AN INTEGRATED VLSI DESIGN ENVIRONMENT BASED ON BEHAVIORAL DESCRIPTION

A Thesis

Presented to the

Department of Computer Science

The Chinese University of Hong Kong

In Partial Fulfillment

of the Requirements for the Degree

Master of Philosophy

by

Teresa W. M. Ng

May 1989

thesis
TK
7874
N4

303916

ABSTRACT

AN INTEGRATED VLSI DESIGN ENVIRONMENT BASED ON BEHAVIORAL DESCRIPTION

By

Teresa W. M. Ng

In the early 70's, a rapid change in technology began in the semiconductor industry. Integrated circuit designers believed that there had to be a breakthrough in the design methodology in order to be able to cope with the increased complexity. By the late 70's, the concept of design automation led to silicon compilation, a term first used by D. Johannsen. This term has been used in a variety of different contexts. However, silicon compilation may be described as a translation process which accepts a high-level functional description as input and produces a layout diagram as output. This process can be broken down into several steps, and each step can be considered a compiler for the lower level.

The aim of this project is focused on providing a suitable environment for the user to generate input specification for silicon compilation - the front-end part of CUISIC (Chinese University Intelligent Silcion Compiler). The user interface of the front-end is designed to minimize human errors and the amount of information the user needs to remember (an emphasis on user friendliness). The idea, 'VHDL generator' transforms the user's input specification into a hardware description layout

which can be compiled using a VHDL compiler. The system adopts the knowledge-based heuristic approach for automating the design process.

ACKNOWLEDGMENTS

The author wishes to express appreciation to Dr. K. W. Ng and Mr. K. H. Lee for their support and guidance during the course of this research. Gratitude is especially given for their friendship and personal concern.

TABLE OF CONTENTS

1. INTRODUCTION

The rapid evolution of semiconductor technology in the late 1970's made possible enormously complex electronic systems to be fabricated on a single chip of silicon. This level of complexity and the increasing demand on chip functions, represented a major problem for the VLSI designers. At that time, it was widely recognised that unless there was a major change in design methodology, this level of VLSI technology would be grossly under-utilized due to the problem of design, layout and verification. This is much in common with the early day programmers who faced increasing memory availability problem along with increasing program complexity. The desire of design automation thus emerged with a goal to replace manual design and layout by a program which, ambitiously would take a high-level functional description as input and automatically produce as output the detailed chip mask geometry. This is the basic idea of silicon compilation - a term first used by D.L. Johannsen in 1979 [Johannsen, 1979], where he used it to describe the concept of assembling parameterized pieces of layout. The term has gained popularity recently throughout the IC CAD community, where it has been used in a variety of different contexts.

A high-level functional description means a description in which some levels of detail are obscured from the user and that is not just a textual equivalent of the layout. This translation process can be broken down into several steps, and each step can be considered a compiler for the lower level. In this way we can define a logic compiler that translates a description into a set of logic gates and flip-flops or a microarchitecture compiler that translates an instruction set description into a set of registers, buses, and ALUs.

In order to understand more about the silicon compilation process, a survey is carried out to gain a brief knowledge about its development history. This is an important step in the research since an idea of the direction for the research effort can be planned. The second step of the research is to develop a system, which, hopefully, could focus on important aspects which are not treated properly by most of the developed systems discussed in the literature. The next and final step is to find a suitable machine and then carry out the actual implementation. It must be emphasized that the system developed in this research (CUISIC) is a joint effort of two groups of people, one is the author (with emphasis on the high-level synthesis approach) and the other group is three under-graduate students (concentrating on the placement and routing methods).

This thesis can be divided into two parts, the first part, Chapters 2 to 3, is concerned with the result of the survey and the proposed improvements for the new system. In the second part, Chapters 4 to 7, some issues of the development of the CUISIC system are addressed. In the following paragraphs, an overview of each chapter is presented.

In Chapter 2, a brief discussion of the development history of silicon compilation is given. The basic principles and structural concepts of silicon compilation are also described.

With an understanding of what a silicon compiler is, an attempt is then made to analyse and comment on the logic synthesis methods used in existing systems.

Chapter 3 gives a discussion on the improvements that can be made to silicon compilation. An outline of several improvements is described. The importance of the proposed issues are also discussed.

In Chapter 4, the architecture of the CUISIC system is discussed.

A detailed description of the implementation is carried out in Chapter 5. The technique of using C as the interface language and PROLOG as the knowledge-representation language is explained. The organization of the rules and data are also presented.

Chapter 6 contains two examples that illustrate the application of the system.

Conclusions are described in Chapter 7.

2. DEVELOPMENT HISTORY

2.1 VLSI Design Process

During the 1950's, Texas Instruments, Fairchild Semiconductor, and others developed the photolithographic process for the fabrication of transistors on crystalline silicon. The steps involved in the design of early IC's are still qualitatively the same today. A top-down design methodology divides the design process into phases. As shown in Figure 2.1, the phases are design specification, functional design, logic design, circuit design, and physical design.

Each design phase is further divided into three steps consisting of synthesis, analysis, and verification as shown in Figure 2.2.

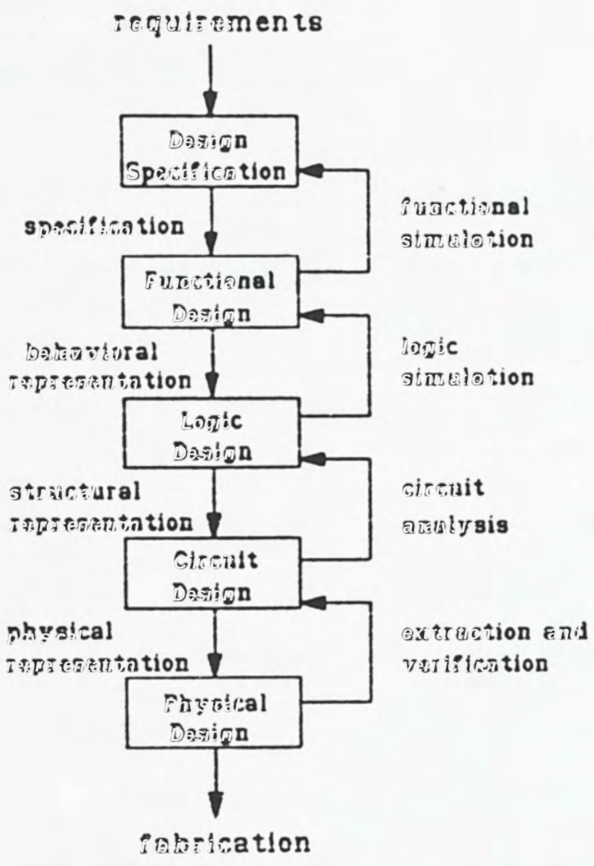


Figure 2.1 The Phases of VLSI system design for a Top-Down Design Methodology

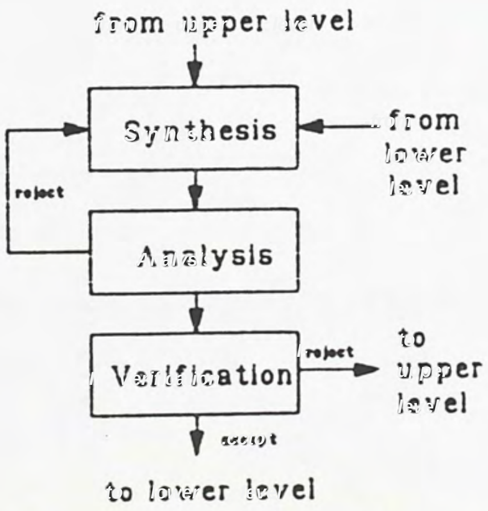


Figure 2.2 Each of the Design Phases of Figure 2.1

2.1.1 The Phases of VLSI Design

Referring to Figure 2.1, the first design phase, specification, is a time-consuming, normally manual step. Important factors to be considered are the application of the system, the performance required to meet the application, etc..

Functional design follows specification. In this phase, a functional behavior is synthesized to meet the specifications. The result may be a purely behavioral representation (for example, an instruction set description or a timing diagram), or it may include structural aspects by partitioning functionality into components. Behavioral simulation is the normal method of analysis.

Logic design, the next phase, concerns the logic structure that implements the functional design. The design representations may be either a textual, register transfer level (RTL) description or a graphic, schematic description. For analysis, these representations are simulated at the transistor, gate, or register level. The logic design is validated by comparing the results from the logic level and behavioral level simulations.

The circuit design phase concerns the electrical laws that govern the detailed behaviour of the basic circuit elements such as transistors, resistors, capacitors and inductors. In this phase, transistors are sized to meet signal delay requirements.

In the physical design phase, the behavioral or structural representations from the previous phases are transformed into the geometric shapes appears in the finished chip.

2.1.2 VLSI Design Automation Developments in the Past Twenty Years

In the 1960's, these five steps were largely manual. The engineer-user would supply a circuit or logic schematic sketched on a piece of paper. The correctness of the circuit could be verified by implementing the same circuit in discrete components ("breadboarding"). An expert layout designer then drew the mask patterns necessary to implement the circuit. The drawings were transferred to a red plastic material called rubylith which were cut away according to the drawing. This step was verified by a careful, independent visual inspection. The rubylith pattern was optically reduced to form photolithographic masks.

As time progressed, the number of devices per chip started to double every year [Moore's law, 1977]. By the late 1960's the layout method began to give way to numerically controlled optical pattern generating machines. Furthermore, commercial turnkey graphics systems [Koford, 1966] began to appear in the early 1970's. Circuit simulation programmes like CIRCAL, ECAP, ASTAP, SPICE and others [Cleemput, 1976], routing algorithms, automatic placement and layout systems, automated verification of system specifications and automatic test patterns generation algorithms were developed in the decade from the mid-1960's to the mid-1970's. Also, there was evidence that structured hardware design, analogous to structured programming, was emerging in design philosophies that emphasized wiring management and hierarchical design development with regular structures [Gray, 1979]. However, there was still a long distance to full automation. Functional elements that were placed around the chip had to be pre-designed and characterized. In order to develop design tools that would take a completely textual description of a design and translate it to layout data of silicon parts, software people began to participate in the design process. At that time, classical compilers took as input some high-level description of the function to be executed (a high level language program) and produced as output a list of low level (normally machine-level) code to execute the desired function. This forced a chip designer, usually an experienced circuit designer, to become an expert in logic design, computer architecture, and application software. This requirement for vertical expertise in one person and increased demand for new designs created a shortage of chip designers. Also, the design complexity prolonged the design cycle, which became almost as long as the lifetime of the product. To solve this design crisis, an advancement in design methodology for VLSI technology is needed. One of the approaches is using silicon compilers.

2.2 The silicon compiler

The term "silicon compiler" was introduced by Johannsen in 1978 to describe his original work. A silicon compiler takes a high-level functional description as input, but produces, instead of machine code, a detailed chip mask geometry as output. The advent of such tools for VLSI system design leads to a revolution in design since they tend to displace human designers instead of assisting them in the design cycle, thus, introducing a methodology completely opposite to the CAD approach. The combination of first time correctness, along with a rapid design cycle and high-level user interface, makes the silicon compiler a prime system development tool of the following years.

The Bristle Block [Johansenn 1979, 1981] is a representative example of early silicon compilers. It is essentially a system that performs the majority implementation compilation while placing a set of constraints on designer. The system was initially based on the structured design methodology and used procedural calls which differed from the existing design practice at that time. The chip design emphasises the hierarchical style.

This hierarchy imposes a locality on the various sections of the chip that can be exploited when performing design rule verification and electrical simulation of the chip. Optimal VLSI designs are based on one of a number of highly concurrent architectures, each with its own design style, correspondence with mask geometries, and timing corrections. Each design style is supported by one or more chip compilers. Each of these compilers accepts formal functional specifications as input and produces geometric information for masks suitable for any one of a number of fabrication technologies.

The FIRST compiler reported by Den Yer et_al in 1982 [Den Yer, 1982] was built around an underlying bit-serial signal representation, and systems are implemented as a hard-wired network of pipelined bit-serial operators. The hardware implementation of the circuit consists of a network of interconnected bit-serial operators. Each bit-serial operation is implemented as a separate functional block, which is made up of leaf cells from the library. A leaf cell may comprise, say, a single bit-slice of a given function, and the complete operator would then be several leaf cells 'glued' together. Systems implemented by the FIRST compiler require the relation of a target floorplan architecture suitable for bit-serial systems.

In 1983, Macpitts was developed at MIT Lircon Laboratory [Southard, 1983]. It was designed to be a synthesizable, algorithm description language. Macpitts allows a designer to specify an algorithm as though completely general and sufficient parallelism are available in some general-purpose machine. In other words, any control/data flow graph can be directly specified in Macpitts. Then the Macpitts compiler "extracts" the minimum-hardware micro-programmmed machine which executes that parallel algorithm, with all the bus and circuit merging into and sharing them as implied by the algorithm. However, MacPitts lacks system partitioning facilities and only generates NMOS designs. The test generation mechanisms of MacPitts still require human interaction.

As stated in session 2.1.2, these first generation silicon compilers could only handle designs that fall into a narrow range of pre-ordained "target implementations", i.e. individual compiler is only good for a particular system. Also, these early silicon compilers seem to exclude or strictly limit the automatic generation of the control structure of the chip. Nevertheless, research efforts toward the ideal silicon compiler appeared to be accelerating in the mid 1980's.

2.2.1 Progress towards an "ideal" silicon compiler

Werner [Werner, 1983] considered the progress towards the ideal silicon compiler really involved solutions to two complementary design problems. The front end solution (also known as behavioral silicon compiler) is the translation of a brief behavioral or functional description into a more precise description that is still implementation independent, and the back end solution (also known as a structural silicon compiler) is the automatic generation of the chip layout from that intermediate description. Research work on logic synthesis plays an important role in tackling the front end problem, for example, the use of knowledge-based expert system in logic transformations [Peskin, 1983]; the conversion of a high level system specification into a net-list suitable for input to a layout system. [Berdas, 1983] and a register-transfer level synthesizer reported by Kowalski & Thomas [Kowalski, 1983]. A workshop on logic synthesis and silicon compilation held in 1984 revealed that although totally automatic "silicon compilers" with a wide application were not yet available, participants were quite optimistic that the technical problem could be overcome shortly.

On the other hand, efforts to automate the "back end" - artwork generation of silicon compilation continued (or somtimes called a structural silicon compilers). In one sense, the problem of converting a structured circuit specification to an IC layout had been solved. Several semi-conductor suppliers, as well as larger system companies, can automatically or semi-automatically lay out a gate array or a standard-cell chip from such an input specification. However, work continued on the developemnt of more powerful and more general tools that will completely automate the implementation-specific portion of the silicon compiler. In contrast to initial silicon compilation efforts in the late 1970's, the emphasis then was a compilation at the module level, rather than at the whole-chip level. Module generators had been reported [VLSI design staff, 1984] to generate silicon instructions or functional modules which were then linked into a working chip design. These silicon-instruction generators and module linkers enabled designers to experiment with alternative floor plans and architectures, because various cell layouts could be produced, quickly, repetitively, and accurately.

More powerful silicon compilers with the ability to verify the designed chip became commercially avaliable as a result of these promising progress. The Genesil system [S. Johnson 1987] for example, allows designers to first refine their ideas and compare different approaches for the best price and performance quickly. After the designer has explored the possibilities and settled on a block diagram, the system verifies the behavior of the chosen architecture, establishes the floor plan, routes the connection between circuit blocks and modules, and creates a tape from which a particular foundry will make the actual silicon parts. The integration of silicon compilation software into an engineering workstation [E. Lee et_al 1984] was another step forward for silicon compilers. The workstation based CMOS silicon compiler, with its built-in utilities, can edit graphics and layouts, verify timing, simulate logic, generate a netlist, and check design and electrical rules. Within the system the silicon compiler transforms information generated by the graphics editor into a format that resembles the physical layout of a handcrafted IC. The Metasyn silicon compiler, an improved version of the MacPitts silicon compiler reported by R. Southard [Southard, 1983] works directly from a functional description of the chip and requires no hardware experience from the designer. The compiler also contains high level simulation that let the designer observe the device's internal operation and its interaction with a simulated environment. Other systems developed along similar line included the YASC high level silicon compiler [Krekelberg et_al, 1985], and a module compiler Concorde [Collett, 1984].

Supporters of this approach in solving the design crisis believe that knowledge is algorithmic, and that translators can be written to generate or synthesize the solution or some part of it automatically from a high-level description of the problem.

There is another school of thought which believes that human knowledge can be captured and stored in the knowledge base of an expert system. The knowledge in the knowledge base can be divided basically into three categories : Concepts include basic terms of the problem domain (VLSI design in this case) which usually can be obtained from textbooks. Rules describe particular situations and desirable actions to be performed. This knowledge is based on experience, and is obtained from an expert. Strategies are precedures that aid in guiding the search through the knowledge base and help resolve conflicts when several equally plausibe rules apply. The other two components of an expert system are a working memory that stores the current design description, and an inference engine that searches the knowledge base for applicable knowledge and makes design refinements on the basis of the current design description.

While expert systems try to solve problems with little regularity and large complexity, compilers are written for problems with predominantly regular structure. These problems are well defined and adaptable to mathematical formulation. However, these tools assume a certain type of solution or target architecture which tends to limit their applicability.

2.2.2 The Age of Intelligent Compilation

In the previous section, the problem of flexibility and optimization were addressed. These shortcomings of silicon compilers can be very efficiently handled if the two different approaches can be combined. Thus, intelligent compilation can be viewed as the most logical step in the evolution of silicon compilers. With silicon compilers, users obtain rapid performance feedback during the design process, enabling them to evaluate how well the design in progress is meeting their goals. In intelligent compilation, users actually submit design goals to the system, which automatically seeks an optimal solution. Rawson & Triberger [R&T, 1987] describes the ability of second generation compilers to optimize semicustom circuits.

