Since information hiding is defined as the process to avoid the unnecessary details with respect to a particular level of formulation, behavioral-level procedures called by functional-level procedure calls " hide " the undesirable information.

INTERACTION BETWEEN BEHAVIORAL AND FUNCTIONAL LEVELS

Theoretically, the procedural co-existence of any behavioral-level description and any functional-level description is possible, if the interaction between them is only through a procedure call .

If a functional-level procedure is called by a behavioral-level statement, then the programming hierarchy goes in parallel with the corresponding required information hierarchy . In the behavioral-level construct :

```
Register : A<0;23>
           R<0;23>
           C<0;23>

Boolean : K

Algorithm : K ⟵ C ⟵ A+R   $ add A to R, and the result
                            to C, if K is true
```

the Algorithm part can be converted into the following construct containing a procedure call :

```
Algorithm : K ⟶ call ADD <A,R,X> ;
                C ⟵ X
```

The procedure body of ADD, then can be written in the functional-level form :

```
$ Functional-level procedure of parallel addition

   Procedure : ADD <aa,rr,xx>
Register : aa<0;23> :: Terminal : a<0;23> $ output terminals
         : rr<0;23> :: Terminal : r<0;23> $ output terminals

Terminal : cry<23> :- 0   $ cry stands for " carry "
           cry<0~22> :- a<1~23>^r<1~23>vr<1~23>^cry<1~23>v
                        cry<1~23>^a<1~23>
           sum<0~23> :- a<0~23>⊕r<0~23>⊕cry<0~23>
```

In the above discussion of the procedure call " call ADD ", the idea of procedural co-existence is clearly shown. The only linkage between the behavioral-level formulation and the functional-level procedure is the set of parameters transmitted in the process of the procedure call.

The reversed process of a behavioral-level procedure to be called by a functional-level procedure call is also possible. Such kind of procedure calls obeys the criterion of information

hiding .

SUMMARY AND COMMENTS

The reserved declarators in the functional level are " Terminal ", " State ", " Subcounter ", " Modulo ", "Counter", " Clock ", and " Couple ". Any additional declarator can be predefined as a reserved declarators in the functional level, to the convenience of the users.

The operators particular to the functional level are listed in Table 3-1.

Table 3-1 The operators assumed particular to the functional level.

Operator	Explanation
:-	operator to associate a terminal variable to a Boolean expression
DO	operator to abstract an iterative mechanism

The separator " :: " separates any modular description from the corresponding identifier for the module, and the separator " % " indicates the absent states in any natural binary sequence . The separator " & " indicates any formation of " double declaration ".

The state variables and the terminal variables in the functional level can be taken as operands to be operated on by the appropriate operators.

The most prominent difference between the behavioral level and the functional level is the presence of the two abstract data types : terminal and state ,and the other linguistic elements in the latter but not in the former.

Procedural co-existence of the behavioral level and the functional level is provided by appropriate procedural calls. The latter defines clear boundaries among constructs structured in terms of both levels. The linkage among the procedurally co-existent constructs is through parameterization.

The criterion of information hiding, however, permits the co-existence of the different levels of AHDL in a different form, i.e. continuous co-existence. This kind of co-existence is one-dimensional or strictly hierarchical. For example, standard operators in the behavioral level may be used in functional-level constructs, without being defined in terms of the functional level. Nevertheless, the reverse operation of using functional-level elements in a behavioral-level construct is incorrect.

Procedural co-existence enhances the applicability of AHDL by allowing it to have a fluid hierarchy. Outside the use of procedures, however, the levels of AHDL are governed by the

continuous co-existence which imposes a strict hierarchy.

EXAMPLE

Two more examples for the constructs at the functional level are presented in the APPENDIX section. The only example attached here as an illustration is the functional-level description of the PIA (MOTOROLA MC6820).

```
$ The functional-level description of the PIA (MOTOROLA MC6820)
$ takes care of the behavioral-level characteristic. At the
$ same time it looks into the internal organisation of the PIA.
$ The basic working principles of the co-operative components
$ are examined. Terminals are identified, and a modular
$ approach has been adopted. This level, however, is
$ concerned with the input/output relationship of modules in a
$ system. It does not provide and support, however, detailed
$ component descriptions.

$ When the PIA is interfaced with a MPU, the operation
$ of the PIA is determined by a " software polling process "
$ in the MPU to check the status of the PIA.
$ This polling process plays an important part
$ in the HANDSHAKE configuration for the operation
$ of the data transfers through the PIA .
$ The result of the polling process is the generation of
$ a set of Boolean signals through the appropriate terminals
$ to the PIA. A simpler HANDSHAKE concept using the
$ PIA as a mediator was presented in in the behavioral-level
$ description of the PIA in Chapter Two.


        Bus : DD<0;7>.   $ between MPU and PIA (bidirectional)

              PA<0;7>    $ between peripheral devices
                           and section A
                           of the PIA (bidirectional)

              PB<0;7>    $ between peripheral devices
                           and section B
                           of the PIA (bidirectional)


    Register : CRA<0;7>  $ control register of section A
```

```
                DDA<0;7> :: Terminal :    $ data direction
                                            register
                                            of section A
                          DDAA<0;7>       $ output terminals
                                            of DDA

                ORA<0;7>                  $ output register
                                            of section A

                CRB<0;7>                  $ control register
                                            of section B

                DDB<0;7> :: Terminal :    $ data direction
                                            register
                                            of section B
                          DDBB<0;7>       $ output terminals
                                            of DDB

                ORB<0;7>                  $ output register
                                            of section B


Subbus : SD<0;5> # DD<0;5>                $ rename DD<0;5>
                                            as SD<0;5>

Subregister : SCRA<0;5> # CRA<0;5>        $ rename CRA<0;5>
                                            as SCRA<0;5>
                                            for clarity

              SCRB<0;5> # CRB<0;5>        $ rename CRB<0;5>
                                            as SCRB<0;5>
                                            for clarity

              CRA<7> :: Terminal : T1     $ output terminal

              CRB<7> :: Terminal : T2     $ output terminal

              CRA<2> :: Terminal : T3     $ output terminal

              CRB<2> :: Terminal : T4     $ output terminal

$ The required " terminal functions " of the PIA are defined
$ below in terms of some of the basic terminals of the
$ integrated circuit itself. These terminals and terminal
$ functions are listed sequentially for clarity.
```

```
Terminal : RS0    ;          $ RS0 is control terminal
                               to PIA

           RS1    ;          $ control terminal to PIA

           CS0    ;          $ control terminal to PIA

           CS1    ;          $ control terminal to PIA

           CS2    ;          $ control terminal to PIA

           SEL:- CS0^CS1^.CS2  ;  $ select PIA

           RW     ;          $ control read(RW)
                               and write(.RW)

$ The above seven terminals carry HANDSHAKE signals from
$ the MPU to the PIA after polling. They are the original
$ pins of the integrated circuit.

           E      ;       $ clock input terminal to PIA

           Reset  ;       $ clear all registers in PIA

           IRQA :- .T1  ; $ interrupt control terminal
                            from section A

           IRQB :- .T2  ; $ interrupt control terminal
                            from section B

$ The terminals IRQA and IRQB carry HANDSHAKE signals from
$ PIA to MPU through " interrupt ". They are original pins.

           A :- .RS1^.RS0^T3  ;   $ ORA enable

           B :- .RS1^.RS0^.T3  ;  $ DDA enable

           C :- .RS1^RS0      ;   $ CRA enable

           D :- RS1^.RS0^T4   ;   $ ORB enable

           F :- RS1^.RS0^.T4  ;   $ DDB enable

           G :- RS1^RS0       ;   $ CRB enable

           DA :-  DDAA<0;7> ;     $ ANDing all bits of DDA

           DB :-  DDBB<0;7> ;     $ ANDing all bits of DDB
```

```
$ The above eight terminals are catalytic variables.

        CA1                 ;   $ control from peripheral
                                  devices to PIA
                                  (section A)

        CA2 :- T1           ;   $ control to peripheral
                                  devices
                                  from PIA(section A)

        CB1                 ;   $ signal from peripheral
                                  devices to PIA
                                  (section B)

        CB2 :- T2           ;   $ control from PIA
                                  to peripheral
                                  devices(section B)

$ The terminals CA1, CA2, CB1, and CB2 are terminals which
$ carry HANDSHAKE signals. They are original pins.

$ In the construct below, " *  --  *." indicates the
$ " blocking mechanism ". The separators " , " and " ; "
$ denote parallel and sequential operations respectively.


   Algorithm : * Clear, Program, Data *  $ three possible
                                           operations


          Clear : Reset^SEL      * X1,X2,X3,X4,X5,X6 *
              X1 : ORA <-- 00000000P8 ,  $ P stands for
                                           binary
              X2 : DDA <-- 00000000P8 ,
              X3 : CRA <-- 00000000P8 ,
              X4 : ORB <-- 00000000P8 ,
              X5 : DDB <-- 00000000P8 ,
              X6 : CRB <-- 00000000P8 ,


        Program : * Read, Write *
```

```
Read : * SectionA, SectionB *

    SectionA : SEL^E^RW^.T1 ⟶ * Y1,Y2,Y3 *

        Y1 : A ⟶ DD ⟵ ORA , $ read ORA

        Y2 : B ⟶ DD ⟵ DDA , $ read DDA

        Y3 : C ⟶ DD ⟵ CRA , $ read CRA

    SectionB : SEL^E^RW^.T2 ⟶ * Y4,Y5,Y6 *

        Y4 : D ⟶ DD ⟵ ORB , $ read ORB

        Y5 : F ⟶ DD ⟵ DDB , $ read DDB

        Y6 : G ⟶ DD ⟵ CRB , $ read CRB


Write : * SecA, SecB *

    SecA : SEL^E^.T1^.RW ⟶ * Z1,Z2 *

        Z1 : B ⟶ DDA ⟵ DD , $ write into DDA

        Z2 : C ⟶ SCRA ⟵ SD , $ write into
                              first five bits
                              of CRA

    SecB : SEL^E^.T2^.RW ⟶ * Z3, Z4 *

        Z3 : F ⟶ DDB ⟵ DD , $ write into DDB

        Z4 : G ⟶ SCRB ⟵ SD , $ write into
                              first five
                              bits of CRB


Data : * Start ; I-O *

    Start : 'CA1 ⟶ CRA<7> ⟵ 1P1 ; $ set CRA<7>

            'CB1 ⟶ CRB<7> ⟵ 1P1 , $ set CRB<7>

    I-O   : * partA, partB *

partA : SEL^RW^E^T1^.T2^.DA ⟶ DD ⟵ PA ; $ read peripheral
                                         deices
```

```
                                    CRA<7> OP1 $ reset CRA<7>

$ The transitional mechanism which resets CRA<7> is not
$ shown because it has not been defined explicitly in the
$ manual for the PIA.

            SEL^.RW^E^T1^.T2^DA^A      * AAA;BBB *

                    AAA : ORA    DD ; $ write into ORA

                          PA     ORA  $ write into PA

                    BBB : CRA<7>    OP1

partB : SEL^RW^E^.T1^T2^.DB^  DD  PB ; $ read peripheral

                                    CRB<7>  OP1 $ reset CRB<7>

$ The transitional mechanism which resets CRB<7> is not
$ shown because it has not been defined explicitly in the
$ manual for the PIA.

            SEL^.RW^E^.T1^T2^DB^D      * CCC; DDD *

                    CCC : ORB    DD ; $ write into ORB

                          PA     ORB

                    DDD : CRB<7>    OP1

$ From the above functional level description, it is clear
$ that how terminals are defined and used efficiently for
$ various control purposes. The complete capabilities of
$ the PIA are presented in terms of various modular
$ components. Most of the characteristics of the AHDL
$ behavioral level have still been retained.

$ Since the exact operational behavior of the hardware in
$ the PIA has not been given in the manual for the PIA,
$ the functional-level description above serves only to
$ fully represent those information available.
```