The Intelligent Compiler described by Johannsen et_al. [Johannsen, 1987] features three main characteristics in this aspect. Firstly, unlike ordinary circuit optimization systems which aim at optimizing a single variable - speed, number of gate, or area - at a time, Intelligent Compilers go one step further towards simultaneous optimization of multiple variables. Secondly, unlike existing back-end tools which could only optimize the design based on an initially debugged circuit, AI techniques are used to learn how to make improvements in a circuit design that affect the entire design process, front end as well as back end. Thirdly, with the Intelligent Compiler's AI-based system, the user enters his circuit schematic and the program examines the function of each element of the design. It then looks through its data base to see if it contains a circuit that provides a better implementation. If the circuit that the user has entered is superior to all of the circuits already in the data base, the software "remembers" the circuit and uses it in future evaluations.

To deliver the flexibility it promises, silicon compilation also relies on the expert system technology. M.Schindler [Schindler, 1986] describes two such efforts in the U.S.A.. One is the Design Automation Assistant (DAA) at Carnegie-Mellon University. Though still experimental, the system blends about 700 rules with algorithmic modules written in C. DAA's designers hoped to broaden the narrow confines of silicon compilers, thereby eliminating many of the shortcomings. Another effort by AT&T Bell Laboratory (Cardre) concentrates on the lower level of IC design (from register-transfer level or down) and could therefore eventually use DAA output as its input. Unlike a single expert system, Cardre will comprise a collection of small expert "agents", all co-ordinated by a manager agent, which implements the system's meta-rules. Following a similar line, other expert systems were developed to perform specialized subtasks along the design process. Synapse [Subrahmanyan 1986], for example, is an expert system which enables very high level specifications of a problem (consisting of the desired functional and performance specifications) to be mapped into custom VLSI circuits. It also supports human interaction and machine learning.

2.2.3 Future Trends

Projecting the scenario of silicon compilation, future developments would be expected to follow the following trends :-

i. The incoporation of AI techniques in silicon compilation will continue and become a mainstream development. Intelligent compilation which adopts AI techniques so far only puts emphasize on the optimization of design. Another area where AI techniques could be utilized efficiently would be the interface between the user and the system. With knowledge-based expert systems, guidance or expert advice can be supplied by the system in acquiring design specification and information which are crucial to an optimal design. This is most helpful for system engineers who do not have a sound knowledge in IC design.

ii. Many of the hardware description languages developed generally operate in an environment that restricts the use of the language to a particular design application, design methodology or set of design tools. Besides, only a few of them are compatible. These issues lead to problems in:

- communication of design data between companies carrying out different parts of the design process,

- transmitting manufacturing and fabrication data for a designed part, and

- encouraging the development of new and innovative CAD tools;

It seems that a standardized language is a must. An extremely important feature required for a modern design and description language is independence of the language's descriptive capability from any tool set. It should allow the designer to create a model of a design that fully describes the hardware aspects the design is intended to include.

3. A PROPOSAL FOR AN INTEGRATED VLSI DESIGN ENVIRONMENT

So far, it can be seen that a lot of work have been done during the 1980's with the intent of enhancing the integrated circuit designer's capabilities by providing him with powerful architecture, function, and logic design tools. Logic design aids, such as logic simulators or design editors for circuit diagrams, have already been implemented. Hardware description languages, which can describe the whole hardware structure in terms of the connective structural description at the gate or the circuit level, have been implemented as well. Architecture and function design tools, generally called logic synthesis methods, for the design of VLSI circuits have received a significant amount of attention. Early day silicon compilers were mainly capable of transforming structural description of systems to geometric layout for device fabrication. Failure analysis and design evaluation must still be done manually, and corrections to the design specification must be iterated through the entire compiler for evaluation. Although intelligent compilers can handle some of the problems discussed above, there are still some of the problems that have not been discussed in the open literature. After carrying out a survey in the required field, a proposal is presented which set up the guidlines for the system developed in this research.

3.1 A Structured Design Method

A structured design method is of major importance for automating the design process. The design method should permit the design to evolve rationally as a modular system. Describing the design as a formal hierarchy leads to well-defined interface between the components of the design system, and produces a simplified design. From the perspective of physical design, hierarchical design provides the framework that makes automation possible and enables various design styles.

The knowledge-based heuristic approach has been widely used in developing design automation systems [T.Uehara, 1982, T.J. Kowalski, 1982]. Hierarchy involves decomposing a system into a set of components. These components are recursively decomposed into subcomponents until all components are small enough to be manipulated [T.M. McWilliams, 1978, W. M. vanCleemput, 1977, K.A. Duke, 1987]. Although some major contributions have been made in this area [P. Six, 1986, S. Costanzo, 1987], there is still a lack of fast and convenient methods for the synthesis of hardware structures from high-level specifications. Furthermore, to upgrade the system to the expert designer's level, artificial intelligence (AI) methods should be incorporated into it. Systems like the one proposed by N. Kawato [N. Kawato, 1982] developed in the FUJITSU Laboratories Ltd., and the USC ADAM system designed by D.W. Knapp and Alice C. Parker in the University of Southern California [D. W. Knapp, 1986], both use the AI approach and are frames and demons based system.

In order for a structural, hierarical decomposition to be of greatest value, the design process must follow certain rules. Two of the most important rules are modularity and locality.

Modularity :

Modularity implies a well-defined, unambiguous functional interface. In the physical domain, modularity dictates clear specifications of connection points. Modularity allows the designer to understand his design and permits a design system to verify the attributes of an electrical component in its environment.

Locality :

Locality implies that the details outside a component are not important while the interior of a component is being considered. Thus, locality is a form of information hiding that reduces complexity.

Therefore, the designed system would be a hierarchical-based system which follows the above rules during the design process.

3.2 The Proposed Logic Synthesis Approach

3.2.1 The Design Automation Tools

Several classification criteria for design automation tools for integrated circuits are usually used in logic synthesis problems, they are:

i the level of the circuit view, i.e. behavioural, structural or geometric,

ii the level of details of the circuit description, e.g. architecture, register-transfer, logic, electric, etc.,

iii the performed design task, e.g. analysis or synthesis and

iv the implementation technique, e.g. full custom, standard cells, gate arrays, etc.

Our approach to synthesis is based principally on classification (i). On the whole, automating the process of micro-architecture design from a behavioral language requires the addition of a large amount of knowledge to the design specification. In the past this was accomplished by restricting the design model to a limited set of alternatives, then casting the behavior onto one of them. In this way the specialized knowledge about design trade-offs can be 'hard-wired' into the synthesis process. Specific examples are SYCO[Jerraya, 1986] and MacPitts[Southard, 1983] which were targeted at microprocessor design, and CATHEDRAL[DeMan, 1986] optimized for signal-processing design. This limits the flexibility of the system, therefore another process has to be introduced to solve the problem.

3.2.2 Two Types of Automation Process

A considerable amount of effort has been devoted to the automation process, and many DA systems have been proposed and constructed. They can be classified into the following two categories.

(1) Successive decomposition type systems that accept some high-level specifications and expand them hierarchically (behavior, function block, gate, circuit, symbolic layout, mask layout).

(2) Incremental refinement type systems that consist of a number of parameterized layout generator for a given set of entities.

As the refinement type systems by pass the conventional circuit-design and layout-design stages for some finite set of functions, or operators, they are efficient for (but limited to) some specific purposes and device technology.

Decomposition-type systems are more flexible and more general-purpose systems. They do not limit device technology

(NMOS, CMOS, ECL, TTL and so on), and device architecture (VLSI system and non-VLSI system). However, no straightforward algorithm for realizing these general purpose tasks is found [Southard, 1983].

The system presented in this project is tackling the decomposition-type system with a knowledge-based expert system approach. The knowledge base consists of rules to determine the appropriate styles, design tradeoffs and strategies. The design itself is iteratively refined by specialized algorithms which quickly search the design space to implement a new potential design based on the component constraints (such as the number of busses, or ALU's). Separation of the knowledge about how to correctly implement the design from the knowledge about meeting design constraints simplifies the design strategies and consequently the implementation of the knowledge-base.

3.3 The Suggested User Interface Languages

In additional to the above drawback, it is found that almost all the logic synthesis systems described in the open literature require the designer to supply specifications written in a hardware description language [Jerraya, 1986]. Although text-style languages are suitable for processing in a computer, they are not suitable for describing hardware behaviour directly. This is evidenced by the fact that schematics have been used as the most dominant hardware design representation, and that text-style languages have been used by few hardware designers. Essentially, there are great differences between the design process by a human and the design process by a computer. Human designers can optimize their designs by using global views and such invisible rules as intuition, heuristics, and know-how. On the other hand, computers have high abilities in repetitive, massive, and numerical processing. Therefore, software implementation and the selection of the user interface for the system seem to be the most important point for developing the silicon compiler.

Based on the above requirements, a system has been developed that accepts several levels of specification of the chip. The idea behind the project is that a user who is not familiar with techniques of digital and VLSI design should be able to create a special purpose chip that runs the user's program. The interface between the user and the system should be as user-friendly and convenient as possible. The implementation language is C, with Prolog as the underlying knowledge-representation language. Knowledge representation is an important issue. A single multi-purpose framework of knowledge representation would be simplest. However, as it is noted that hardware logic design employs various kinds of knowledge, and design data must be represented as well. In addition, human designers usually switch from one representation to another in the course of their work. For these reasons, no existing tool is adopted here for knowledge representation.

3.4 The Specification Language : VHDL

As mentioned in the above section, the developed system accepts several levels of specification of the chip, and it is understood that in many cases, designers describe the hardware behaviour in a text-style Hardware Description Language. A hardware description language,therefore, is still needed in describing the designed circuits to provide the completeness of the system. VHDL is chosen since it provides a standard textual means of description for hardware components at abstraction levels ranging from the logic gate level to the digital system level. It also provides precise syntax and semantics for these hardware components, enabling design transfer both within and among organizations. The language is designed to be efficiently simulated and natural for hardware designers. The key concepts embodied in VHDL provide the designer with the ability to

- create hierarchical design descriptions,
- create functionally equivalent alternative models,
- create design models that intermix design descriptions at many levels of abstraction,
- describe parallelism and concurrency among units under designed,

- describe timing relationships among units under designed,

- differentiate between combinational and sequential logic,

- separate control flow from dataflow,

- describe models that can be automatically synthesized, and

- analyze models for functional equivalence.

4. THE CUISIC SYSTEM

The design of this system was undertaken as a joint effort of two groups of people, a group of three undergraduate students as their final year project, and the research project described here. The "Chinese University Intelligent Silicon Compiler" (CUISIC) project, is designed to oversee a suite of knowledge-based and hard coded design activities. It is designed to construct sequences of design activities which are not explicitly coded into the system, but are constructed in response to the needs of a particular set of specifications and constraints. The research effort is concentrated on providing a suitable environment for the user to design a particular chip. The scope of the research covers the front end part of the system, as shown in Figure 4.1.

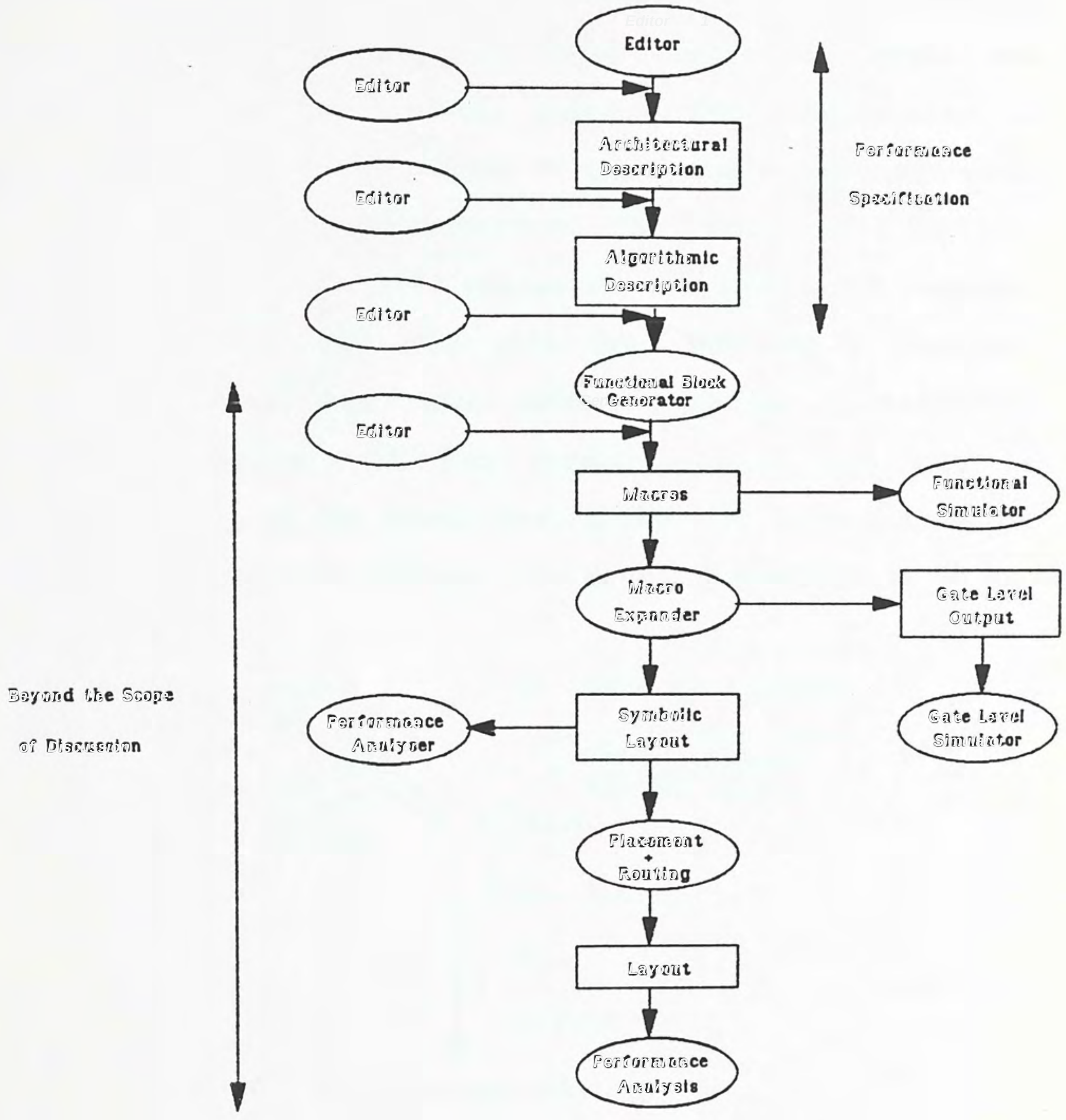


Figure 4.1 Scope of the Research Work

4.1 Domain Description

As mentioned in section 3.3, the system allows the designer to enter into the system at different abstraction levels and styles. In order to represent these different approaches to the system, a well-known Y-chart (a tripartite representation) is used for the design representation. The Y-chart is a succinct description of the different phases of the process of designing VLSI systems [COMPUTER, 1983, 1986, Proc. IEEE, 1983]. The axes in the Y-chart represent three different domains of description : behavioral, structural, and physical. Along each axis is different levels of the domain description. The farther from the center the Y, the more abstract the level of description.

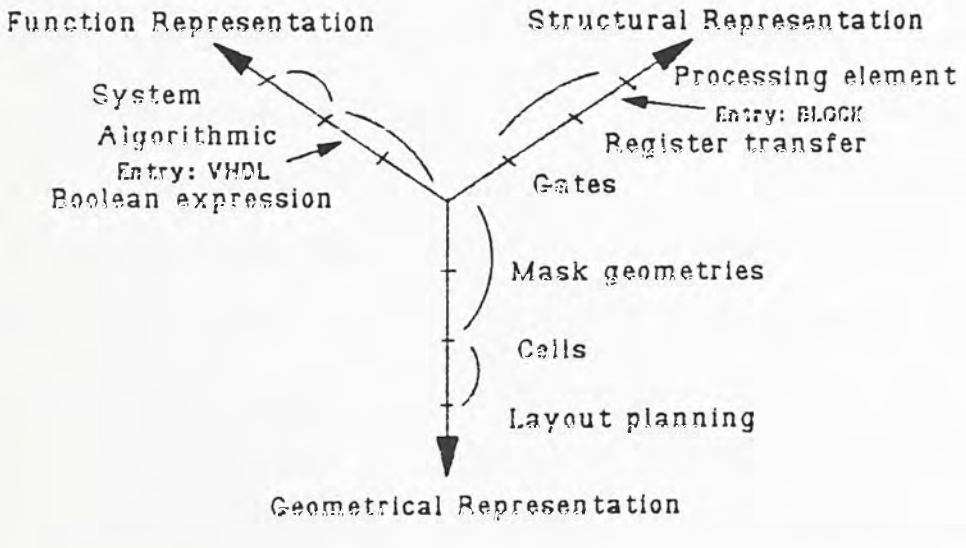


Fig. 4.2 Design Methodology in CUISIC

The idea behind the project is that a user who is not familiar with the techniques of digital and VLSI design should be able to create a special purpose chip that performs the user's required functions. The interface between the user and the system should be as user-friendly and convenient as possible.

4.2 System Overview

The design system (part of CUISIC) is implemented on a SUN workstation, taking the advantages of high performance and the availability of the window-based environment equipped with million bytes of main memory.

The system developed in this project supports hierarchical logic design. It is a system with its own set of planning rules. The compiler uses knowledge-based heuristics in order to give advice to the user. The compiler also uses knowledge about the design process declaratively represented by a network of frames; these frames contain knowledge about the design of hardware including taxonomy and methodology. Frames were originally developed for AI by M. Minsky [P.H. Winston, 1977]. The static structure of the hardware is embedded in the frames of the system. The dynamic behavior of the hardware and the designer's intention are attached to the frames as demons, which are data-driven actions stored in frames. Figure 4.3 shows typical frames for storing the structural information. A complete circuit can be represented by a network of such frames. It is believed that this frame representation for a logic circuit gives the system flexibility and power [T. Saito, 1981].

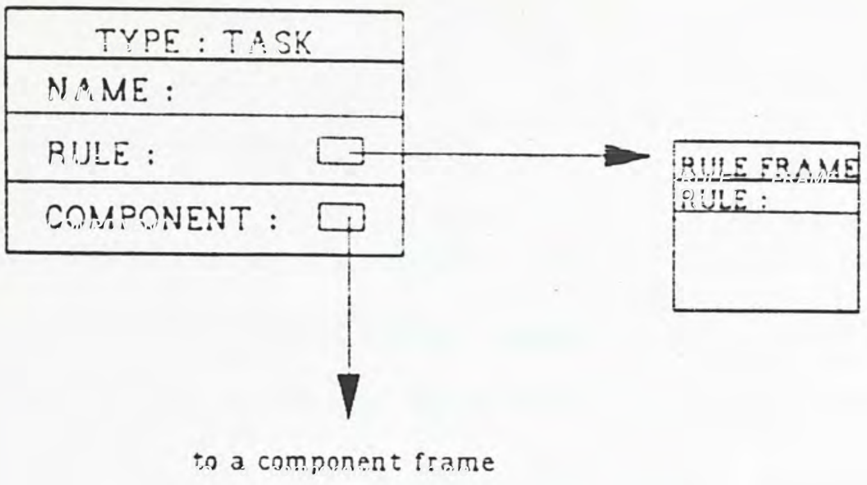


Figure 4.3 Network Representation Using Frames

The start up screen is written in C and is a window handling program. It creates windows for the user to select the design item. After the user has selected an item, the system starts processing the data given by the user. If everything goes well, a diagram for the designed system would be displayed, together with their associated VHDL descriptions. If errors occur during the process, the system will ask for help from the user interactively. However, if the errors are so serious that the system cannot recover from them, it will prompt the user with the appropriate error(s) and stop the process.

4.3 System Description

Figure 4.4 shows an overview of the complete system which consists of eight subsystems, namely, the user interface, the form-library, the strategy rule-base, the information data-base, the VHDL generator, the enquiry rule-base, the verifier and the timing verifier. The system provides a choice for the user to specify the function or the item name for the circuit under design.

The design tasks are carried out in the following manner: firstly the selected design function/item is read by the system. If it is a function name, the system searches the rule-base to retrieve the task frame which contains that specified function as its name, it proceeds to the field 'rule' inside that frame. This field may point to a rule frame which contains a set of planning rules governing the design function, these rules determine the component list for the specified task. With each of the components, the system searches the form-library for the particular entry form. If the circuit under design is specified by its name, the system will search directly from the form library to get that particular entry form. Each entry form may contain some guidance for the user to enter the information related to the design. After the form is filled, the system searches the rule-base to find the task frame with the specified name, and then proceeds to the field 'rule' inside that frame. The rules contain the necessary information that will be needed for the hardware implementation at the later stage. If no more rules are found in the respective rule frame, CUISIC proceeds to the 'component' field in the task frame. The system supports hierarchical logic design, therefore each item may be partitioned into several components. For each component the above procedures repeat. If no more component is found, the input procedure is considered complete and the data is passed to the verifier for investigation. If no unrecoverable error occurs, the data will be passed to the VHDL generator to generate the VHDL description. This generator finds the hardware frame for that name and inserts the dynamic behavior into the slot. The VHDL associated with it will be placed into the data base. A description of the sub-systems follows.

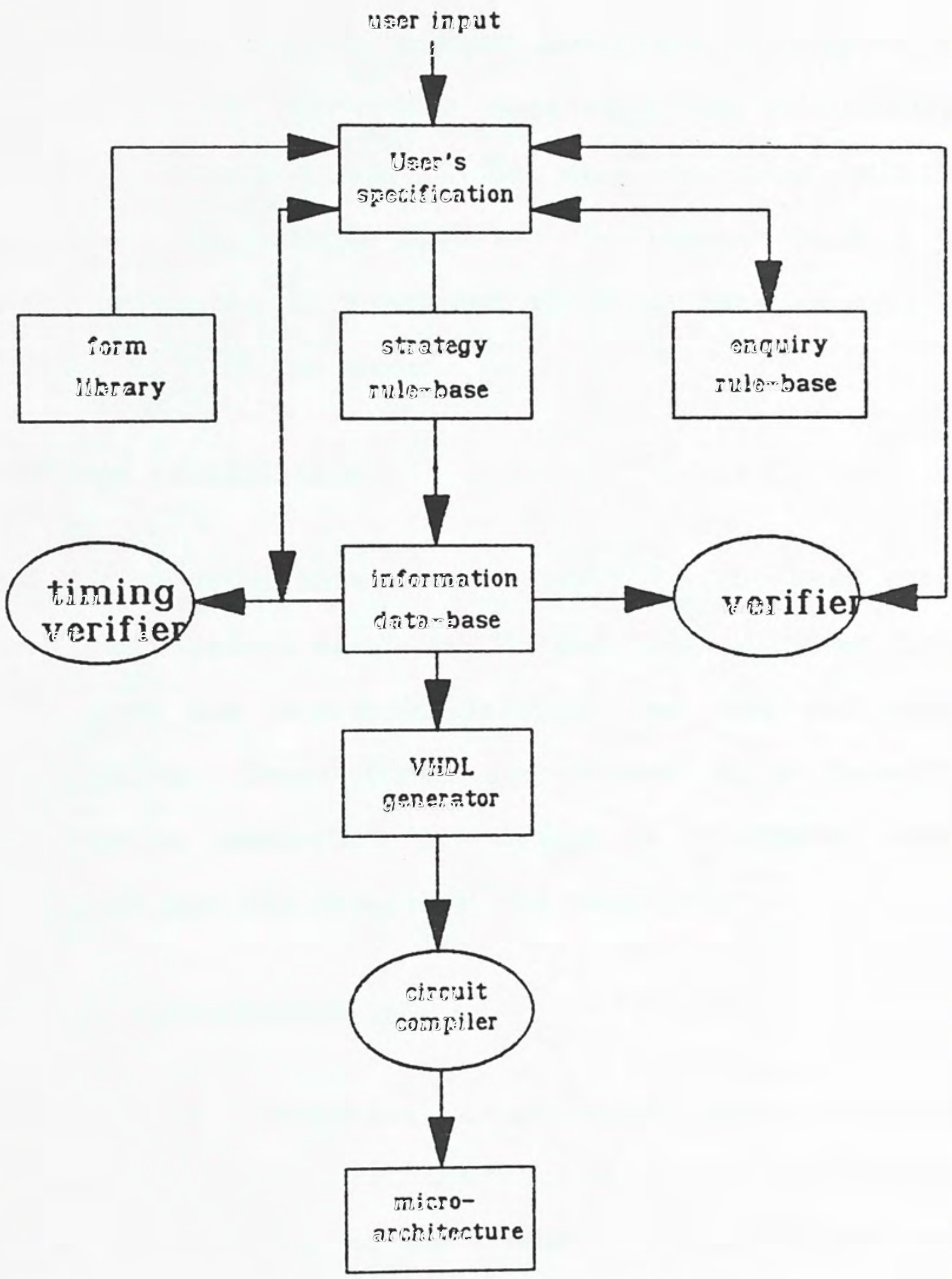


Figure 4.4 An Overview of the CUISIC System

4.3.1 User Interface

It is important for a human machine interface to support a tool which realizes quick *turn-around time* with few possibilities of user errors. Designers should be able to work with little disturbance in the thinking process. To realize such a design environment, a system is developed which accepts several levels of input entries from the user:

a. Algorithmic organization

The menu and question-answer tools would be the best choice in this case. The control structure of the tool is hidden from the user. The form has automatic defaults and are self-prompting and self-checking. These forms are stored in a form-library. Once the form is completed, the design is considered complete. A block diagram for the design will be displayed.

b. Functional organization

For this kind of description, a graphical editor would be a better choice. In this case, the system allows the designer to provide the architecture of the design using predefined modules or blocks selected from the function set database provided. It can be treated as a means for the designer to update the knowledge base of the system. The knowledge-based design environment applied here is called DISCDA (DIgital System

Controller Design Assistant). DISCDA is an interactive system with a user-friendly interface that provides multiple windows and popped-up menus modelled after the Smalltalk-80 environment (K.W. Ng, 1988). Using the graphic editor, the designer can create, modify, and delete the design data in frames.

c. VHDL description

Most of the silicon compilers developed to date use hardware description languages as a means to specify the behavior of a design. Since CUISIC is designed to be an interactive system with user-friendly interface, it also allows the designer to enter into the system using an industry standard algorithmic hardware design language VHDL. The VHDL description will be stored in the hardware frame in the knowledge base discussed above.

d. PROLOG - knowledge representation language

This is another way of describing the behavior of the VLSI chip. PROLOG is a logic programming language. A designer describes the system by specifying its operations or functions without necessarily giving implementation details such as the hardware components needed to implement the system. There is a VHDL generator converting the PROLOG description into the VHDL form.

4.3.2 Form-Library

The library contains all the forms for design items included in the system. There are a set of rules associated with each of the forms so that errors will be minimized when the form is filled. Since the system uses a hierarchical design approach, the rules also contain the component list of the specified item.

4.3.3 Strategy Rule-Base

This is the heart of the system. It contains a set of planning rules which govern the design of the required item. The compiler uses knowledge-based heuristics in order to guide its choice of tasks where more than one option is possible. The compiler also uses knowledge about the design process declaratively represented by a network of frames; these frames contain knowledge about the design of hardware including taxonomy and methodology. It is believed that this frame representation for a logic circuit gives the system flexibility and power.

The strategy rule-base is a collection of frames, which collectively describe design activities, styles of hardware, and functional classes of hardware. It is organized into three classes:

I. task frame (Figure 4.5)

It describes the procedure of a design, which can be specified by the item name or the function to be designed. Several designs are considered:

a. items :

i. microprocessor

ii. controller

iii. co-processor

iv. CUISIC primitive (random logic)

v. micro-computer

b. functions :

i. addition

ii. multiplication

iii. division

iv. boolean algebra

type: task
name: ALU
rule:
component

Figure 4.5 Task Frame

II. hardware frame (VHDL)

It describes ways to implement particular functions and classes of functions in hardware (Figure 4.6).

type: hardware
name: ALU
VHDL:
component:

Figure 4.6 Hardware Frame

III. style frame

It describes variations on basic hardware structures (Figure 4.7).

type: style
name: CLA
variety:

Figure 4.7 Style Frame

4.3.4 Information Data-Base

All the information, whether they are provided by the user or are deduced by the system using rules, are stored in this area. The information can be retrieved at any time if necessary. Information can also be stored directly from the editor to the data base if they do not require the rule check.

4.3.5 VHDL Generator

This generator takes in the information stored in the information data-base, searches the hardware frame from the hardware-frame library, gets the correct description of the design and then creates a new hardware frame which stored all the necessary information for the required design.

4.3.6 Enquiry Rule-Base

The system provides interaction with the user through the enquiry rule-base. This rule base provides suggestion for the user if there is any query about the questions asked by the system.

4.3.7 Verifier

Although rules have been applied during the form entry process in the input stage, some important alternatives may not have been investigated, inappropriate techniques may have been applied, or relevant information may not have been examined. Thus we need a final verifier to verify if there is any constraint that the system has overlooked.

4.3.8 Timing Verifier

The successful design of large scale integrated systems require careful management not only of the two-dimensional silicon area but also of the operation of the system in the time dimension. The digital design may be functionally correct but may be unreliable or too slow. This is particularly important for the hierarchical design since each component that works well individually may not guarantee to perform correctly for the combined system. Therefore it is necessary to have a timing verifier to analyse the timing problem for the system, to make sure that no violation exists for different subsystems and to set out the timing constraints for the system, if any.

5 THE IMPLEMENTATION OF CUISIC

This research effort implements the ideas of the design environment and the structure of an ideal silicon compiler described in chapter 4. This implementation is part of the CUISIC system and it consists of a coherent set of programs that perform the synthesis of a digital system from its behavioral description to the final hardware circuit specified in the hardware description language - VHDL.

The system is implemented on a SUN3 workstation with SUNTOOLS as the interface medium. The implementation langauge is C, with NU-Prolog as the underlying knowledge-representation language. It runs under Berkeley 4.3 Unix. The program consists of four parts : information retrieval (user's input specification), design generation (strategy rule-base), verification (verifier) and VHDL file generation (part of the micro-architecture compiler). The control of the program flow is governed by a controller, its function is to determine which of the function blocks in the system should have control.

The following is a detailed description of the implementation carried out for each part of the program described above.

5.1 Information Retrieval (User's Input Specification)

The system starts with a start up screen as shown in Figure 5.1. It is a window handling program written in C. The modules included in the system are displayed for selection. Once an item (specified by its name or its function) is selected, a symbol with the specified name will be shown on the screen (this serves to remind the user that this particular item has been selected), together with its entry form for the user to fill in. For complex items, the whole internal connections will be displayed. By selecting each internal block on the screen, its respective property lists (in terms of form entry) will be displayed. A behavioral description of the selected item is given by the user through the completion of the form. One of the reasons for choosing form entry is that high-level functions typically require many parameters. Also, for easy comprehension, schematics for the objects must both be rich in graphics and contain textual information. Therefore, the system generates diagrams for engineering documentation. The entry form contains the default value for each of the questions in the form. Figure 5.2 is an example of an ALU specification form which exemplifies the number of parameters required for specifying an ALU block.

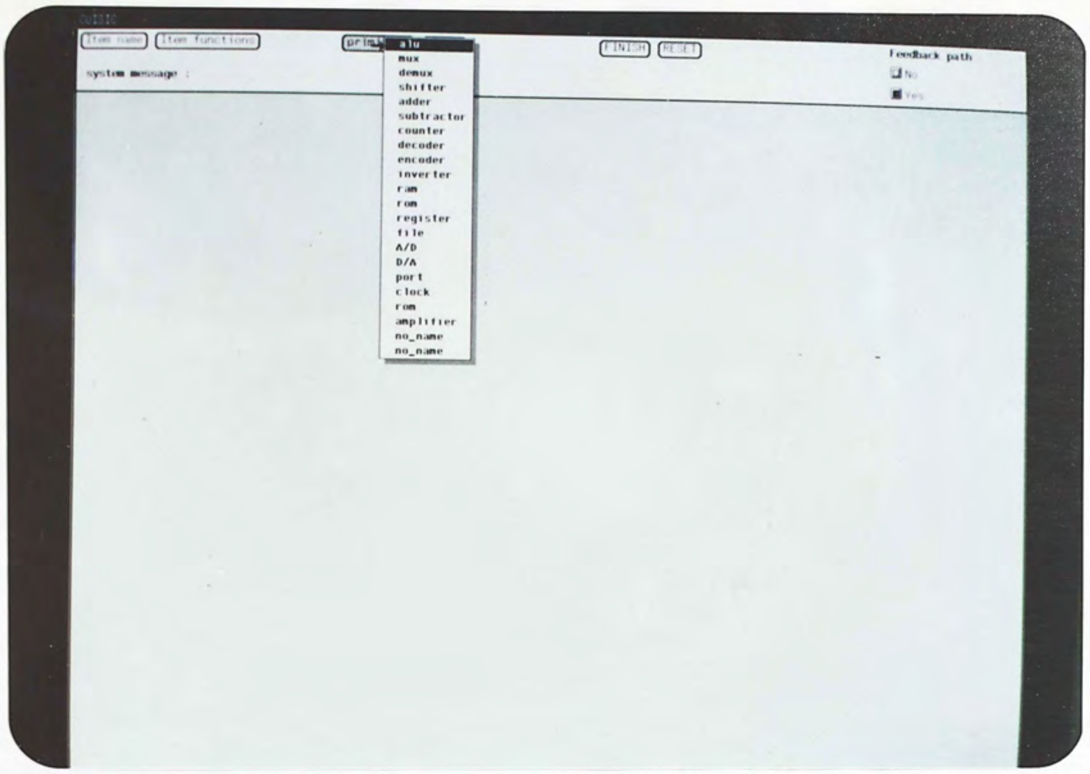
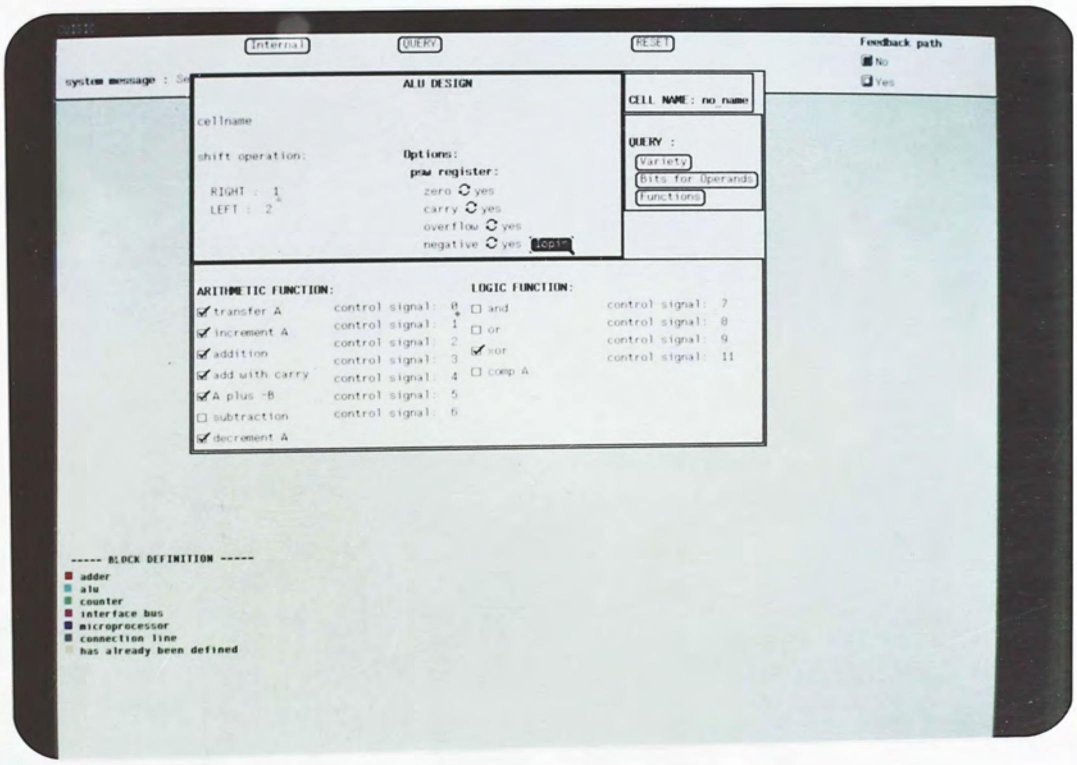