CHAPTER FOUR

STRUCTURAL LEVEL OF AHDL

INTRODUCTION

The structural level of AHDL is the level of logic design in terms of gates and flip-flops. It defines the physical behavior of structural primitives with respect to discrete timing pulses. Design techniques and IC technologies employed in design processes can be clearly represented by appropriate constructs at this level. Normally, the structural level would only be used by engineers to specify the SSI logic circuitry required to adapt medium and large scale integrated circuits to a particular design requirement. It could also be used in the design of the integrated circuits themselves, as demonstrated in the example presented at the end of this chapter.

The structural level is an extension of the functional level with one additional abstract data type : " link ". Most of the syntactic characteristics in the functional level appears also in this level. The basic framework to express any control transfer is also based on the concept of a time block :

condition ——→ a single action or a block of single actions

The " condition " part is always a Boolean expression of ANDing all the pertinent control variables. These variables are the

identifiers of different single timing pulses. When a condition becomes true, it is also a single timing pulse.

Since the basic unit of control is a single timing pulse, all the single actions in a time block to be executed within the duration of the corresponding condition must always be concurrent . The one condition to one single action correspondence makes the structural level nonprocedural , i.e. by definition the lexicographical ordering of single actions has no meaning. Activation of a block of single actions by a condition may be done in one of the three ways :

1. At the trailing-edge transition.
2. At the leading-edge transition.
3. During the period when a condition is true.

Detailed description of modules is one of the major applications of the structural level. A modular description may be presented in terms of terminals and states (described in Chapter Three). States, however, can be defined in more detail in this level, through statement assignments, specification of flip-flop types and circuit equations.

The interconnection of different modules in a digital system can be clearly specified at the structural level. Modular interconnections are defined by wiring terminals in the forms of terminal variables. The wiring of the terminals is accomplished

by the operator " :-: ".

In accordance with the principle of information hiding, it is permissible to use behavioral-level and functional-level entities in any structural-level construct without first defining them in terms of the structural level. For example, standard behavioral-level operators may be used in a structural-level construct with presumed non-procedural behavior. Behavioral-level and functional-level entities used in this way are treated like black boxes. The input/output relationship of these black boxes is defined, but their internal structures are unspecified. This corresponds closely to the actual practice of circuit design with MSI and LSI, where the functions or behavior of the more complex modules are known to the designer, but not the internal structures. The designer uses his knowledge of gate-level logic design, where necessary, to connect and adapt the larger modules of standard functions. He does not, however, try to emulate them with the SSI components.

Procedure calls in the structural level are subject to the same rules as in the behavioral and functional levels. The incorporation of procedure calls in the structural level does not contradict with the non-procedural nature of the structural level. The reasons are :

1. Procedure calls have the character of a single statement.
2. The virtual execution time of a procedure call is restricted to a single clock period.

Procedure calls in the structural level ensure programming hierarchy. They also provide a means to interact the structural level with the other two levels of AHDL.

The structural level of AHDL is designed to fulfill the following objectives :

1. Representation of circuit designs with discrete logic elements, i.e. gates and flip-flops.
2. Explicit description of modular designs where appropriate .
3. Specification of logic for the interconnection of modules.
4. Defining the co-operation of different modules within a digital system with respect to discrete single timing pulses.

LINK

The " link " is the abstract data type exclusively defined in the structural level. A link corresponds to a common connection such as a wire or a data bus in a digital system. By itself, it has no implication of direction . Since links serve as interfaces between modular data paths, the directionality imposed by a module on the path may in turn impose directionality on the corresponding links connecting the module. In simple terms, for example, a link may be considered as a single strap to which a number of modular terminals are

connected. Links are expressed in the form of " link variables ". Link variables are validated only by the corresponding " link declarations " of the generalised syntax :

Link : link variables

" Link " is a reserved word in the structural level. Any link variable can be an unsubscripted or a subscripted identifier. Generally, link variables can be dimensioned and decomposed like registers. In the example :

Link : L1

L<0;7>

L1 is a single link, and there are eight links embedded in L<0;7>, namely L<0>, L<1>, L<2>, L<3>, L<4>, L<5>, L<6>, and L<7> . The application of link variables is shown in the structural-level description of the PIA (MOTOROLA MC6820) in the section EXAMPLE.

MODULAR INTERCONNECTION

Modules are wired together by the wiring operator " :-: ". The syntax of wiring of two terminals is the following :

terminal variable :-: terminal variable

Any wiring is declared by the declarator " Wiring ". In the example :

```
$ Terminal wiring explication

Wiring    : A :-: B
            C<0;11> :-: D<0;11>
```

the single terminals A and B are wired, and the twelve terminals specified by C<0;11> are wired to the twelve terminals specified by D<0;11>. i.e. C<0> to D<0>,C<1> to D<1>, and so on.

STATE ASSIGNMENT

" State assignment " is the process of assigning a binary code to a state of a sequential network. The books by Friedman (Friedman-25) and Torng (Torng-24) respectively give clear dicussions of the problem.

State assignment is a complex problem. Detailed description of the techniques involved for state assignment is out of the scope of the present work. The structural level of AHDL provides only a " state-association mechanism ", allowing the users to associate freely any binary combination in terms of bits to a state. Specifically, the functions are as follows :

1. Specification of the number of bits used to encode the required states.
2. Assignment of unique identifiers to each of these bits.
3. Definition of the binary code to represent a state, in terms of the identifiers assigned to the bits.

The state-assignment provision in the structural level represents a user-selected mapping of binary codes to states. It is not, however, intended to provide any algorithmic means of optimization.