Figure 5.1 Start Up Screen of the System



ALU DESIGN

CELL NAME: no_name

cellname

shift operation:

RIGHT : 1
LEFT : 2

Options:
psw register:
zero yes
carry yes
overflow yes
negative yes

ARITHMETIC FUNCTION:

✓ transfer A	control signal: 0	☐ and	control signal: 7
✓ increment A	control signal: 1	☐ or	control signal: 8
✓ addition	control signal: 2	✓ xor	control signal: 9
✓ add with carry	control signal: 3	☐ comp A	control signal: 11
✓ A plus -B	control signal: 4		
☐ subtraction	control signal: 5		
✓ decrement A	control signal: 6		

LOGIC FUNCTION:

----- BLOCK DEFINITION -----
🟥 adder
🟦 alu
🟩 counter
🟪 interface bus
🟦 microprocessor
⬛ connection line
⬜ has already been defined

Figure 5.2 An Example of an ALU form entry

The specifications of the input and output pins for an ALU (for example) can be specified through the selection of the "IO pin" button, which is used to change the default values of the I/O pin specifications. When this button is selected, an I/O form for the item will be displayed. For example, the user may change the input pins of the ALU to :

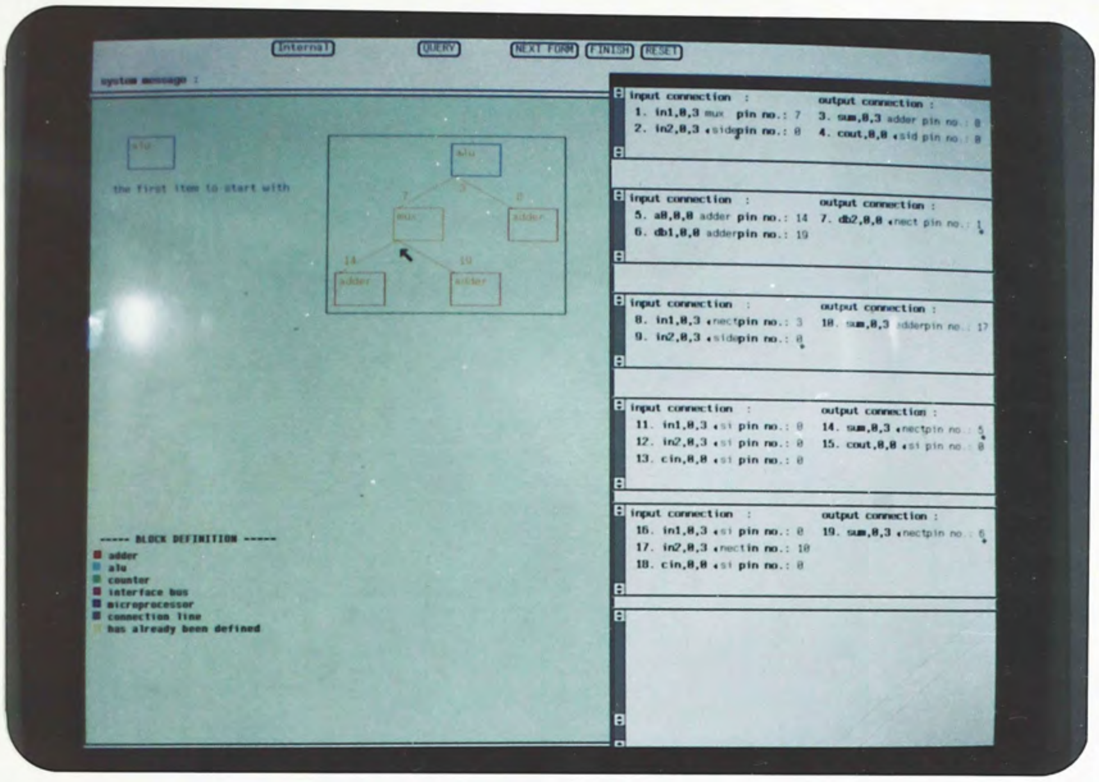
in1,0,2 in1,2,3 in2,0,3 cin,0,0

note : in1,0,2 means the input 'in1' has lines from 0 to 2

The number of lines corresponds to each of the pins cannot exceed the size of the data bus that has been specified in the form, otherwise, the system will automatically truncate the number to be the same as the size of the data bus.

After the user has specified the function of the chip, the system searches the rule-base library to retreive the required frame which contains a set of planning rules. These rules govern the design of a circuit for a particular function. Take the specified function 'addition' as an example. The system searches the rule-base library to retreive the frame with name 'addition', then it goes to the 'rule' field which specifies the design rules. The first rule determines the components for the function, therefore the system passes them to the component field. In this case, the list only contains an "adder". The system starts to search the component field, it finds an 'adder', then it searches the form-library to get the corresponding form entry. The data retreived from the form is passed back to the 'rule' field to determine the internal structure of the design. For example, if the required number of operands is 3, the rules determine the number of adder connected together is 2. Then a connection form is displayed showing the connection for the design. The internal structure for each of the adder is required to be specified by the user.

Most of the time the required design is not just one primitive block, it may be a connection of several blocks defined in the module library. Therefore a connection form is created to serve the purpose. Each time the respective entry form is filled, this connection form will be displayed. The form contains the names of the input and output pins for the item (the pins are specified by the user when he/she fills in the entry form), Figure 5.3 is an example of a simple connection form.



```
input connection  :                     output connection :
 1. in1,0,3 mux  pin no.: 7             3. sum,0,3 adder pin no.: 0
 2. in2,0,3 •sidepin no.: 0             4. cout,0,0 •sid pin no.: 0

input connection  :                     output connection :
 5. a0,0,0 adder pin no.: 14            7. db2,0,0 •nect pin no.: 1
 6. db1,0,0 adderpin no.: 19

input connection  :                     output connection :
 8. in1,0,3 •nectpin no.: 3             10. sum,0,3 adderpin no.: 17
 9. in2,0,3 •sidepin no.: 0

input connection  :                     output connection :
 11. in1,0,3 •si pin no.: 0             14. sum,0,3 •nectpin no.: 5
 12. in2,0,3 •si pin no.: 0             15. cout,0,0 •si pin no.: 0
 13. cin,0,0 •si pin no.: 0

input connection  :                     output connection :
 16. in1,0,3 •si pin no.: 0             19. sum,0,3 •nectpin no.: 6
 17. in2,0,3 •nectin no.: 10
 18. cin,0,0 •si pin no.: 0
```

```
----- BLOCK DEFINITION -----
■ adder
■ alu
■ counter
■ interface bus
■ microprocessor
■ connection line
■ has already been defined
```

Figure 5.3 A Simple Connection Form

from (pin number)	to (pin number)
adder (14)	mux (5)
adder (19)	mux (6)
mux (7)	alu (1)
alu (3)	adder (8)
adder (10)	adder (17)

The graphical representation would be as follows (Figure 5.4):

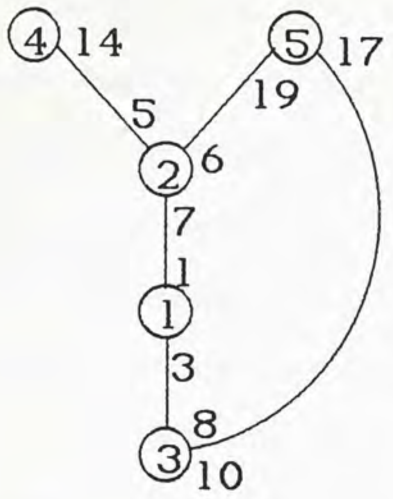
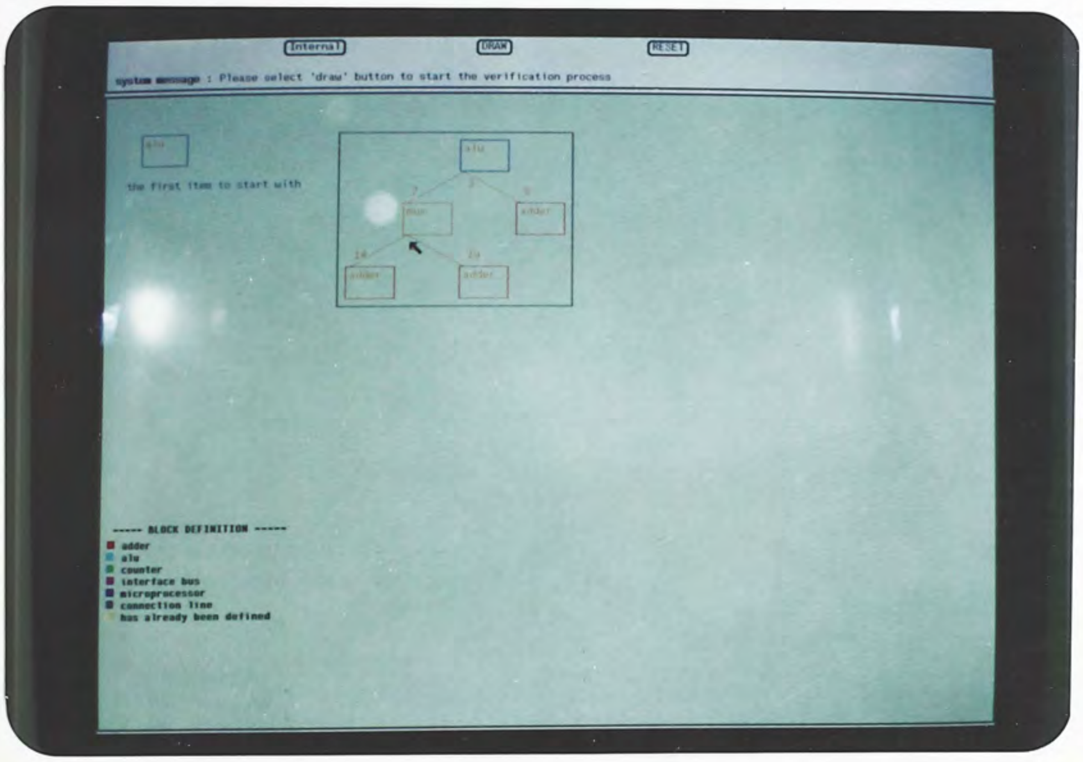


Figure 5.4 The Graphical Representation of the Specified Connection

The user simply specifies the module that a particular pin is connected to, together with the pin number (since the input and output pins for the module have not been specified in this stage, the user can leave it till the later stage). When this form has been filled, the system searches the pins of the item to determine if there is any module connected to them. This searching is breadth-first style, that is, the first connection form will be processed until no more modules are connected to the first item before the second connection form (the form for the next connected module) is started. The block diagram of the specified connection is shown immediately on the screen once the user has made the specifications. This gives a clear idea of the connection that he/she has already made.

The process continues until no more connection is required for either one of the modules. At this stage, the user can respecify the pin number, if required. Figure 5.5 shows the block diagram of the circuit under design when the user has finished the connection specifications. Once all the forms are filled, the system starts storing the data from the connection forms to a data file 'data1'. These data are stored up in a breadth-first manner too. However, in the verification program, it is required that the data passes in a depth-first style. So a process is required to rearrange the data as well as the corresponding data files for the design. Figure 5.6 shows the format for the two different manners.



system message : Please select 'draw' button to start the verification process

Figure 5.5 The Block Diagram of the Circuit under Design

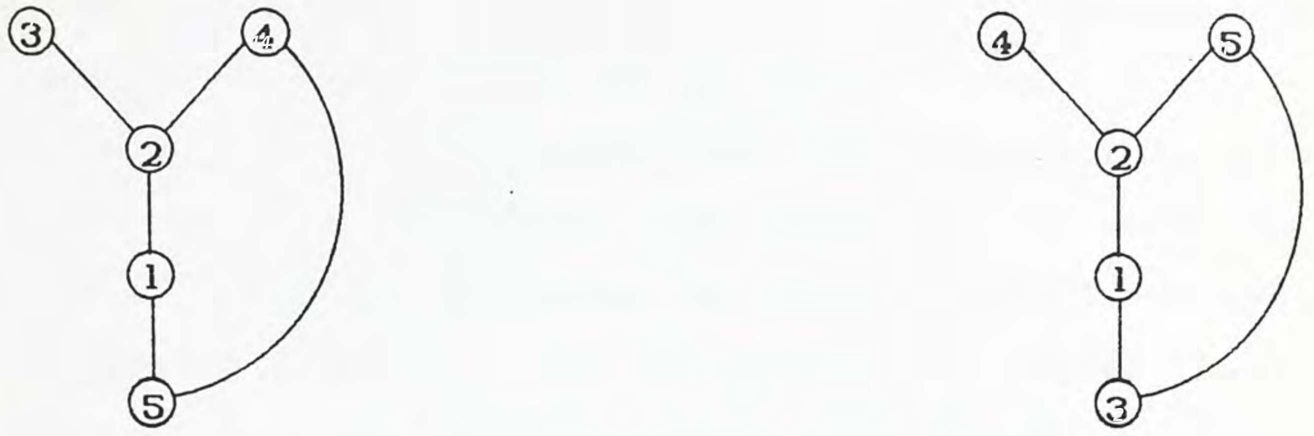


Figure 5.6 The Format of the Two Different Manners

Another point needs to be emphasized is that the connection specified by the user has to be a one-to-one mapping style. Figure 5.7 illustrates the accepted and unaccepted connections for the circuits under design in the system. (Parametric blocks allow virtually an unlimited number and variety of functions to be created. However, for simplicity, only three levels are shown).

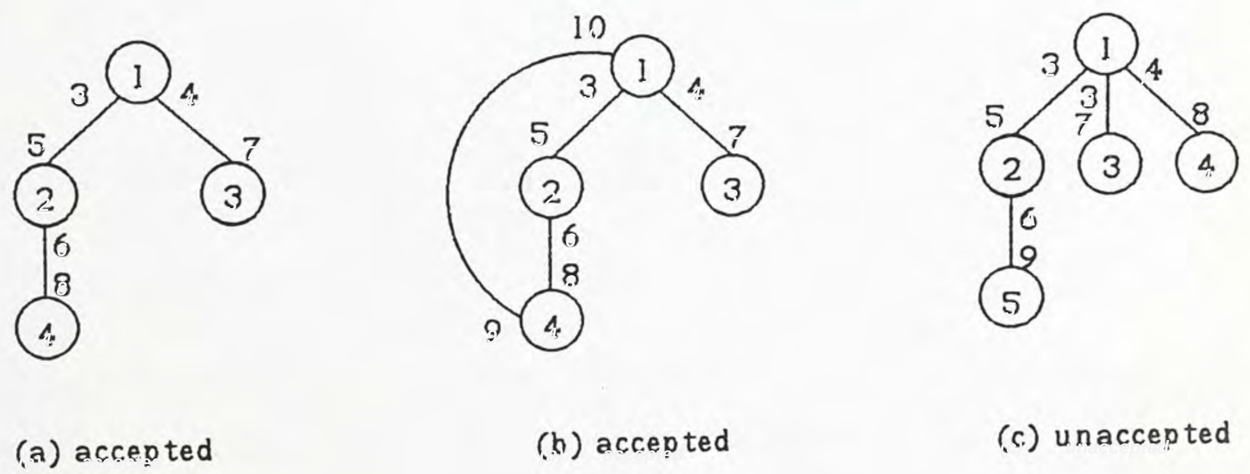


Figure 5.7 The Accepted and Unaccepted Connections

Figure (c) is not allowed since the output pin (3) from node 1 is connected to both node (2) and (3). This restriction is due to the design of the connection form. Each entry in the form allows the user to specify one item only. To solve this problem, i.e., if the user wants the design to be a one-to-many mapping for a particular pin, two methods can be used. Either the user has to explicitly connect that output pin to an amplifier (Figure 5.8), or the user has to repeat that particular pin name (as if it is another separate pin for the item) when specifying the I/O pins for the item. The first solution creates a redundant module in the circuit design, therefore this redundancy will be handled later at the verification stage and will be discussed in section 5.2.2.

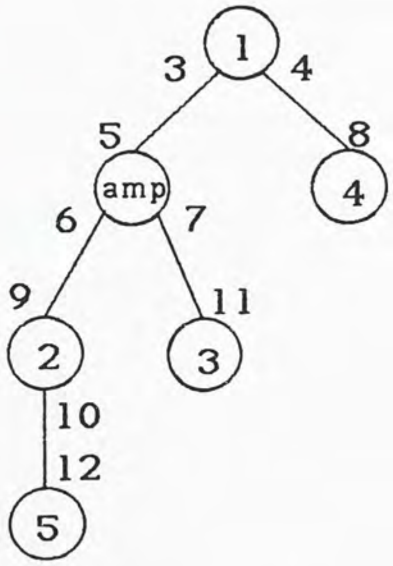


Figure 5.8 The Connections Using an Amplifier

For bi-directional ports, they are classified as the output port in the IO pin specification form. The system will distinguish between them during the verification process.

The next section is a description of the verifier.

5.2 Design Generation (Strategy Rule-Base Sub-System)

As described in chapter 4, all the required rule frames are stored in the rule-base library. The block diagram showing the relationship between the verifier and the libraries is shown in Figure 5.9. The verifier consists of several functional blocks, namely, the individual verifier, the connection verifier and the feedback-path verifier. The functions of each of the blocks will be discussed below.

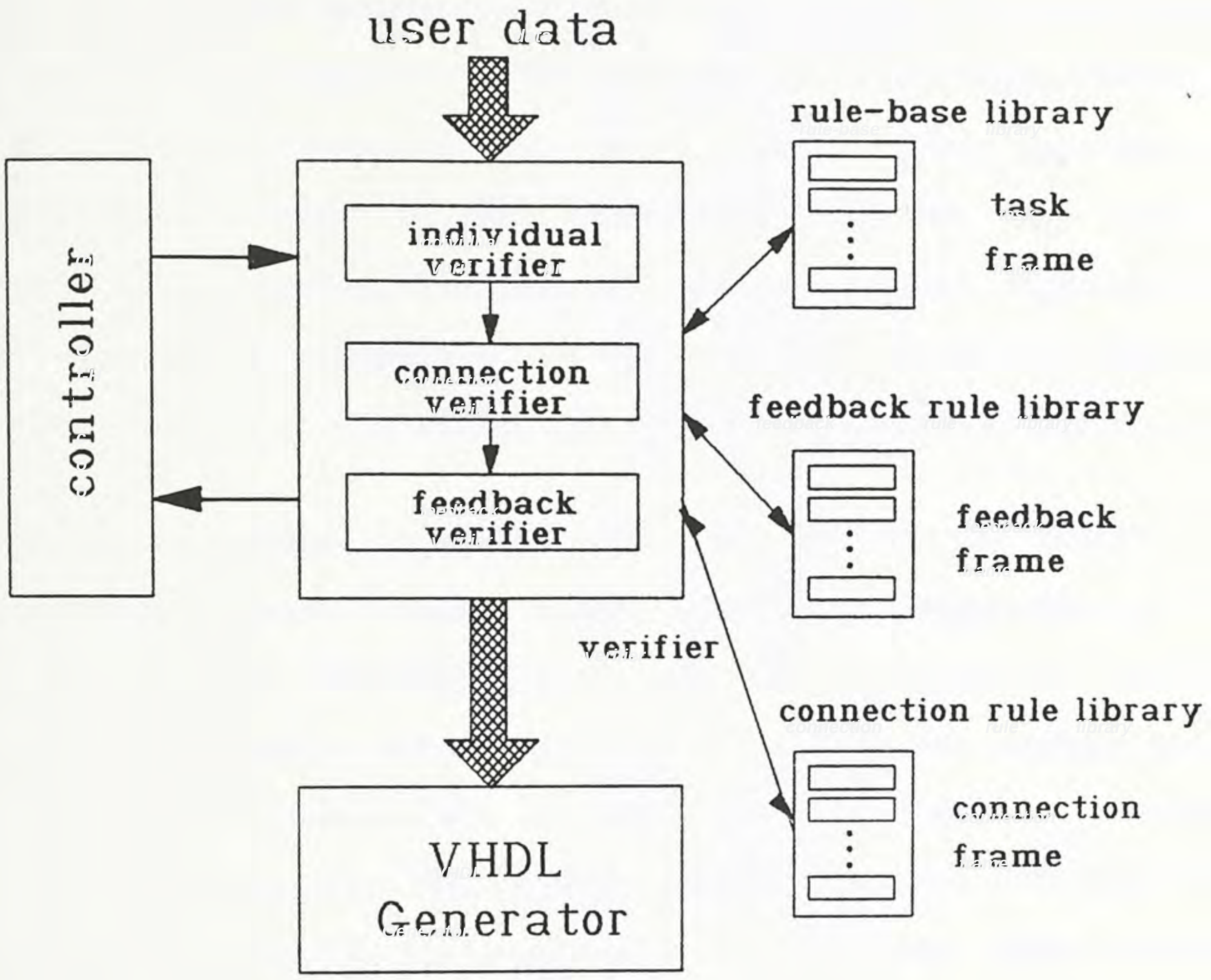


Figure 5.9 the Relationship Between the Verifier and the libraries

5.2.1 Individual Verifier

The rule selection process is data driven. First of all, the item that is going to be designed is passed to the individual verifier. This verifier searches through the rule-base library for a frame which has the frame name matching the design element. Then the consequences of the rule (in the field 'rule' inside the frame) are applied, and the process is repeated until no more rules are available or until a rule explicitly stops the process (this may be due to the occurence of some unrecoverable errors or some queries raised by the verifier). The same example described in section 5.1 is used as an illustration. Since the design type found is an "ALU", the verifier searches the rule-base library for the frame with the task name "ALU". It immediately passes the data (all the information retreived from the form entry described above, which is stored in a data file named as '1') to the respective rule frame for investigation.

This rule frame contains rules that govern the design of a particular item. These rules check the validation of the specifications describing the internal structure of the item. With the example given, to specify an ALU, the system has to check the number of bit slice required for a particular specifications. For the control signals, the system has to check whether there is any duplication in specifying these data. If there is, the verifier stops the process and passes the control back to the editor to display the form entry again for re-specification. Any queries raised by the verifier will stop the verification process until the queries have been answered, then the verifier resumes control and the process continues. If no more rules for that particular item are found, the verifier proceeds to the 'component' field. This component field links up all the components required for that particular name, if any. For each component, the above procedure repeats. If no more components are found, the verification process terminates and the controller passes the control to the connection verifier for connection checking.

5.2.2 Connection Verifier

As the name indicates, the rules in this verifier govern the connection contraints for the circuit under design. Therefore, the data from the connection forms are passed in for examination. Using the example described in section 5.1 again, the connection diagram in Figure 5.4 is displayed once more (Figure 5.10) for easy references.

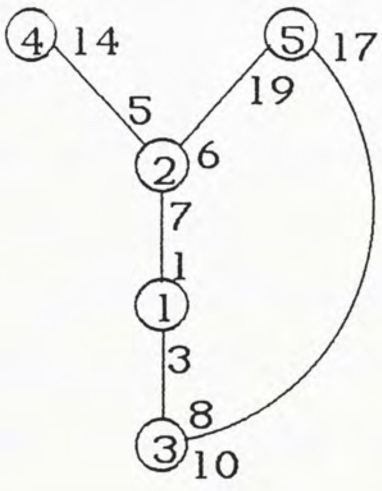


Figure 5.10 An Example of the Connection

The verifier checks the connections from the root of this tree structure first. The order of the checking is in accordance with the pin numbers, in ascending order. This verifier checks whether the two objects can be connected together. There are several rules that govern the general connections. They are :

1. The type of the pair of connection pins must be checked so that none of the following connections would happen :

   (a) I -> I connection

   (b) I -> O connection

   (c) O -> O connection

2. The number of data lines from the output pin should not exceed the number of data lines from the receiver pin. If this is the case, it is assumed that the lines with the lower significant bit number are connected, leaving the lines with the higher significant bit number, for example, if the output pin is specified to be out1,0,3 and the input pin is in1,0,2, then the output line out1,3,3 would be left untouched. A warning message will be displayed. The following (Figure 5.11) is an extraction from the rule-base :

```
% Sour(Dest) = Source (Destination),
% Cs(Cd) = count number for Source item,
% S1(S2) = starting line number for Source (Destination),
%                    '0' ('0') in the case,
% D1(D2) = ending line number for Source (Destination),
%                    '3' ('2') in  this case,
%
checkline(Sour,Cs,S1,S2,Dest,Cd,D1,D2) :-
        S2-S1 > D2-D1,
        writeln('warning : some of the output lines
         are floating'), ...
        plus(S1,D2-D1,S3),
        write(S,'connect_to('), write(S,Sour),
        printf(S,',%d,%s,%d,%d,%d,%d,%d',
         [Cs,Dest,Cd,S1,S3,D1,D2]),

checkline(Sour,Cs,S1,S2,D,Cd,D1,D2) :-!.
```

Figure 5.11 The Sample Rules for connection Checking

3. The number of data lines from the receiver pin should not exceed the number of data lines from the output pin. If this is the case, the error flag will be set since no input of any of the items should be left floating.

The verifier checks each of the items according to the above rules. It starts from the root of this tree structure, in this example, an "ALU". Then for each of the pins, it checks if the pin is connected to any item. With this example, it finds an "mux". The connection verifier stops the process and passes the control to the individual verifier to examine the data for that adder. If nothing goes wrong, the connection verifier resumes control and validates the connection according to the rules. Each of the connections are inserted in the data-base for later use. Any warning or error message found will be displayed on the screen immediately. However, the process will not be terminated until all the items have been examined. The next item to be examined is the adder. In this case, no connection is found; therefore, the verifier goes one level back and examines the pin number 6 for the "MUX". The searching algorithm is a depth-first search. To check the redundancy of the amplifier introduced in section 5.1, the system makes use of the connection list stored in the database, this is illustrated in the following program segment (Figure 5.12).

```
handle_amp :-
        connect_to(Source, Sc, amp,
                C, S1, S2, D1, D2, Frompin, Topin_amp),
        connect_to(amp, C, Dest,
                Sd, S3, S4, D3, D4, Frompin_amp, Topin),
        assert(connect_to(Source, Sc, Dest, Sd,
                S1, S2, D3, D4, Frompin, Topin)),
        fail.
handle_amp :- !.
```

Figure 5.12 Program Segment for Handling the Redundancy of the Amplifier

This amplifier will not be considered as the component of the new module. Once the process is finished, the controller regains control and tests the feedback flag. If it is set, the control will be passed to the feedback verifier, otherwise, the data files and the analytical results will be passed to the VHDL generator next to the verifier in Figure 4.4.

5.2.3 Feedback-Path Verifier

In checking for the feedback connection, the three rules described in section 5.2.2. are also applicable. Moreover, one more important criterion has to be examined, this is the timing problem. A sequential circuit could easily be unstable if the timing constraints have not been considered. To check the feedback path, the followings are some of the rules :

1. if the output of an item is connected to itself, the verifier will insert a register in between the two ports, as shown in Figure 5.13.

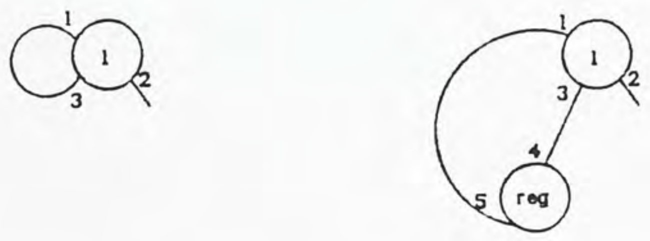


Figure 5.13 A Register is inserted between the Feedback Path

The coding is illustrated below (Figure 5.14) :-

```
feedback_handler(no_Stream2) :- !.
feedback_handler(yes, Stream2) :-
        see(feedback),
        read(Item), feedback_path(Item, Stream2), close(Stream2),
        !, seen.

feedback_path(end,S) :-!.
feedback_path(Item1,S) :-
        read(Frompin), findcounto(Item1, C1, Frompin),
        read(Data), read(Topin),
        findcounti(Data,C2,Topin),
        node(Item1, C1, V1), node(Data, C2, V2),
        timing(C1, C2, V1, V2),
        asserta(connect_to(Item1, C1, Data, C2, Frompin, Topin)),
        open('fback.dat', append, S2),
        fdalli(Data, C2, S2), close(S2), !,
        read(Item2),
   feedback_path(Item2, S).
```

Figure 5.14 Program Segment for Handling the Feedback-Path

2. if the feedback path is not allowed, it would be deleted from the connection list.

This verifier will not set any error flag because it will try to modify the feedback connections so that the circuit under design can still be a valid one. When the verifier finishes the process, the whole verification process is considered finished and the controller passes the control to the VHDL generator to generate the layout description.

5.3 VHDL Generation

Since most of the silicon compilers to date accept the user input through the use of a hardware description langauge, this method of describing the behaviour of the system is also included here. To translate a graphical inputs of a circuit design into a hardware description language, the following method is adopted.

It is understood that each design specified by the user is a combination of the primitive logic circuits contained in the module library in the system. Since the internal connections of a primitive circuit (for example, a full adder, an cla) is fixed, these parts can be shared among the circuits under design. Each of them are stored in a frame specified as 'hardware' type in the hardware-module library. These frames (described in chapter 4) contain the behavioral or structural description of the corresponding circuits with the respective frame name. Each primitive has a VHDL frame,therefore each design would have a set of variables for each different primitives. The VHDL generator analyses the connection list passed in. For each item from the list, it searches the respective frame, creates a new VHDL description with the arguments associated with the items.

Take an adder as an example. To generate the VHDL description of a specified adder, the generator carries out the following steps :

Generate the body of an adder (i.e., the fixed part from the hardware frame), then from the argument list, determine :

```
if (tristate)
    call tristate_gen_buf(bits);
    insert into the new_vhdl;

if (zero_detect)
    call zero_detect_gen_buf(bits);
    if (output result latch)  call latch(bits);
            paste into the output of the new_vhdl;
    paste into the output of the new_vhdl;

if (input operand latch)
    call latch(bits);
    paste into the input of the new_vhdl;
```

A similiar procedure will apply to other modules. With the VHDL descriptions, the layout circuit can be easily generated by passing the descriptions through the VHDL compiler and the simulation can also be carried out using this coding. The implementation of the VHDL generator has not been finished yet, however, the structure of this function block is well designed so that it would not be too difficult in building the generators.

5.4 Final Output

Charts and diagrams make data easy to be recognized and understood. Therefore when the data has been verified, a block diagram of the designed logic circuit is displayed. The VHDL description of each of the blocks can be examined by selecting that component in the display. The type of blocks is distinguished by the color used. This new module will be inserted into the module-library. This involves inserting its corresponding task frame and rule frame into the strategy rule-base (Figure 5.15) :

```
    savenewrules(Name) :-
        rulename(Rule_name),
        open(Rule_name, write, S), setOutput(S),
        ...,
        close(S).

    newtaskframe(Framename, Newtaskname) :-
        open(Framename, append, S), setOutput(S),
        ...,
        close(S).
```

Figure 5.15 Program Segment

and also its component list is also inserted into the 'component' field of the task frame. Once the system is reset, the menu of the CUISIC primitives will be updated.

5.5 DISCDA : Graphical Editor

It is understood that graphics are fast becoming indispensible to logic design. Therefore a graphical editor is included for the sake of completeness. The role of the editor is to capture the designer's view of the digital circuit at the functional level. The design specifies the circuit elements (the primitive elements in the primitive circuits library) and the interconnections between them. The outputs of the graphics editor are two descriptions of the circuit, one for the simulator, and the other for the layout designer, each in the appropriate format. The implementation work is carried out by the under-graduate students group, details of the work can be found in [internal paper, 1988].

6. DESIGN METHOD EXAMPLES

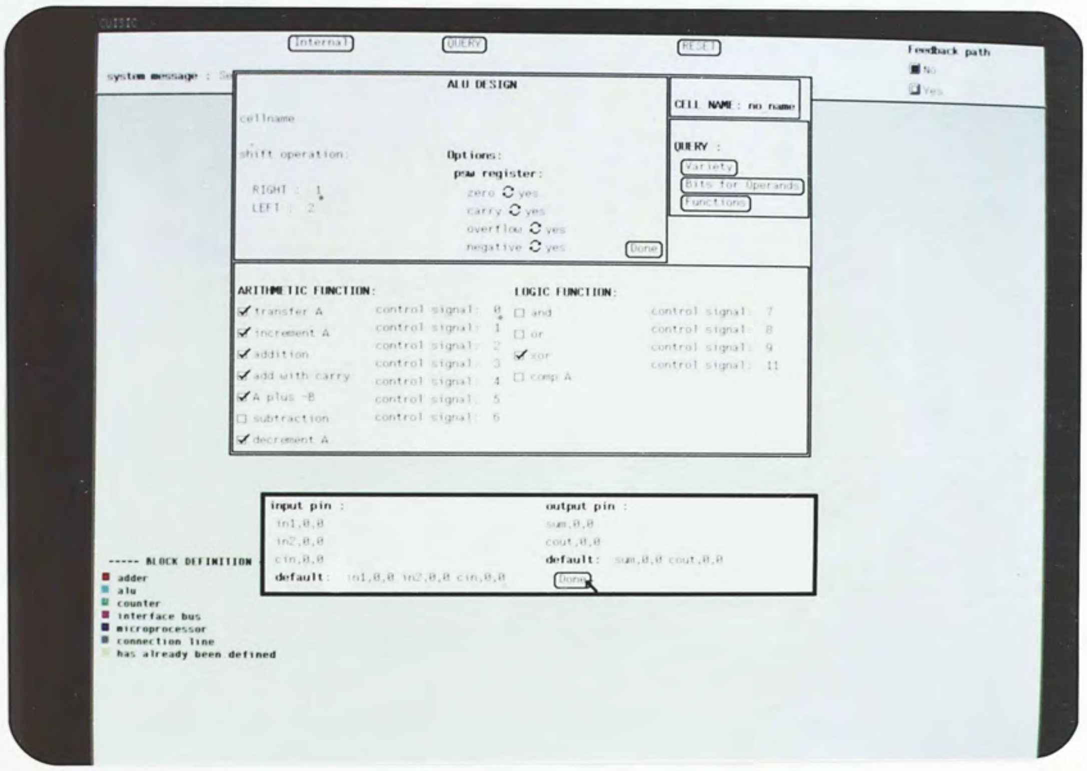
6.1 ALU

To illustrate the working of the system, two examples are given below. They are presented with program segments to clarify various points. The first one is an example which the design item is specified by its function. When the user has selected 'multiplication' as the design function from the start up window, the system searches the task frame with name 'multiplication', it finds that the first rule contains a pointer pointing to a style frame, which contains two choices : "shift with add" and "Carry Save Adder". Therefore the system prompts the user for a selection. If, in this case, the user chooses "shift with add", the second rule in the task frame will be searched, it finds the component list which contains an 'ALU', therefore a form for the ALU will be displayed. This contains the property list of the selected item. Figure 6.1 shows that the user has specified a 4-bit ALU with zero detect, negative detect, overflow detect, carry out detect, output latch, and the tristate output options. The implementation of the logic circuitry is based on a carry lookahead adder. If the user has any queries in filling the form, he/she may invoke the enquiry rule-base by selecting one of the items under the label "Queries". For example, if the user cannot determine which implementation he/she is going to use, he/she may select the button "Queries". The system then searches in the enquiry rule-base for the style-frame containing

6. DESIGN METHOD EXAMPLES

6.1 ALU

To illustrate working of the system, two examples are given below. They are presented with program segments to clarify the points. The first one is an example which the design item is specified by its function. When the user has selected 'multiplication' as the design function from the start up window, the system searches the task frame with name 'multiplication', it finds that the first rule contains a pointer pointing to a style frame, which contains two choices : "shift with add" and "Carry Save Adder". Therefore the system prompts the user for a selection. If, in this case, the user chooses "shift with add", the second rule in the task frame will be searched, it finds the component list which contains an 'alu', therefore a form for the alu will be displayed. This contains the property list of the selected item. Figure 6.1 shows that the user has specified a 4-bit ALU with zero detect, negative detect, overflow detect, carry out detect, output latch, and the tristate output options. The implementation of the logic circuitry is based on a carry lookahead adder. If the user has any queries in filling the form, he/she may invoke the enquiry rule-base by selecting one of the items under the label "Queries". For example, if the user cannot determine which implementation he/she is going to use, he/she may select the button "Queries". The system then searches in the enquiry rule-base for the style-frame containing the name "adder_imple", as shown in Figure 6.2. The specified type is "diagram", therefore the diagram for the implementation is shown on the screen, as in Figure 6.3.