For the inclusion of the necessary state assignment , a state declaration expressed in the syntax below is required :

State : state identifier[a;b]<state assignment>

The state assignment enclosed in the brackets " < > " is represented by a string of binary digits. For example, if the binary string " nP3 " is the formal state assignment for a particular sequential logic network, then n symbolises any possible bit pattern for a state. In the example :

state identifier[x]<nP3>

" state identifier [x] " designates a particular state, and "<nP3> " specifies the binary code for this state.

Every bit in n can be assigned a unique identifier through the application of the corresponding blocking mechanism . For example, the three bits of n may be represented by the corresponding identifiers positionally in the form :

n* A, B, C *P3

After the binary code for a state has been represented in terms of A, B and C, the qualifier "P3" becomes redund ant. The part of " P3 " can, therefore, be omitted. The following example illustrates. the concept of state assignment at the structural level.

```
$ Description of state assignment for a 8421-code counter
    Clock : K     $ activation at the leading-edge transition

    Counter : CT :: State : S[0;15]< n* A, B, C, D *P4 >

$ A, B, C, and D are bit identifiers of the counter CT.
$ These bits are assigned by the blocking mechanism " * --- * ".

   :: Sequence :

        'K^S[0] ---> S[1]<.A,.B,.C,D>  ,  $ n is "0001"
        'K^S[1] ---> S[2]<.A,.B,C,.D>  ,  $ n is "0010"
        'K^S[2] ---> S[3]<.A,.B,C,D>   ,  $ n is "0011"
        'K^S[3] ---> S[4]<.A,B,.C,.D>  ,  $ n is "0100"
                  ----------           ,
                    ------             ,
                      --               ,
        'K^S[14] ---> S[15]<A,B,C,D>   ,  $ n is "1111"
        'K^S[15] ---> S[0]<.A,.B,.C,.D>,  $ n is "0000"
```

In the above example, the representation in ".A,.B,.C,D" is the bit pattern of "0001", and so is ".A,.B,C,.D" for "0010", ".A,.B,C,D" for "0011", and so on. In an actual AHDL construct,

however, it would be necessary to specify all the sixteen assignments.

INPUT EQUATIONS TO FLIP-FLOPS

Flip-flops change state in response to an appropriate external excitation, usually on or near the transition of a timing pulse. There exist many different types of flip-flop designs, and correspondingly many different ways to respond to a given excitation. At the structural level, AHDL has the capacity to define such a response through the " equation declaration " of the syntax :

Equation : input equations

Equation declarations are functionally similar to terminal declarations. They are, however, applicable only to flip-flop descriptions. A description of the RS flip-flop, which is shown in Figure 4-1, is given below to illustrate this type of declaration :

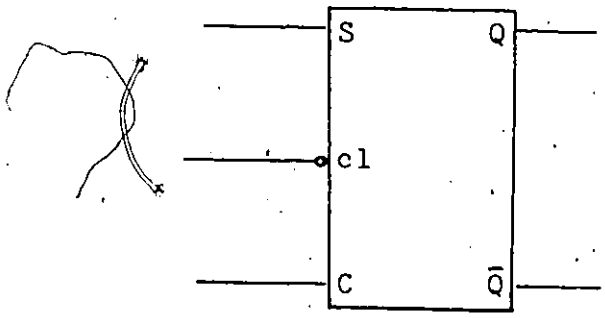


Figure 4-1 Symbolism of a RS flip-flop.

1. S and C are the input terminals to which external triggers are connected. These external triggers are expressed in Boolean expressions so that the input equations to the RS flip-flop are :

S :- Boolean expression 1

and C :- Boolean expression 2

2. The RS flip-flop changes state at the trailing-edge transition of the clock pulse on terminal " cl ". Change of state takes place in accordance with the truth table shown in Figure-4-2.

input		next state
Sn	Cn	Qn+1
0	0	Qn
0	1	0
1	0	1
1	1	Ambiguous

Figure 4-2 Truth table for a RS flip-flop.

In Figure 4-2 the subscripts " n " and " n+1 " differentiate

"before" and " after " of the " 1 to 0 " clock transition.

DESIGN AT THE STRUCTURAL LEVEL

If a 2-bit Gray-code counter as shown in Figure 4-3 is designed from two RS flip-flops described by both Figure 4-1 and Figure 4-2, the design contains the following information :

1. The block diagram of the 2-bit counter is shown in Figure 4-3.

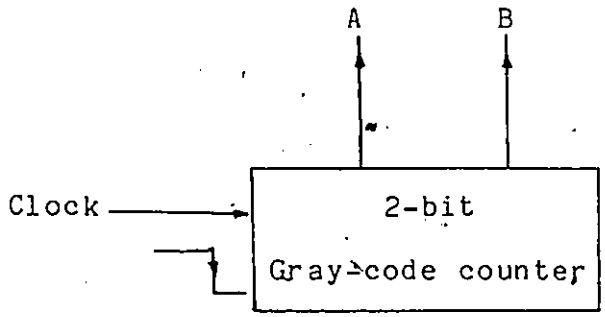


Figure 4-3 Block diagram of the 2-bit counter.

2. The count sequence of the 2-bit counter is shown in Figure 4-4. " A " represents the output of the first flip-flop, and

" B " represents the output of the second flip-flop.

A	B
0	0
0	1
1	1
1	0
0	0

Figure 4-4 Count sequence of the 2-bit Gray-code counter.

3. The input equations to the first flip-flop are :

$$Sa :- B$$

and

$$Ca :- .B"$$

4. The input equations to the second flip-flop are :

Sb :- .A

and Cb :- A

5. The completed design and the implementation for the 2-bit Gray-code counter is shown in Figure 4-5.

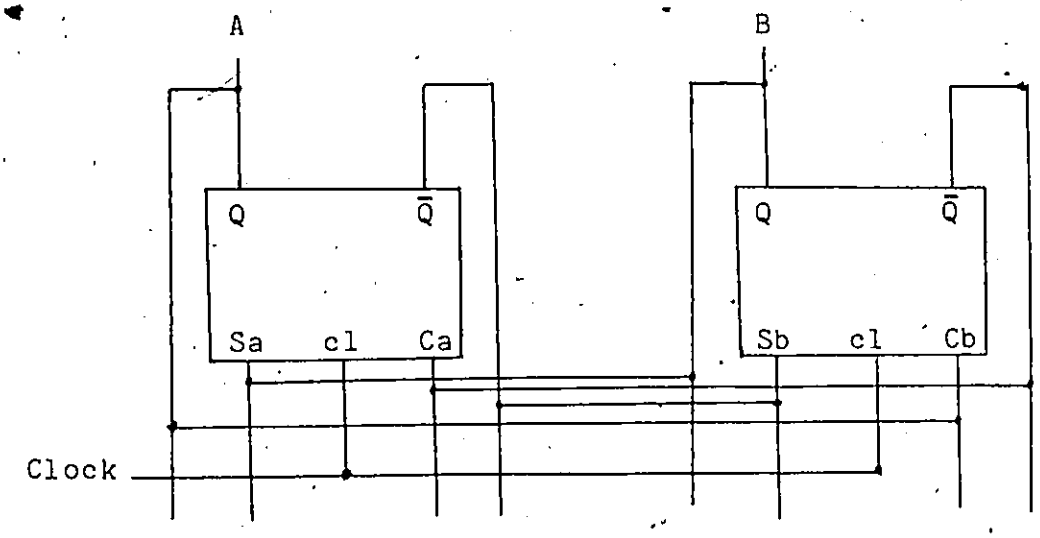


Figure 4-5 Completed design and the implementation of the 2-bit Gray-code counter.

The design process above for the 2-bit Gray-code counter comprises standard procedures of applying conventional design techniques and IC components at the SSI level. However, the whole design process can be expressed in the form of a structural-level construct in AHDL to the very same details.

Such a construct is presented as in the following :

```
$ Structural-level construct of the 2-bit Gray-code counter
$ as shown in Figure 4-5.

   Clock : K

   Counter : CT :: State : S[0;3]<n*A,B*P2>  $ P stands
& Synchronous                                  for binary


$ The  synchronous operation  of the counter CT has been
$ declared explicitly by the declarator " Synchronous ".
$ K' indicates that the trailing edge of the clock
$ synchronizes the transitions.

      :: Sequence : K'^S[0] ----> S[1]<.A,B>  ,
                    K'^S[1] ----> S[2]<A,B>   ,
                    K'^S[2] ----> S[3]<A,.B>  ,
                    K'^S[3] ----> S[0]<.A,.B> ,


      :: Terminal :
          A   $ output terminal of 1st flip-flop
          B   $ output terminal of 2nd flip-flop
          Sa  $ input terminal of 1st flip-flop
          Ca  $ input terminal of 1st flip-flop
          Sb  $ input terminal of 2nd flip-flop
          Cb  $ input terminal of 2nd flip-flop


      :: Equation :
                    Sa :- B^K'
                    Ca :- .B^K'
                    Sb :- .A^K'
                    Cb :- A^K'
```

In the equation declaration of the above structural-level construct, K' is ANDed to the corresponding variables to make the input equations more informative. In any actual practice, such an ANDing is not strictly necessary .