CUISIC

Internal | QUERY | RESET

Feedback path
■ No
☐ Yes

system message : Se

ALU DESIGN

cellname

shift operation:

RIGHT : 1
LEFT : 2

Options:
psw register:
zero ↻ yes
carry ↻ yes
overflow ↻ yes
negative ↻ yes

Done

CELL NAME : no name

QUERY :
Variety
Bits for Operands
Functions

ARITHMETIC FUNCTION:

✓ transfer A	control signal: 0	☐ and	control signal: 7
✓ increment A	control signal: 1	☐ or	control signal: 8
✓ addition	control signal: 2	✓ xor	control signal: 9
✓ add with carry	control signal: 3	☐ comp A	control signal: 11
✓ A plus -B	control signal: 4		
☐ subtraction	control signal: 5		
✓ decrement A	control signal: 6		

LOGIC FUNCTION:

input pin :
in1,0,0
in2,0,0
cin,0,0
default: in1,0,0 in2,0,0 cin,0,0

output pin :
sum,0,0
cout,0,0
default: sum,0,0 cout,0,0
Done

----- BLOCK DEFINITION
🟥 adder
🟦 alu
🟩 counter
🟪 interface bus
🟦 microprocessor
⬛ connection line
⬜ has already been defined

Figure 6.1 Form Entry for an ALU

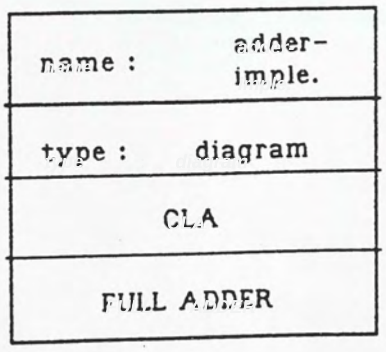
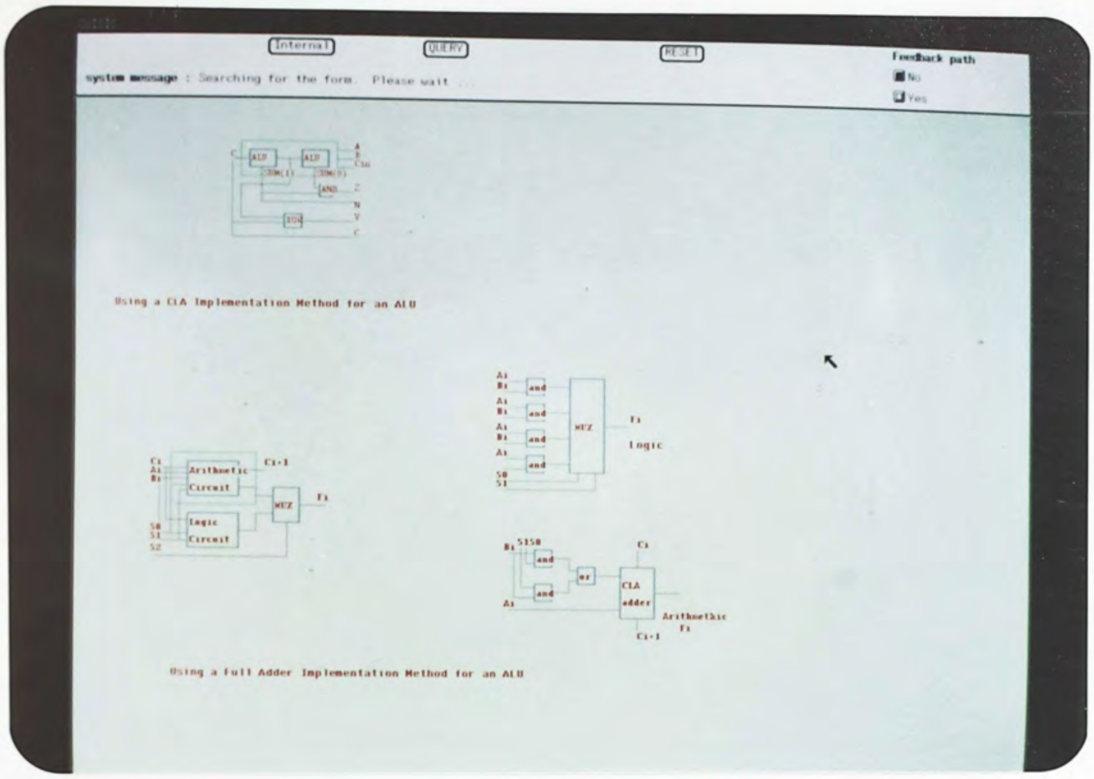


Figure 6.2 Style Frame with Name="adder imple"

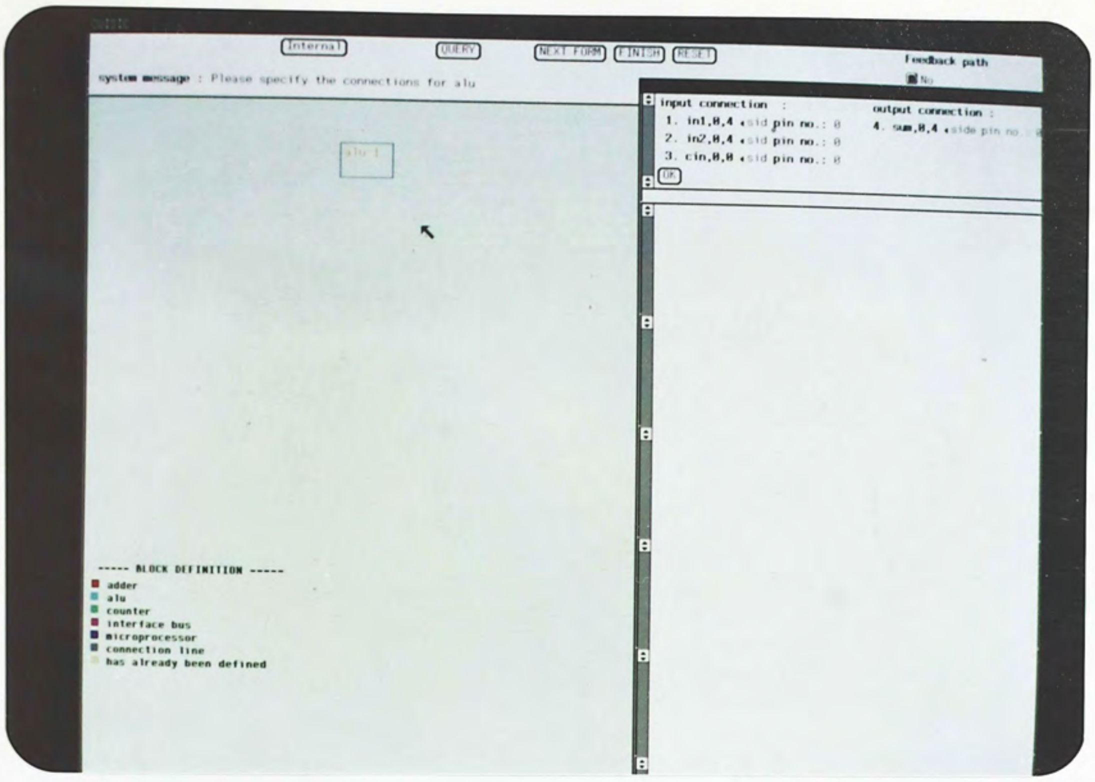

system message : Searching for the form. Please wait ...

Feedback path
[✓ No] Yes

Figure 6.3 the Two Different Implementations for an ALU

Once the form is filled, the system searches for its component list to see if the design is hierarchical. In this case, an adder is found. Once again, the system searches the form entry library to get the required form.

When the form is filled. A connection form for the ALU is displayed with the input and output pin names specified in the form entry (Figure 6.4).



system message : Please specify the connections for alu

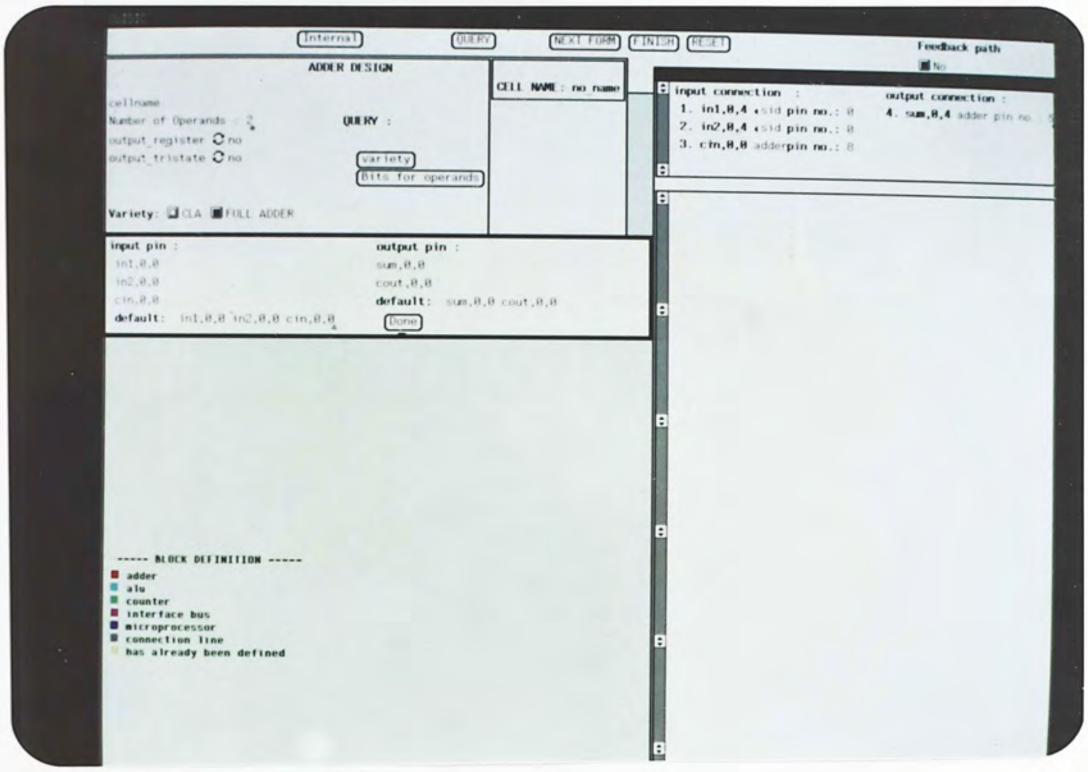
input connection :
1. in1,0,4 ◂sid pin no.: 0
2. in2,0,4 ◂sid pin no.: 0
3. cin,0,0 ◂sid pin no.: 0

output connection :
4. sum,0,4 ◂side pin no. : 0

----- BLOCK DEFINITION -----
🟥 adder
🟦 alu
🟩 counter
🟪 interface bus
🟦 microprocessor
⬛ connection line
⬜ has already been defined

<u>Figure 6.4 the Connection Form for the ALU</u>

Only one ALU is necessary for the mulitplication process. Therefore the system waits to see if the user wants to connect any modules to the ALU. Figure 6.5 shows an example of the connection.



```
ADDER DESIGN                                CELL NAME : no_name

cellname
Number of Operands : 2            QUERY :
output_register  no
output_tristate  no

Variety: ☐ CLA  ■ FULL ADDER

input pin :                        output pin :
in1,0,0                            sum,0,0
in2,0,0                            cout,0,0
cin,0,0                            default:  sum,0,0 cout,0,0
default:  in1,0,0 in2,0,0 cin,0,0

Feedback path
■ No

input connection  :                output connection :
1. in1,0,4 sid pin no.: 0          4. sum,0,4 adder pin no.: 5
2. in2,0,4 sid pin no.: 0
3. cin,0,0 adderpin no.: 8

----- BLOCK DEFINITION -----
🟥 adder
🟦 alu
🟩 counter
🟪 interface bus
🟦 microprocessor
⬛ connection line
⬜ has already been defined
```

<u>Figure 6.5 The Connection Form and the Entry Form for the Adder</u>

When the form is filled, the connection form for the adder will be displayed. Since now none of the modules is connected to the adder, the design is considered complete. The data then passes to the verifier for verification.

These data include:

i. the number of bits in the ALU,

ii. optional left/right shift function,

iii. optional registers at the ALU output,

iv. the desired ALU functions,

v. the corresponding control signals and

vi. the data for the internal adder as well as

vii. the external connected adder.

All the processed data will be stored in a file called 'datalu4' as well as in the information data-base inside the system.

The structure of the design is based on a 4-bit-slice architecture, and therefore with the specified bit number, the system has to determine how many bit-slice structure that is going to be used, the following is the program segment for illustration (Figure 6.7) :

```
data(data_bus, Nbits),
Lbits is 4,
divides(Nbits, Lbits, Mblits, Dummy),
alu(Mbits, Dummy, Calu),
write('you need'),
write(Calu),
writeln(' 4-bit slice alu'),
```

Figure 6.7 Program Segment for the Design of an ALU

For the optional functions and registers, the system stores the data as facts for later references at the VHDL generation part :

```
    shift_signal(alu,1,right,left,0,1).
```

These data are checked for any duplication before the following processes can be carried out. If there is a repeat assignment for the ALU control signals, the form entry will be displayed once again for clarification. For the ALU signals, the system has to determine which of them has not been selected (the predicates no_alufun(P) indicates this), then gather them together and delete them from the default signal list (Figure 6.8).

```
    solutions([P2,Sig2],alu_signal(arith,P2,Sig2),S2),
    solutions([P3,Sig3],alu_signal(logic,P3,Sig3),S3),
    append(S2,S3,S4),
    solutions(P,no_alufun(P),S1),
    delete1(S1,[0,1,2,3,4,5,6,7,8,9,10],Tmp1),
    convert([],Tmp1,L),
```

Figure 6.8 The Program Coding for Handling ALU Signals

To find out the number of bits for the micro-instruction word, the maximum code for the control signal has to be found.

```
    sort(L4,Sorted),
    reverse(Sorted, [H|Backlist]),
    loop(H,1,Len,M).
```

The input and output pins for the ALU are stored as follows:

```
    iport(alu,Count,[[1,[in1,0,3]],[2,in2,0,3]],[3,cin,0,1]]].
    oport(alu,Count,[[4,[sum,0,3]],[5,[cout,0,1]]].
          where Count = 1 in this case.
```

Now the component list is examined by the following codes :

```
    complist(X) :-
          component(X,L),
          partition(L).


    partition([]).
    partition([X|Y]) :-
          count(X,C),
          selection([X,C]),!,
          partition(Y).
```

The data is examined as follows :

1. Since the number of bits is determined by the ALU, there is no need to verify this again.

2. asserts the choice of the adder into the data file 'datalu'. (Since this adder is part of the internal structure of alu1, the data is part of the alu1 as well).

   choice(adder,cla).

3. the input and output pins are stored as :

   iport(adder,[1,[in1,0,3]],[2,[in2,0,3]],[3,[cin,0,1]]].

   oport(adder,[4,[sum,0,3]],[5,[cout,0,1]]].

Notice that the pin number remains from 1 onward since this is the internal structure of the ALU, not an external adder connected to the ALU.

Next the system examines the component list for the adder.

component(adder,[]).

Since the component list is empty, the individual verification has finished. The controller then passes the control to the connection verifier :-

```
open(datconnect, append, Stream2),
see(iodataf),
any_connection(no, S, H, Stream2, 0),
seen, ... .
```

'iodataf' is a file that stores the data from the connection forms.

For the connection list, the verifier considers the connection for the input pins first before going to the output pins. The rules extracted from the connection verifier are given below.

```
any_connection(_,outside,H,S,_) :-
     . . . , !.
any_connection(Item,H,S,Tab) :-
     ...,
     append(T,[Item],Newcomp), ... ,
     count(Item,C1),
     count(vertice,V),
     plus(V,1,V1),
     .
     .
     iport(Item,C1,I),
     input_connect(Item,I,S,Tab),
     .
     .
     oport(Item,C1,O),
     output_connect(Item,O,S,Tab),
     .
     .
     .
```

The verifier examines the connection so that no violation to the rules (described in session 5.2.3) occurred :-

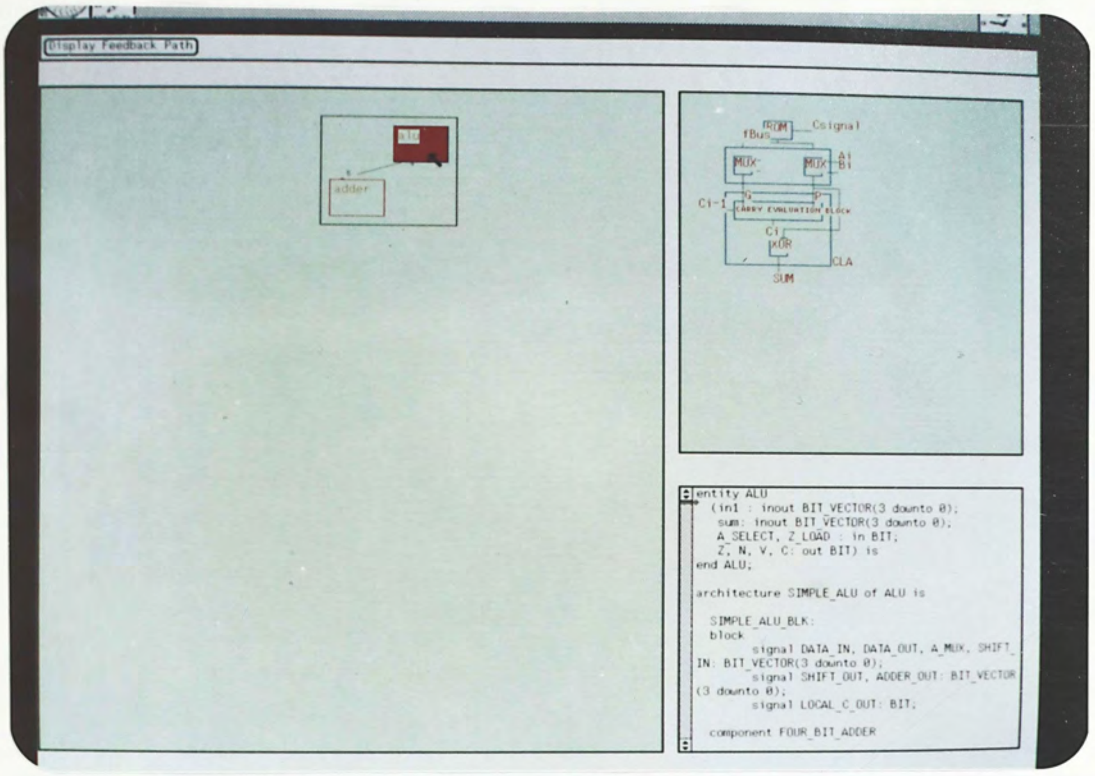
```
verify(Sour,Cs,Dest,Cd,[Pin1,Start1,End1],[Pin2,Start2,End2]) :-
     checkline(Sour,Cs,Start1,End1,Dest,Cd,Start2,End2),
     ....


checkline(S,Cs,S1,S2,D,Cd,D1,D2) :-
     S2-S1 > D2-D1,
     writeln('warning: ...').


checkline(S,Cs,S1,S2,D,Cd,D1,D2) :-
     S2-S1 < D2-D1,
     writeln('warning: ...').


checkline(S,Cs,S1,S2,D,Cd,D1,D2) :- !.
```

The node number has been assigned to each of the items so that connection can be easily referenced. With the example given, there is only one connection for the output pin number 4 of the ALU, and a feedback path connected from pin number 8 to pin number 3, therefore the output_connect predicates are invoked. The connection is stored in the data file 'datconnect'. The connection process continues for the 'adder', since none of the pins are connected to any other items, the process terminates.

When the system examines the connection, it automatically identifies the feedback path and groups them together. When the connection verifier has finished the process, these data are passed to the feedback verifier for validation check. The segment of the coding is as follows :

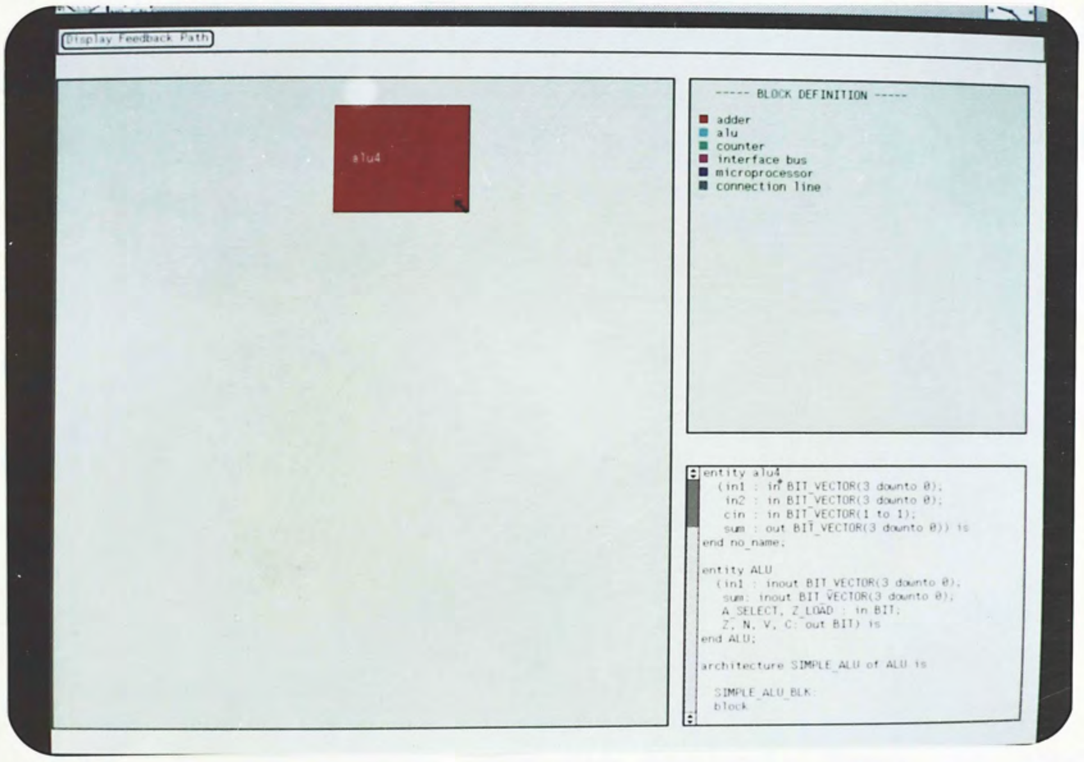
```
    feedback_handler(no, Stream2) :-
        ... ,
        feedback_path(Item, S),
        ... , seen.


    feedback_path(end,S) :-!.
    feedback_path(Item1,S) :-
        read(Frompin),
        findcounto(Item1,C1,Frompin),
        read(Data), read(Topin),
        findcounti(Data,C2,Topin),
        node(Item1,C1,V1), node(Data,C2,V2),
        timing(C1,C2,V1,V2),
        ...,
        read(Item2),
        feedback_path(Item2,S).
```

First of all, the verifier has to identify the node numbers as well as the count numbers in which the pair of connected pins belong to. This is not difficult since the input ports and the output ports of all the items are in the data base. Each port list has the count number assiocated with it. With the count number known in advance, it is quite easy to find out the node



```
entity ALU
  (in1 : inout BIT_VECTOR(3 downto 0);
   sum: inout BIT_VECTOR(3 downto 0);
   A_SELECT, Z_LOAD : in BIT;
   Z, N, V, C: out BIT) is
end ALU;

architecture SIMPLE_ALU of ALU is

  SIMPLE_ALU_BLK:
  block
        signal DATA_IN, DATA_OUT, A_MUX, SHIFT_
IN: BIT_VECTOR(3 downto 0);
        signal SHIFT_OUT, ADDER_OUT: BIT_VECTOR
(3 downto 0);
        signal LOCAL_C_OUT: BIT;

  component FOUR_BIT_ADDER
```

(a)



----- BLOCK DEFINITION -----

🟥 adder
🟦 alu
🟩 counter
🟪 interface bus
⬛ microprocessor
⬛ connection line

```
entity alu4
  (in1 : in BIT_VECTOR(3 downto 0);
   in2 : in BIT_VECTOR(3 downto 0);
   cin : in BIT_VECTOR(1 to 1);
   sum : out BIT_VECTOR(3 downto 0)) is
end no_name;

entity ALU
  (in1 : inout BIT_VECTOR(3 downto 0);
   sum: inout BIT_VECTOR(3 downto 0);
   A_SELECT, Z_LOAD : in BIT;
   Z, N, V, C: out BIT) is
end ALU;

architecture SIMPLE_ALU of ALU is

  SIMPLE_ALU_BLK:
  block
```

(b)

Figure 6.8 The Final Display on the Screen

6.2 MICRO-PROCESSOR

To illustrate how the program works in a hierarchical sense, a design method for a complicated example is shown in this section to clarify the design method. Consider the schematic diagram in Figure 6.9.

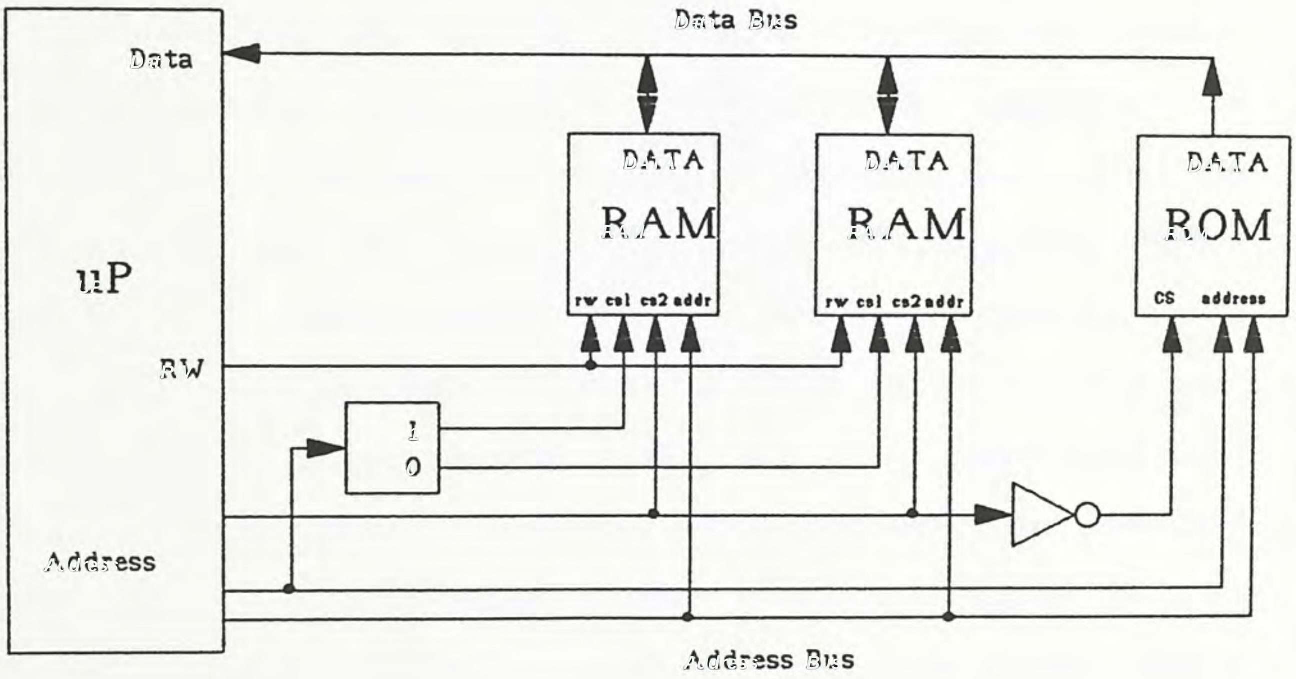


Figure 6.9 A Simple Microcomputer

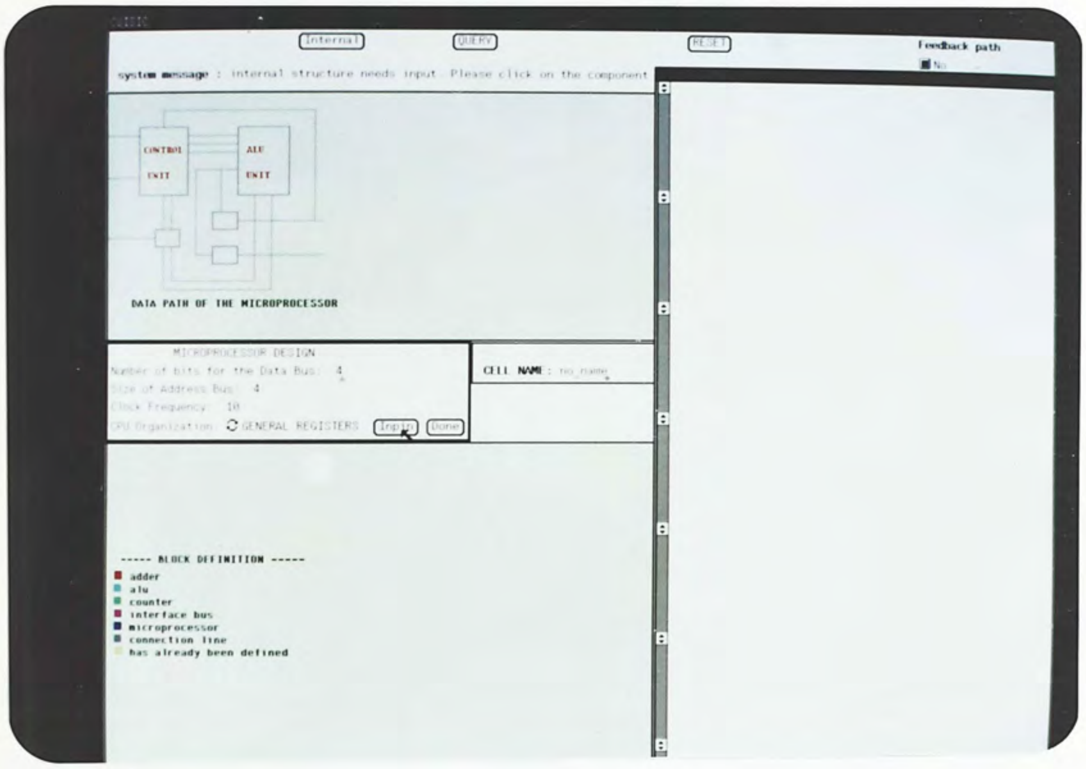
If the user wants to design a micro-computer, he may select the name under the 'Item name' and ask for help from the system by selecting the 'Query' button in the panel. The implementation of a simple microcomputer involves several components, including a microprocessor, a decoder, two RAM chips, a ROM, and an number. Besides calling the checklineno predicates to verify the connection of the pair of pins, the verifier also has to determine whether the feedback path is directly connected back to itself. If so, a register will be inserted in between the two nodes. This verification is done in the following predicate :

```
    timing(I1,I2,V1,V2) :-
         I1 =:= I2,
         assert(register(V1,V2)).
```

If no error flag has been set during the verification process described above, the VHDL generator will generate the descriptions for the ALU and the adder. The final version of the block diagram for the design will be shown on the screen (Figure 6.8(a) and (b)). These descriptions can easily be refered to by the user if he/she simply selects the block on the diagram that he/she is interested in. All the data files concerning the design would be stored under the directory with name specified early in the 'ALU' form entry (in this case, it is called 'ALU-4'), the design will be treated as a new module in the module-library later when the system is invoked again.

inverter. There are five different kinds of components involved. It is quite obvious to choose the 'microprocessor' as the start off item. Therefore the system will reply:

system message : select microprocessor please



system message : internal structure needs input. Please click on the component

Feedback path
[✓ No]

DATA PATH OF THE MICROPROCESSOR

MICROPROCESSOR DESIGN
Number of bits for the Data Bus: 4
Size of Address Bus: 4
Clock Frequency: 10
CPU Organization: GENERAL REGISTERS

CELL NAME : no_name

----- BLOCK DEFINITION -----
- 🟥 adder
- 🟦 alu
- 🟩 counter
- 🟪 interface bus
- 🟦 microprocessor
- ■ connection line
- ⬜ has already been defined

<u>Figure 6.10 The Entry Form for a Micro-Processor</u>

Figure 6.10 is the entry form of a micro-processor. The block diagram of a (Figure 6.11) simplified micro-processor is displayed on the screen for reference.