OPERATOR AND DECLARATOR

The operator which is particular to the structural level is the wiring operator " :-: ". The reserved declarators which are particular to the structural level are " Link ", " Wiring ", and " Equation ".

PROCEDURE CALL

The basic structures and rules for the procedure calls in the structural level are the same as in the behavioral and functional levels. Other than ensuring programming hierarchy , procedure calls in the structural level interact this level with the other two levels of AHDL in the form of procedural co-existence , in accordance with the principle of information hiding .

SUMMARY AND COMMENT

Basically, structural-level constructs are formulated to accommodate details for the subsequent implementations. However AHDL, in accordance with the principle of information hiding, permits the structural level to interact with the other two levels through either procedural or continuous co-existence. Structural-level constructs, however, must be non-procedural .

A structural-level construct may, however, be completely defined in terms of terminals, and input equations to flip-flops. The construct then appears like a wiring list, specifying how different terminals to be wired .

EXAMPLE

One more example for the structural level is presented in the APPENDIX section. The one provided here as an illustration is the structural-level description of the PIA (MOTOROLA MC6820).

```
$ Structural-level description for the PIA (MOTOROLA MC6820).

$ The structural-level description of the PIA looks into the
$ gates and flip-flops of the PIA. The different modules are
$ clearly expressed and described. Since the exact working
$ principles of the gates and flip-flops in the different
$ modules were not presented in the manual for the PIA,
$ it was assumed that the registers in the PIA were all
$ structured from flip-flops of the RS type. In addition,
$ parallel operations of these registers have been assumed.
$ The specification of a LSI module in such a manner
$ would normally only be done in the engineering
$ department of the manufacturer. This example serves
```

```
$ to compare the examples of the PIA description in the
$ previous two chapters.


     Clock : E          $ system clock

Bus : DD-IN<0;7>        $ from MPU to PIA

      DD-OUT<0;7>       $ from PIA to MPU

      PA-IN<0;7>        $ from peripheral devices
                          to PIA

      PA-OUT<0;7>       $ from PIA to peripheral
                          devices

      PB-IN<0;7>        $ from peripheral devices
                          to PIA

      PB-OUT<0;7>       $ from PIA to peripheral
                          devices

Link : L-IN<0;7>        $ internal bus to register
                          within the PIA

       L-OUT<0;7>       $ internal bus where
                          register drain
                          within the PIA


Path : DD-I :-: L-IN :-: PA-OUT     $ first possible
                                      data path

       DD-IN :-: L-IN :-: PB-OUT    $ second possible
                                      data path

       DD-O :-: L-OUT :-: PA-IN     $ third possible
                                      data path

       DD-OUT :-: L-OUT :-: PB-IN   $ fourth possible
                                      data path

     Register : CRA<0;7> :: Terminal :
& Synchronous

 S1<0;7>       $ set input to flip-flop

 C1<0;7>       $ clear input to flip-flop
```

```
Q1<0;7>      $ output terminals of CRA
K<0;7>       $ clock input to flip-flops
RET<0;7>     $ reset terminals to flip-flops

                :: Operation :

'K^S1<n>^.C1<n>──►CRA<n>◄──1P1  ,  $ set flip-flops
'K^.S1<n>^C1<n>──►CRA<n>◄──0P1  ,  $ reset flip-flops
.RET ──────►CRA◄──00000000P8  ,  $ clear flip-flops

          : DDA<0;7> :: Terminal :

 S2<0;7>      $ set input to flip-flops
 C2<0;7>      $ reset input to flip-flops
 Q2<0;7>      $ output terminals of DDA
 K<0;7>       $ clock input to flip-flops
 RET<0;7>     $ clear terminals to flip-flops

                :: Operation :

'K^S2<n>^.C2<n>──►DDA<n>◄──1P1  ,  $ set flip-flops
'K^.S2<n>^C2<n>──►DDA<n>◄──0P1  ,  $ reset flip-flops
.RET ──────►DDA◄──00000000P8  , $ clear flip-flops

          : ORA<0;7> :: Terminal :

S3<0;7>      $ set input to flip-flops
C3<0;7>      $ reset input ot flip-flops
Q3<0;7>      $ output terminals of ORA
K<0;7>       $ clock input to flip-flops
```

RET<0;7> $ clear terminals to flip-flops

:: Operation :

'K^S3<n>^.C3<n>⟶ORA<n>⟵1P1 , $ set flip-flops

'K^.S3<n>^C3<n>⟶ORA<n>⟵0P1 , $ reset flip-flops

.RET ⟶ ORA ⟵ 00000000P8 , $ clear flip-flops

: CRB<0;7> :: Terminal :

S4<0;7> $ set input to flip-flops

C4<0;7> $ reset input to flip-flops

Q4<0;7> $ output terminals ofCRB

K<0;7> $ clock input to flip-flops

RET<0;7> $ clear terminals to flip-flops

:: Operation :

'K^S4<n>^.C4<n>⟶CRB<n>⟵1P1 , $ set flip-flops

'K^.S4<n>^C4<n>⟶CRB<n>⟵0P1 , $ reset flip-flops

.RET ⟶ CRB ⟵ 00000000P8 , $ clear flip-flops

: DDB<0;7> :: Terminal :

S5<0;7> $ set input to flip-flops

C5<0;7> $ reset input to flip-flops

Q5<0;7> $ output terminals of DDB

K<0;7> $ clock input to flip-flops

RET<0;7> $ clear terminals to flip-flops

:: Operation :

```
'K^S5<n>^.C5<n> ⟶ DDB<n> ⟵ 1P1   , $ set flip-flops
'K^.S5<n>^C5<n> ⟶ DDB<n> ⟵ 0P1   , $ reset flip-flops
.RET ⟶ DDB ⟵ 00000000P8 , $ clear flip-flops
```

: ORB<0;7> :: Terminal :

```
S6<0;7>      $ set input to flip-flops
C6<0;7>      $ reset input to flip-flops
Q6<0;7>      $ output terminals of ORB
K<0;7>       $ clock input to flip-flops
RET<0;7>     $ clear terminals to flip-flops
```

:: Operation :

```
'K^S6<n>^.C6<n> ⟶ ORB<n> ⟵ 1P1   ,  $ set flip-flops
'K^.S6<n>^C6<n> ⟶ ORB<n> ⟵ 0P1   ,  $ reset flip-flops
.RET ⟶ ORB ⟵ 00000000P8   , $ clear flip-flops
```

```
: Subregister : SCRA<0;5> # CRA<0;5>
                SCRB<0;5> # CRB<0;5>
                CRA<7> :: Terminal : T1  $ output terminal
                CRB<7> :: Terminal : T2  $ output terminal
                CRA<2> :: Terminal : T3  $ output terminal
                CRB<2> :: Terminal : T4  $ output terminal
```

```
: Terminal :          $ the terminals are sequentially
                        defined here for clarity

        RS0  ;

        RS1  ;

        CS0  ;

        CS1  ;

        CS2  ;

        E    ; $ clock output terminal

        K :-: E  ;

        RW   ; $ read(RW) and write(.RW) terminal

        RESET ; $ terminal to clear flip-flops

        RET :-: RESET  ;

        IRQA :- .T1  ;  $ interrupt from section A
                          of the PIA

        IRQB :- .T2  ;  $ interrupt from section B
                          of the PIA

$ The above terminals are the original pins of the IC(PIA).

        A :- .RS1^.RS0^T3  ;   $ ORA enable

        B :- .RS1^.RS0^.T3  ;   $ DDA enable

        C :- .RS1^RS0    ;    $ CRA enable

        D :- RS1^.RS0^T4  ;   $ ORB enable

        F :- RS1^.RS0^.T4  ;   $ DDB enable

        G :- RS1^RS0     ;   $ CRB enable

        DA :- ^Q2<0;7>   ;   $ ANDing all bits of DDA

        DB :- ^Q5<0;7>   ;   $ ANDing all bits of DDB

$ The above terminals are catalytic variables.
```

```
        CA1          ;        $ from peripheral devices
                                to the PIA

        CA2 :- T1    ;        $ from PIA to peripheral
                                devices

        CB1          ;        $ from peripheral devices
                                to the PIA

        CB2 :- T2    ;        $ from PIA to peripheral
                                devices

        SEL :- CS0^CS1^.CS2  ;  $ select PIA

$ The above five terminals are original pins of the IC(PIA).


  $ Identifications of the input terminals to flip-flops
  $ without using any  equation declaration .

S1<0;5> :- L-IN<0;5>^SEL^.RW^T1^C  ;

C1<0;5> :- .S1<0;5>  ;

S2<0;7> :- L-IN<0;7>^SEL^.RW^.T1^B  ;

C2<0;7> :- S2<0;7>  ;

S3<0;7> :- L-IN<0;7>^SEL^.RW^T1^.T2^DA^A  ;

C3<0;7> :- .S3<0;7>  ;

S4<0;5> :- L-IN<0;5>^SEL^.RW^.T2^G  ;

C4<0;5> :- .S4<0;5>  ;

S5<0;7> :- L-IN<0;7>^SEL^.T2^.RW^F  ;

C5<0;7> :- .S5<0;7>  ;

S6<0;7> :- L-IN<0;7>^SEL^.RW^.T1^T2^DB^A  ;

C6<0;7> :- .S6<0;7>  ;


$ Change of state owning to external request from
$ peripheral devices.
```

```
        S1<7> :- CA1  ;
        C1<7> :- .CA1  ;
        S4<7> :- CB1  ;
        C4<7> :- .CB1  ;

$ terminals with respect to " read data into MPU "

        O[0]<0;7> :- Q1<0;7>^SEL^RW^K^.T1^C  ;
        O[1]<0;7> :- Q2<0;7>^SEL^RW^K^.T1^B  ;
        O[2]<0;7> :- Q3<0;7>^SEL^RW^K^.T1^A  ;
        O[3]<0;7> :- Q4<0;7>^SEL^RW^K^.T2^G  ;
        O[4]<0;7> :- Q5<0;7>^SEL^RW^K^.T2^F  ;
        O[5]<0;7> :- Q6<0;7>^SEL^RW^K^.T2^D  ;
        O[6]<0;7> :- PA<0;7>^SEL^RW^K^T1^.T2^.DA  ;
        O[7]<0;7> : PB<0;7>^SEL^RW^K^.T1^T2^.DB  ;

    Couple : O[0;7]<0;7> :-: L-OUT<0;7> $ WIRE-OR coupling

$ It is clear from the structural-level description
$ for the PIA above how different modules are
$ interconnected through the corresponding terminals.
$ Furthermore, the interaction of the different
$ modular components have been well-defined by
$ the appropiate Boolean functions.
```

CHAPTER FIVE

CLOSING REMARKS

THE LANGUAGE AHDL

The objective of this research was to propose a consensus, multi-level language suitable for the description and simulation of computing hardware structures. AHDL (Algorithmic Hardware Design Language) is the textual language which has been developed and presented in this thesis.

AHDL consists of three separate and distinctive levels of application. They are the behavioral level, the functional level, and the structural level. The three levels of AHDL are concisely described in the following :

1. The behavioral level supports algorithmic formulations at the top or schematic level.
2. The functional level defines explicitly the principle of any mechanism which contributes to the operation of a digital system. The definition is expressed in terms of the input/output relationship of the system components.
3. The structural level is the level of logic design in terms of gates and flip-flops. This level would normally be used only by engineers.

The three levels of AHDL can be applied separately and distinctively for different digital purposes. They can, however, co-exist in two different forms : procedural co-existence, and continuous co-existence, in accordance with the principle of information hiding. Procedural co-existence refers to the structuring of procedures in terms of one level to be called by procedure calls constructed in terms of another level. Such co-existence permits the three levels of AHDL to interact in the sense of a fluid hierarchy. Continuous co-existence, however, represents only the application of the elements of a higher level in a lower-level construct. This co-existence is strictly hierarchical, and the reverse operation is incorrect.

The inclusion of procedure calls in AHDL gives its application a further dimension because they provide AHDL a potential mechanism for interaction with other languages. It is possible that a procedure, which is constructed in terms of a " non-AHDL " language, may be called by an AHDL procedure call, or vice versa. Since this feature represents a question of the further development of AHDL, the rules concerned are not discussed here.

COMPARISON WITH EXISTING RT LANGUAGES

The capabilities of AHDL, compared favourably to existing RT languagea, are listed below :

1. The three levels of AHDL are consensus.
2. The three levels of AHDL are compatible to any top-down and

multi-level digital task.

3. Procedural co-existence of the three levels permits the interaction of them in the form of a fluid hierarchy, in accordance with the principle of information hiding.
4. Procedures in AHDL can easily accommodate new MSI and LSI modules as they become available.

THE DEVELOPMENT OF AHDL

The textual structure for AHDL was adopted for five reasons as presented in the following :

1. Experiences from textual conventional programming languages have concluded that a textual structure is better able to express subtleties.
2. The well-proven control structures of the textual conventional programming languages can be adopted.
3. The techniques of compilation for the textual conventional programming languages are potentially modifiable for compiling AHDL.
4. Most of the powerful RT languages presently in use are textual.
5. A high degree of extensibility can be incorporated easily into the textual language statements. Such extensibility is required to generate a multi-level linguistic structure for AHDL.

AHDL was basically tuned from different textual conventional programming and RT languages. The inclusion of linguistic elements for expressing parallelism and concurrency was based on the abstractive power and familiarity of the elements. The basic framework of AHDL is structured from the seven basic abstract data types : register, variable, declarator, operator, terminal, state, and link. The structure of AHDL was thoroughly tested by different examples worked out on each of the three levels. Some of these examples are attached in the APPENDIX section, while others are presented in the appropriate places throughout the text.

EVALUATION OF AHDL

AHDL is potentially applicable to the description, design, and simulation of digital systems. The general evaluation of AHDL is the following :

1. Its textual similarities to conventional programming languages make it easy to be learned and compiled.
2. The simplicity and the expressive power of its linguistic elements make it suitable for clear specification or description of digital systems.
3. Its hierarchical nature corresponds with the hierarchical nature of any digital system.
4. Special structures and techniques can accommodate the application of standard design techniques and IC technologies.

THE POTENTIAL OF AHDL

Based on the qualities of AHDL evaluated above, the further development of this language could include the following :

1. Certain parts of AHDL could be modified to make it more informative and constructive.
2. The framework of AHDL could be tuned for computerization.
3. AHDL could be compiled and used like an ordinary programming languages.
4. It could be adopted as a suitable language to drive any design automation system.
5. It could be interacted with conventional programming languages to support the formulation of algorithms for parallel processing.

APPENDIX

- APPENDICES -

BEHAVIORAL-LEVEL EXAMPLES

Example 1 :

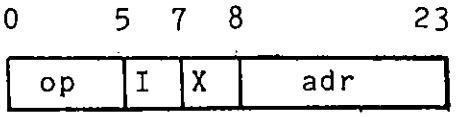
```
$ Design of a digital system which counts from "0" to "255"
$ in terms of the behavioral level of AHDL. This is the
$ schematic formulation of the algorithm for the system.

    Clock : K

    Register : C1<0;3>

               C2<0;3>

    Operator : display     $ abstract operator

    Algorithm : * Counting , Displaying * $ application of the
                                            blocking mechanism

      Counting : K'──→C1←── count-up C1 ,

                 (C1=15)'──→C2←── count-up C2

      Displaying : display C1  ,

                   display C2


$ The operator " display " is a dummy operator
$ introduced only for descriptive purpose and
$ clarity. " count-up ", however, is a standard
$ operator in the behavioral level of AHDL.
$ For more information about the system, the
$ respective functional-level construct will be
$ presented as Example 1. in the next section.
```

Example 2 :

Object : Design of a stored program computer at the schematic level in terms of the behavioral level of AHDL.

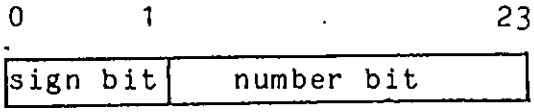
Specification :

1. The instruction and the number formats are :



(instruction format)

and



(number format)

The symbols "op" and "adr" represent respectively the op-code and the address part.

2. The instruction set is shown in Table-BL 1.

Instruction	Symbolic code	Opcode
addition	ADD m	00
subtraction	SUB m	01
jump on plus	JOP m	02
store	STO m	03
jump	JMP m	04
shift right	SHR	05
circular left shift	CIL	06
clear and add	CLA m	07
no operation	NP	10

Table BL-1 The instruction set.

The character "m" denotes a memory address. It means that

the symbol m represents an instruction or operand address in the the address field of the instruction preceding m. The opcode is octal. The explanation of the instructions are the following :

a. ADD and SUB adds and subtracts respectively the numerical value in address m to or from that in the accumulator.
b. CLA clears the accumulator before addition.
c. STO stores the content of the accumulator into the memory location specified by m.
d. JMP takes the next instruction from the memory location specified by m.
e. JOP performs as JMP, except only when the content of the accumulator is positive.
f. SHR shifts the accumulator one bit to the right, but the sign bit extended and the least significant bit dropped off.

g. CIL shifts the accumulator one bit to the left, but the most significant bit enters into the position of the least significant bit for every bit shifted.
h. NP stops the computer operation.

3. The stored program computer has six registers : R (buffer

register), A (accumulator), T (instruction register), P (program counter), C (address register), and F (control register). When the computer starts, the content of P is transferred to C, and then the content of P is counted up by one. The content of the memory location specified by the content of C is then transferred to the register T. The register T can be decomposed into two separate parts : op-code and address. The op-code is always transferred to the control register F to be decoded. If the I and X fields of T is assumed ineffective, then the address part is the "m" as shown in Table BL-1. The operand in the memory location at the address m will be fetched and loaded into R. The contents of R and A must be operated on by appropriate operators.

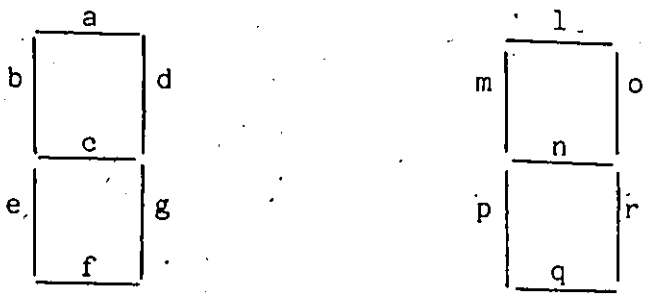
```
$ Behavioral-level construct for the stored program computer.

  Register : R<0;23>    $ buffer register
             A<0;23>    $ accumulator
             T<0;23>    $ instruction register
             P<0;14>    $ program counter
             C<0;14>    $ address register
             F<0;5>     $ control register

  Subregister : T<op/0~5>    $ op-code of T
                T<adr/9~23>  $ address part of T
                T<I/6>       $ indexing bit of T
                T<X/7~8>     $ indirecting addressing
                               bit of T

  Memory : M[0;32767]<0;23>

  Algorithm : * Fetch;Execution;Fetch * $ application of
                                          blocking mechanism

$ It should be noted that the cyclical relationship of
$ Fetch and Execution within * * is clearly indicated.
```

```
Fetch : C←—P ;  $ content of P to register C
        P←— count-up P ; $ increase P by one
        T←— M[C]  ;  $ instruction to instruction
                       register
        F←—T<op>     $ op-code of T to F

Execution : * Decoding,Activation *
    Decoding : K←—decode F  $ the value of K is octal
  Activation : switch K := * ADD,SUB,JOP,STO,JMP,SHR,
                             CIL,CLA,NP *
      ADD : R←— M[T<adr>] ;      $ addition
            A←— A+R
      SUB : R←— M[T<adr>] ;      $ subtraction
            A←— A-R
      JOP : (A<0>>=0)—→P ←— T<adr>  $ jump on positive
                                    accumulator
      STO : M[T<adr>]←—A         $ store accumulator
      JMP : P←—T<adr>            $ unconditional jump
      SHR : shift A              $ right shift
      CIL : rotate A             $ left circular-shift
      CLA : * AA;BB *
          AA : R←—M[T<adr>] , $ AA is a block of two
                              parallel data
                              operations
               A←—0
          BB : A←—A+R
      NP :                       $ pause of no operation

$ The above behavioral-level construct has shown the
$ algorithms of the stored program computer. For the
$ functional-level construct of this computer, refer to
$ Example 2. in the next section.
```

FUNCTIONAL-LEVEL EXAMPLES

Example 1 :

$ Design of a digital system which counts cyclically from
$ "0" to "255". The result of the count will be displayed
$ by two 7-segment modules. The two 7-segment modules are
$ shown in Figure FL-1.



(1) M1 (2) M2

Figure FL-1 The two 7-segment modules.

$ The counting is carried out by two 8421-code synchronous
$ counters coupled together as shown in Figure FL-2.

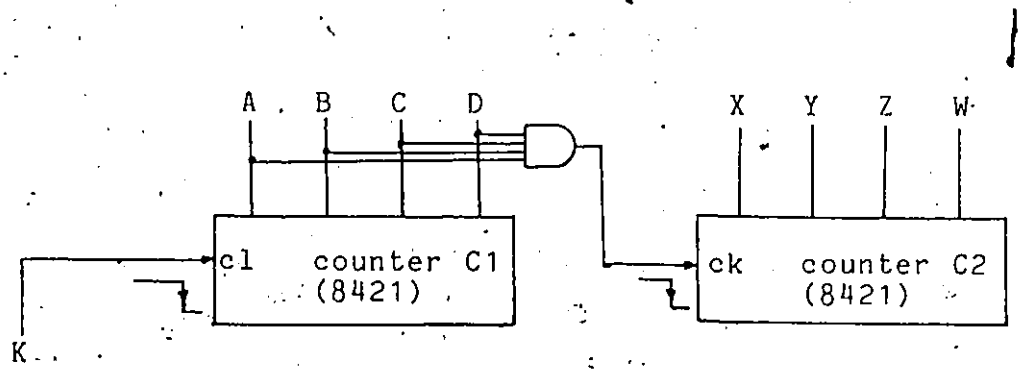


Figure FL-2 Two 8421-code synchronous counters coupled together.

```
$ Therefore the components necessary for such a system are :
$ two 8421-code synchronous counters, two 7-segment modules,
$ and a clock module.

$ The functional-level construct for the design is
$ presented below.

Clock : K              $ clock to drive the system

Counter & Synchronous :

    C1 :: Modulo : S1[n/0;15]    $ 16 sequential states

       :: Terminal : c1 :- K     $ c1 is clock input to C1

                     <A,B,C,D>   $ output terminals of C1
                                 $ and the order of the
                                 $ terminals is indexed
                                 $ from left to right
                                 $ in the brackets < >

       :: Operation : c1'^S1[n] ⟶ S1[n+1] , $ counting
                                              algorithm
                      c1'^S1[15] ⟶ S1[0]


    C2 :: Modulo : S2[n/0;15]    $ 16 sequential states

       :: Terminal : ck :- A^B^C^D  $ ck is clock input to C2

                     <X,Y,Z,,W>     $ output terminals of C2

       :: Operation : ck'^S2[n] ⟶ S2[n+1] , $ counting
                                              algorithm
                      ck'^S2[15] ⟶ S2[0]

7-segment : M1 :: Terminal :

   a :- .A^C v .A^B^D v A^.B^.C v .B^.C^.D
   b :- .A^.C v .B^.C v .A^B^.D
   c :- .A^B^.C v A^.B^.C v .A^.B^C v .A^C^.D
   d :- .A^.C^.D v .B^C^.D v .A^C^D v .A^.B^C
   e :- .B^.C v .A^C^.D
   f :- .A^B^.C^D v .B^.C^.D v A^.B^.C v .A^.B^C v .A^C^.D
   g :- .A^B v .A^C^D v A^.B^.C v .B^.C^.D
```

```
7-segment : M2 :: Terminal :

   l :- .X^Z v .X^Y^W v X^.Y^.Z v .Y^.Z^.W
   m :- .X^.Z v .Y^.Z v .X^Y^.W
   n :- .X^Y^.Z v X^.Y^.Z v .X^.Y^Z v .X^Z^.W
   o :- .X^.Z^.W v .Y^Z^.W v .X^Z^W v .X^.Y^Z
   p :- .Y^.Z v .X^Z^.W
   q :- .X^Y^.Z^W v .Y^.Z^.W v X^.Y^.Z v .X^.Y^Z v .X^Z^.W
   r :- .X^Y v .X^Z^W v X^.Y^.Z v .Y^.Z^.W
```

```
$ The structural-level construct for this design will be
$ presented in Example 1 of the next section. In this
$ example, though the counters are defined as generic
$ types, the terminal connections are specified explicitly.
```

Example 2 :

```
  $ The functional-level construct of the stored program
  $ computer described in Example 2. of the previous section
  $ is presented below.  Only the significant parts
  $ of the behavioral-level construct are extended. The
  $ specifications and working principles of the system
  $ remains the same. The application of procedure-calls
  $ is illustrated.


Register : R<0;23> :: Terminal : RT<0;23>  $ RT<0;23> are the
                                             output terminals
                                             of register R

         : A<0;23> :: Terminal : AT<0;23>  $ AT<0;23> are the
                                             output terminals
                                             of register A

           : T<0;23>          $ instruction register

           : P<0;14>          $ program counter

           : C<0;14>          $ address register

           : F<0;5>           $ control register


  Subregister : T<op/0;5>     $ op-code part of T

                T<adr/9;23>.  $ address part of T
```

```
                T<I/6>          $ indexing bit of T

                T<X/7~8>        $ indirect addressing
                                  bit of T


Memory : M[0;32767]<0;23>

Algorithm : * Fetch;Execution;Fetch *   $ note the application
                                          of the blocking
                                          mechanism


$ It should be noted that the cyclical relationship
$ of Fetch and Execution within the bractets * *
$ is clearly indicated.

  Fetch : C←—P ;          $ content  of P to C

          P←—count-up P ;  $ increase the content of P by one

          T←—M[C] ;        $ instruction to instruction register

          F←—T<op> ;       $ op-code of T to  F

  Execution : * Decoding,Activation *

     Decoding : K←—decode F     $ the value of K is octal

     Activation : switch K := * ADD,SUB,JOP,STO,JMP,SHR,CIL,
                                CLA,NP *


   ADD : R←—M[T<adr>] ;

         call Addition (RT,AT,SUM) ; $ procedure-call, and SUM
                                       is a catalytic variable
         A←—SUM

   SUB : R←—M[T<adr>] ;

         call Subtraction (RT,AT,DIF) ; $ DIF is a catalytic
                                          variable
         A←—DIF

   JOP : (A<0>>=0)——→P←—T<adr>    $ jump on positive
                                    accumulator

   STO : M[T<adr>]←—A             $ store accumulator

   JMP : P←—T<adr>                $ unconditional jump
```

```
SHR : A<n+1> ←— A<n>          $ right shift, and A<0>
                                is the most significant
                                bit

CIL : A<n> ←— A<n+1> ,        $ circular left shift

      A<23> ←— A<0>

CLA : * AA;BB *   $ application of blocking mechanism

     AA : R ←— M[T<adr>] , $ AA is a block of two parallel
                             data operations
          A ←— 0

     BB : call Addition (RT,AT,SUM) ;

          A ←— SUM

NP :                          $ pause of no operation


Procedure : Addition ( x,y,z )

          Terminal : x<0;23>

                     y<0;23>

                     z<0;23>


          Operator : + : Terminal :  $ parallel addition

              C<23> :- 0

              C<0~22> :- x<1~23>^y<1~23> v x<1~23>^C<1~23>
                         v y<1~23>^C<1~23>

              z<0~23> :- y<0~23> ⊕ x<0~23> ⊕ C<0~23>

$ The symbol C represents the carry generated by the
$ corresponding circuit.


Procedure : Subtraction ( p,q,r )

          Terminal : p<0;23>

                     .q<0;23>   $ complemented q<0;23>

                     r<0;23>
```

```
Operator : - :: Terminal :  $ parallel subtraction

    CC<23> :- CC<0>  $ feedback loop

  CC<0~22> :- p<1~23>^q<1~23> v p<1~23>^CC<1~23>
              v q<1~23>^CC<1~23>

   r<0~23> :- p<0~23> ⊕ q<0~23> ⊕ CC<0~23>

$ The symbol CC represents the carry generated by the
$ corresponding circuit.
```

STRUCTURAL-LEVEL EXAMPLE

Example :

```
$ Design of the digital system which was described in
$ Example 1. of the last section : "FUNCTIONAL-LEVEL
$ EXAMPLES", in terms of the structural level of AHDL.
$ The system counts cyclically from "0" to "255", and
$ the result will be displayed by two 7-segment modules.
$ For reference of the two 7-segment modules and the
$ coupling configuration of the two 8421-code synchronous
$ counters, see Figure FL-1 and Figure FL-2 respectively.
$ Since the functional-level construct of the
$ system already defined the working principles of
$ the system and the components involved, the construct
$ at the structural level concentrates mainly on the
$ design of the two 8421-code synchronous counter using
$ RS flip-flops.

Clock : K          $ clock to drive the system

Counter & Synchronous :

     C1 :: Modulo : S1[0;15]     $ 16 sequential states

        :: Terminal : cl :- K  $ cl is clock input to C1
                      A        $ output terminal of 1st FF
                      B        $ output terminal of 2nd FF
                      C        $ output terminal of 3rd FF
                      D        $ output terminal of 4th FF

        :: Equation : Sa :- .A $ Sa is set-input to 1st FF
                      Ca :- A  $ Ca is clear-input to 1st FF
                      Sb :- A^.B $ set-input to 2nd FF
```

```
                    Cb :- A^B  $ clear-input to 2nd FF
                    Sc :- A^B^.C $ set-input to 3rd FF
                    Cc :- A^B^C  $ clear-input to 3rd FF
                    Sd :- A^B^C^.D $ set-input to 4th FF
                    Cd :- A^B^C^D  $ clear-input to 4th FF

: C2 :: Modulo : S2[0;15]      $ 16 sequential states

    :: Terminal : ck :- A^B^C^D $ ck is clock input
                                    to 1st FF
                  X       $ output terminal of 1st FF
                  Y       $ output terminal of 2nd FF
                  Z       $ output terminal of 3rd FF
                  W       $ output terminal of 4th FF

    :: Equation :

       Sx :- .X        $ set-input to 1st FF
       Cx :- X         $ clear-input to 1st FF
       Sy :- X^.Y      $ set-input to 1st FF
       Cy :- X^Y       $ clear-input to 2nd FF
       Sz :- X^Y^.Z    $ set-input to 3rd FF
       Cz :- X^Y^Z     $ clear-input to 3rd FF
       Sw :- X^Y^Z^.W  $ set-input to 4th FF
       Cw :- X^Y^Z^W   $ clear input to 4th FF


7-segment : M1 :: Terminal     $ 1st 7-segment module

   a :- .A^C v .A^B^D v A^.B^.C v .B^.C^.D
   b :- .A^.C v .B^.C v .A^B^.D
   c :- .A^B^.C v A^.B^.C v .A^.B^C v .A^C^.D
   d :- .A^.C^.D v .B^C^.D v .A^C^D v .A^.B^C
   e :- .B^.C v .A^C^.D
   f :- .A^B^.C^D v .B^.C^.D^ v .A^.B^C v A^.B^.C v .A^C^.D
   g :- .A^B v .A^C^D v A^.B^.C v .B^.C^.D


    M2 :: Terminal             $ 2nd 7-segment module

   l :- .X^Z v .X^Y^W v X^.Y^.Z v .Y^.Z^.W
   m :- .X^.Z v .Y^.Z v .X^Y^.W
   n :- .X^Y^.Z v X^.Y^.Z v .X^.Y^Z v .X^Z^.W
   o :- .X^.Z^.W v .Y^Z^.W v .X^Z^W v .X^.Y^Z
   p :- .Y^.Z v .X^Z^.W
   q :- .X^Y^.Z^W v .Y^.Z^.W^ v .X^.Y^Z v X^.Y^.Z v .X^Z^.W
   r :- .X^Y v .X^Z^W v X^.Y^.Z v X^.Y^.Z v .Y^.Z^.W
```

```
$ The structural-level construct presented above contains
$ all the necessary information in terms of terminals and
$ input equations to flip-flops. It provides the base
$ for the actual implementation of the system. In order to
$ illustrate the complete design of the counters,
$ Figure ST-1 shows the implementation of the 8421-code
$ synchronous counter C1.
```

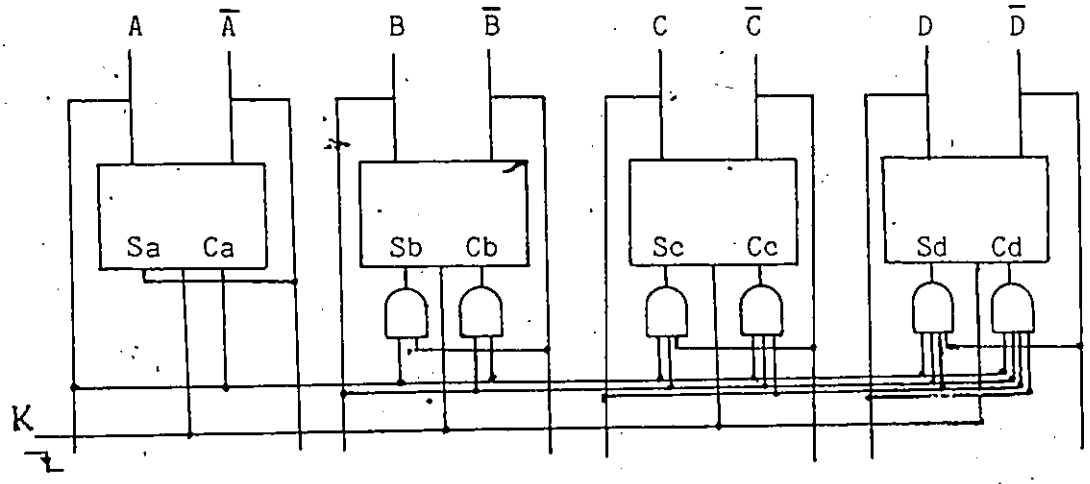


Figure ST-1 Complete design and implementation of counter C1.

```
$ The counters C1 and C2 are basically the same in physical
$ configuration.
```

TABLES OF NOTATIONS

Table APP1. Separators

Symbol	Indication	Example
,	concurrency	see EXAMPLE (p. 60)
;	sequence	see EXAMPLE (p. 60)
"	real number	15"55
:	declaration	Register : A
:=	switching	switch k := *a,b,c*
-	limits	Register : A<0-23>
$	comment	see EXAMPLE (p. 60)
/	equivalence	see Remark 3 (p. 35)
//	parallel association	see Remark 2 (p. 35)
begin end	brackets	see A-to-D converter (p. 43)
( )	brackets	see " call COSINE(A,B,C) " (p. 51)
* *	brackets	Algorithm : * a,b,c,d *
< >	brackets	Register : B<0;4>
[ ]	brackets	Memory : M [0;1023]<0;4>
{ }	brackets	freely applicable
::	modularization	Register : A :: Terminal : AT<0;7>
%	absent states	Modulo : CT [0;15%3,4]
&	double declaration	Counter & Synchronous : CT

Table APP2. Indicator

Symbol	Indication	Example
	active transition	K´⟶ A ⟵ B-C

Table APP3. Operators

Symbol	Indication	Example
+	addition	A+B
-	subtraction	A-B
x	multiplication	AxB
÷	division	A÷B
	NOT	.0001/1110
v	OR	0001 v 1000/1001
^	AND	0001 ^ 1000/0000
0	EXCLUSIVE-OR	A0B/.A^B v A^.B
shift	right-shift	shift A
rotate	circular left-shift	rotate A

exchange	swap	A exchange B
count-up	increase by one	count-up A
count-down	decrease by one	count-down A
decode	decoding	n ⟵ decode A
⟵	data transfer	B ⟵ A
>	greater than	A>B
<	less than	A<B
<=	less than or equal to	A<=B
>=	greater than or equal to	A>=B
=	equal to	A=B
.=	not equal to	A.=B
⟶	control transfer	K ⟶ A ⟵ B+C
switch	dynamic multiplexed control transfer	switch n := * a,b,c *
call	sequential control transfer	call COSINE (A,B,C)
@	concatenation	A@B
#	renaming	AB#A@B
:-	Boolean equation	X :- Y^Z
DO	iteration	DO add I=0;23 ;1
:-:	wiring	A<0;23> :-: B<0;23>

Table APP4. Declarators (reserved words)

Symbol	Déclaration	Example
Register	one-dimensional arrays	Register : A<0;5>
Subregister	one-dimensional subarrays	Subregister : A<0;2>, A<3;5>
Memory	two-dimensional arrays	Memory : M[0;1023]<0;7>
Operator	special operators	Operator : convert
Integer	integer numbers	Integer : 1,2,3,4
Real	real numbers	Real : 15"234
Boolean	logical values	Boolean : K
Operation	algorithms	Operation : .K→call AD
Sequence	state change	Sequence : S [0]→S [1]
Bus	common data path	Bus : L
Procedure	procedures	Procedure : COSINE(A,B,C)
Terminal	terminals	Terminal : T<0;7>
State	states	State : ST [0;15]
Counter	counters	Counter : E<0;23>
Subcounter	subcounter	Subcounter : E<0;12>
Modulo	counters	Modulo : S [0;15 % 3,4]
Clock	clock pulses	Clock : P<0;3>
Couple	coupling	see p. 78
Link	data path	Link : DA<0;7>
Wiring	interconnection	Wiring : A<5> :-: B<7>
Equation	input equation to flip-flops	see p. 105

- REFERENCES -

(Dietmeyer-01) Dietmeyer D.L. : Ed., "LOGIC DESIGN OF DIGITAL SYSTEMS", Allyn and Bacon Series, Mass., 1971

(Peatman-02) Peatman J.B. : Ed., "THE DESIGN OF DIGITAL SYSTEMS", McGraw-Hill, N.Y., 1971

(Fairchild-03) Fairchild Semiconductor Corp., Calf., : Ed., "THE TTL APPLICATION HANDBOOK", August-1973

(Gardner-04) Gardner R.I.,Estrin G., and Potash H. : "A Structural Modeling Language for Architecture of Computer Systems", proc., International Symposium on CHDL and Applications, September-1975, pp. 161-171

(Ellis-05) Ellis R.A. : "Modular Computer Systems ", proc., IEEE Comput. Conf., COMPCON'72, September-1972, pp. 301-302

(Franklin-06) Franklin M.A. and Ellis R.A. : "High-level Logic Modules : A Qualitative Comparison", proc., IEEE Comput. Conf., COMPCON'72, September-1972, pp. 313-316

(Clark-07) Clark W.A. : " Macromodular Computer Systems", proc., Spring Joint Comput. Conf., 1967, pp. 335-401

(1-Iverson-08) Iverson K.E. : "A Common Language for Hardware, and Software, and Applications", proc., Fall Joint Comput. Conf., 1962, pp. 253-263

(2-Iverson-09) Iverson K.E. : Ed., "A PROGRAMMING LANGUAGE", John-Wiley, N.Y., 1962

(1-Chu-10) Chu Y. : "Introducing the Computer Design Language", proc., IEEE Comput. Conf., COMPCON'72, September-1972, pp. 215-218

(2-Chu-11) Chu Y. : Ed., "COMPUTER ORGANIZATION AND MICRO-PROGRAMMING", Prentice-Hall, N.Y., 1972

(Siewiorek-12) Barbacci M. and Siewiorek D.P. : "Application of an ISP Compiler in a Design Automation Laboratory", proc., International Symposium on CHDL and Applications, September-1975, pp. 69-74

(Barbacci-13) Barbacci M., Bell C.G., and Newell A. : "ISP : A Language to Describe Instruction Sets and other Register Transfer Systems", proc., IEEE Comput. Conf., COMPCON'72, September-1972, pp. 219-222

(Barbacci-14) Barbacci M. : "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems", IEEE Trans. on Comput., vol. c-24, February-1975, pp. 137-150

(Su-15) Su S. : "Hardware Description Languages and Applications : An Introduction and Prognosis", Computer, June-1977, pp. 10-13

(Su-16) Su S. and Barray M. : "LALSD - A Language for Automated Logic and System Design", proc., International Symposium on CHDL and Applications, September-1975, pp. 30-31

(Su-17) Su S. : "A Survey of Computer Hardware Description Languages in the U.S.A.", Computer, December-1974, pp. 45-51

(Lipovski-18) Lipovski G.J. : "Hardware Description Languages : from the Tower of Babel", Computer, June-1977, pp. 14-17

(Schuman-19) Schuman S.A. and Jorrand P. : "Definition Mechanisms in Extensible Programming Languages", proc., Fall Joint Comput. Conf., 1970, pp. 9-20

(Jordan-20) Jordan H.F. and Smith B.J. : "The Assignment Statements in Hardware Description Languages", Computer, June-1977, pp. 43-49

(1-Breuer-21) Breuer M. : Ed., "DESIGN AUTOMATION OF DIGITAL SYSTEMS : THEORIES AND TECHNIQUES", Prentice-Hall, N.J., 1972

(2-Breuer-22) Breuer M. : "Recent Developments in the Automated Design and Analysis of Digital Systems", proc., IEEE, vol. 60, January-1972, pp. 12-27

(Proceedings-23) IEEE : Ed., "1975 INTERNATIONAL SYMPOSIUM ON C.H.D.L. AND APPLICATIONS PROCEEDINGS"

(Torng-24) Torng H.C. : Ed., "LOGICAL DESIGN OF SWITCHING SYSTEMS", Addison-Wesley, Mass., 1964

(Friedman-25) Friedman A. and Menon P. : Ed., "THEORY AND DESIGN OF SWITCHING CIRCUITS", Computer Science Press, Calf., 1975

(Stewart-26) Stewart J.H. : "LOGAL : A CHDL for Logic Design and Synthesis of Computers", Computer, June-1977, pp. 18-26

(Bell-27) Bell C.G. and Newell A. : Ed., "COMPUTER STRUCTURE : READINGS AND EXAMPLES", McGraw-Hill, N.Y., 1971

(Vogel-28) Vogel E.W. : "A Model Approach to the Description of Hardware Systems", proc., International Symposium on CHDL and Applications, September-1975, pp. 32-37

(Flon-29) Flon L. : "Program Design with Abstract Data Type", Dept. of Comput. Sc., Carnegie Mellon U., Pittsburg, Pa., June-1975

(Opler-30) Opler A. : "Procedure-Oriented Language Statements to Facilitate Parallel Processing", Comm. vol. 8, no. 5, May-1965, pp. 306-307

(Bartee-31) Bartee T., Lebow I., and Reed I. : Ed., " THEORY AND DESIGN OF DIGITAL MACHINES", McGraw-Hill, N.Y., 1962

(Schorr-32) Schorr H. : "Computer Aided Digital System Design and Analysis Using a Register Transfer Language", IEEE Trans., vol. EC-13, December-1964, pp. 730-737