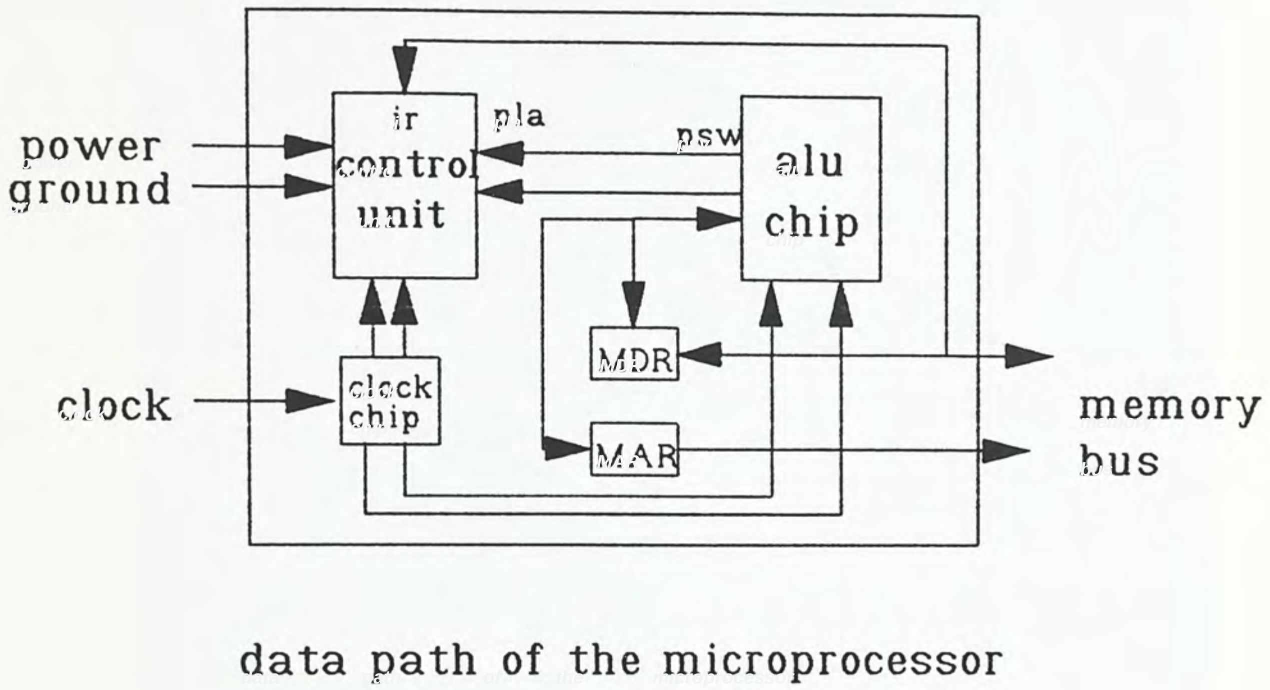
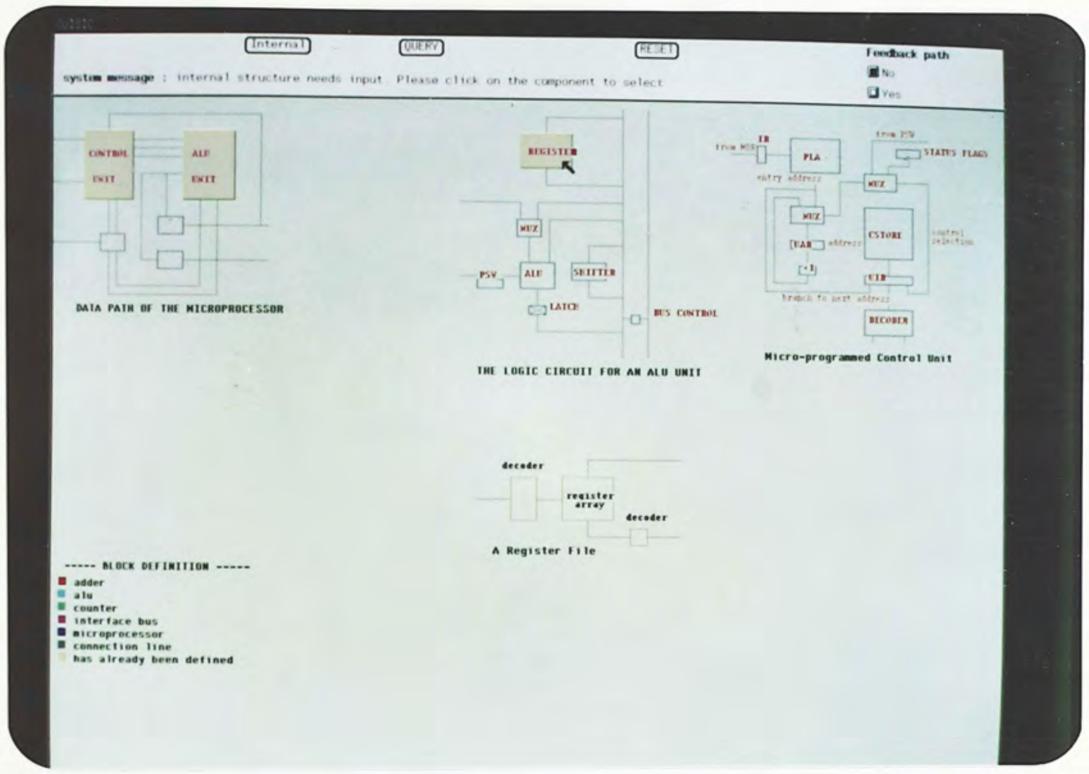


Figure 6.11 A Simplified Micro-Processor

It basically consists of three units : the ALU unit, the control unit and the timing unit. Once the form is completed, the system searches for its component lists, and the diagrams for each of the units will be displayed (Figure 6.12).



system message : internal structure needs input. Please click on the component to select

Feedback path
- [✓ No]
- Yes

----- BLOCK DEFINITION -----
- 🟥 adder
- 🟦 alu
- 🟩 counter
- 🟪 interface bus
- 🟦 microprocessor
- 🟩 connection line
- ⬜ has already been defined

Figure 6.12 The Components of the Micro-Processor are Displayed

The forms for the units will only be invoked if the user selects that units. This is different from the alu in which the entry form is invoked by the system. This has an advantage that the user can have a clear idea of which form he/she is filling. This effect will only be significant if the design is complex (with the alu, there is no need to display the block diagram of its internal structure).

6.2.1 ALU Unit

The ALU chip consists of the register file, and the ALU. The number of registers in the register file is determined by the user. Therefore the size of the register address bus can be calculated by the system. The size of the register file chosen by the user will be stored in the data base as:

```
reg_array :-
        register(Num),
        loop(Num,1,Count,1), ...,
        assert(register(Num)),
        assert(reg_size(register,Count)), ... .
```

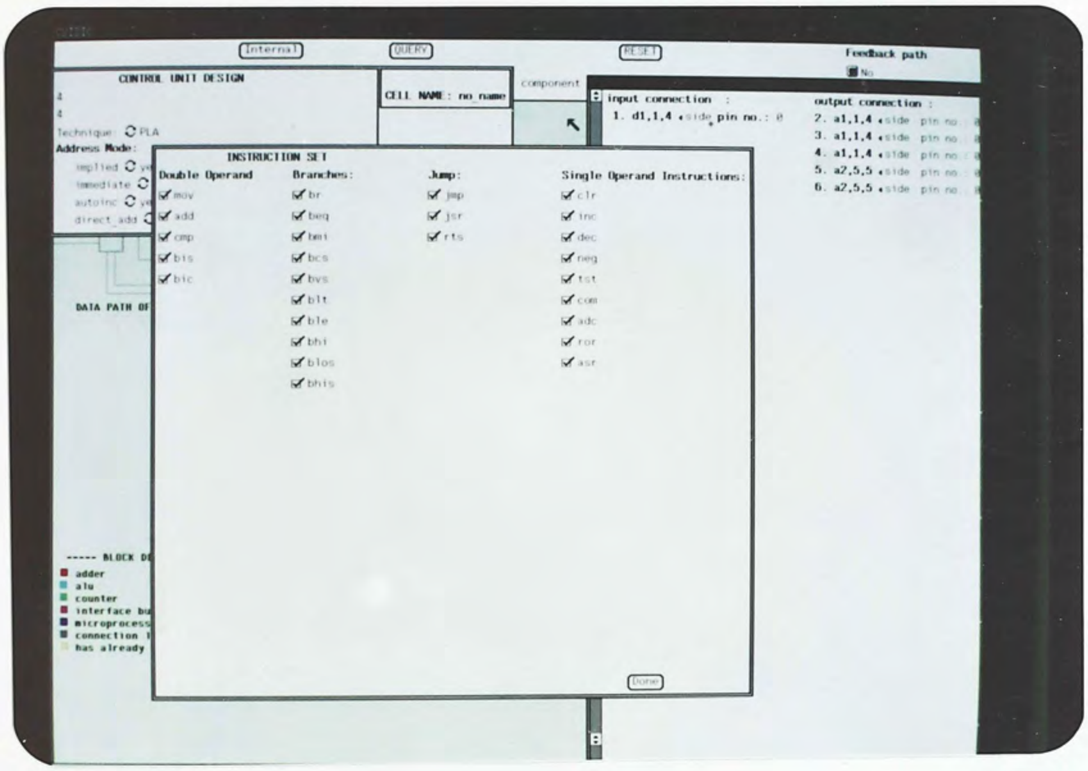
The ALU has been illustrated in the previous section, so it will not be repeated.

6.2.2 Control Chip

The control chip controls the control path of a micro-processor. The system allows the user to specify the size of the microinstruction register, its coding, and to specify the micro-routines for the instruction set, which is again specified by the user.

6.2.2.1 Micro-Instruction

The micro-instruction set is based on the PDP-11, the user can select the required signals from the set (Figure 6.13).



input connection :
1. d1,1,4 side pin no.: 0

output connection :
2. a1,1,4 side pin no
3. a1,1,4 side pin no
4. a1,1,4 side pin no
5. a2,5,5 side pin no
6. a2,5,5 side pin no

INSTRUCTION SET

Double Operand	Branches:	Jump:	Single Operand Instructions:
✓ mov	✓ br	✓ jmp	✓ clr
✓ add	✓ beq	✓ jsr	✓ inc
✓ cmp	✓ bmi	✓ rts	✓ dec
✓ bis	✓ bcs		✓ neg
✓ bic	✓ bvs		✓ tst
	✓ blt		✓ com
	✓ ble		✓ adc
	✓ bhi		✓ ror
	✓ blos		✓ asr
	✓ bhis		

Figure 6.13 the Form for the Micro-Instruction Set

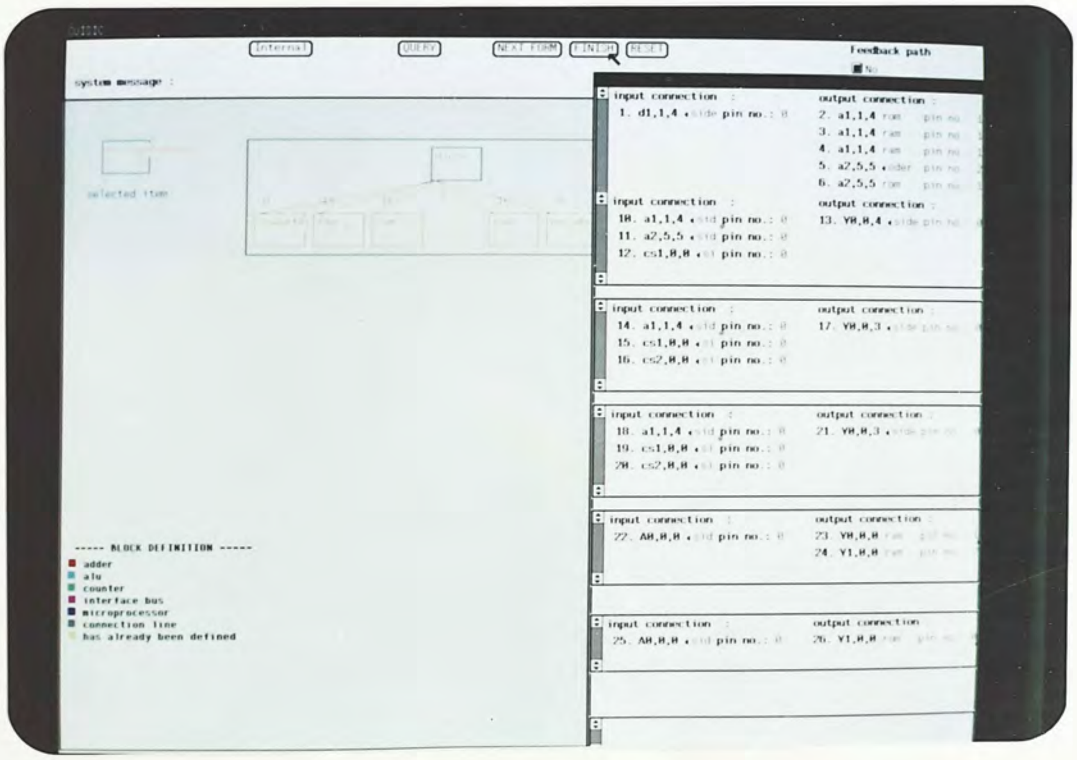
6.2.3 Clock Chip

This component will not be discussed since it involves the transmission of data which would be part of the job of the timing verifier. This timing verifier has not been considered yet.

6.2.4 The Connection Details

If each part of the micro-processor has been specified, a connection form will be displayed. It requires the user to specify the connection for the micro-processor. Again if the user does not know how to continue, he/she may select the 'Query' button again for advice. This time the system message will be:

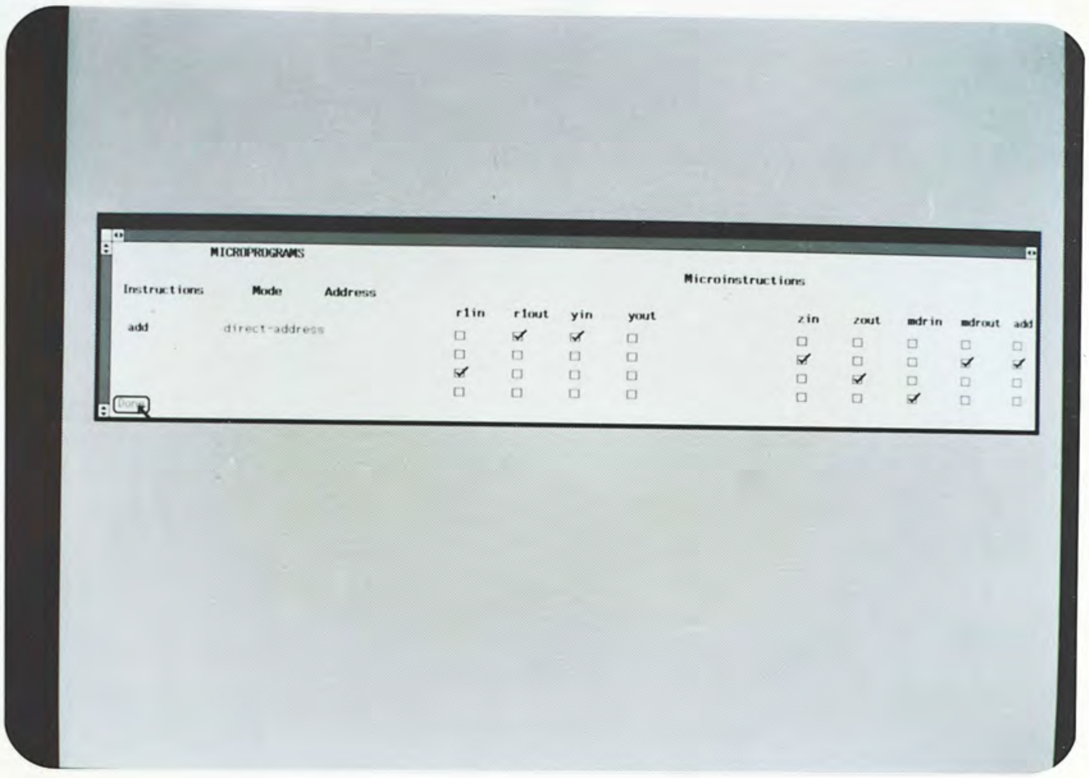
system message : you need some memory unit : RAM and ROM attached to the address bus



```
system message :

input connection  :                output connection :
 1. d1,1,4 .side pin no.: 0          2. a1,1,4 rom   pin no
                                     3. a1,1,4 ram   pin no
                                     4. a1,1,4 ram   pin no
                                     5. a2,5,5 .adder pin no
                                     6. a2,5,5 rom   pin no

input connection  :                output connection :
 10. a1,1,4 .std pin no.: 0         13. Y0,0,4 .side pin no
 11. a2,5,5 .std pin no.: 0
 12. cs1,0,0 .sl pin no.: 0

input connection  :                output connection :
 14. a1,1,4 .std pin no.: 0         17. Y0,0,3 .side pin no
 15. cs1,0,0 .sl pin no.: 0
 16. cs2,0,0 .sl pin no.: 0

input connection  :                output connection :
 18. a1,1,4 .std pin no.: 0         21. Y0,0,3 .side pin no
 19. cs1,0,0 .sl pin no.: 0
 20. cs2,0,0 .sl pin no.: 0

input connection  :                output connection :
 22. A0,0,0 .std pin no.: 0         23. Y0,0,0 ram   pin no
                                    24. Y1,0,0 ram   pin no

input connection  :                output connection
 25. A0,0,0 .std pin no.: 0         26. Y1,0,0 rom   pin no

----- BLOCK DEFINITION -----
■ adder
■ alu
■ counter
■ interface bus
■ microprocessor
■ connection line
■ has already been defined
```

Figure 6.14 The Connection for the Micro-Processor

The above process continue until no more connection is required. Then the system starts searching for each of the connected pins and displays the corresponding form entry. Figure 6.14 shows the complete connection forms. Note that the user has specified output pin 'out1,1,8' thrice in order to connect them to three different components. Since no feedback path is required for the design, the data will be passed to the verifiers. When the verifier examines the 'micro-processor', it finds that its component 'control unit' needs further information from the user. This is the assignment of the signals, together with the group numbers, since the structure of the micro-instruction is a vertical one. The system will display a form for the user to fill in. These data will be verified at once to ensure no duplication of the signal occurs. Since the specified style of the 'control unit' is 'microprogrammed', a table of bit-patterns is displayed (Figure 6.15).

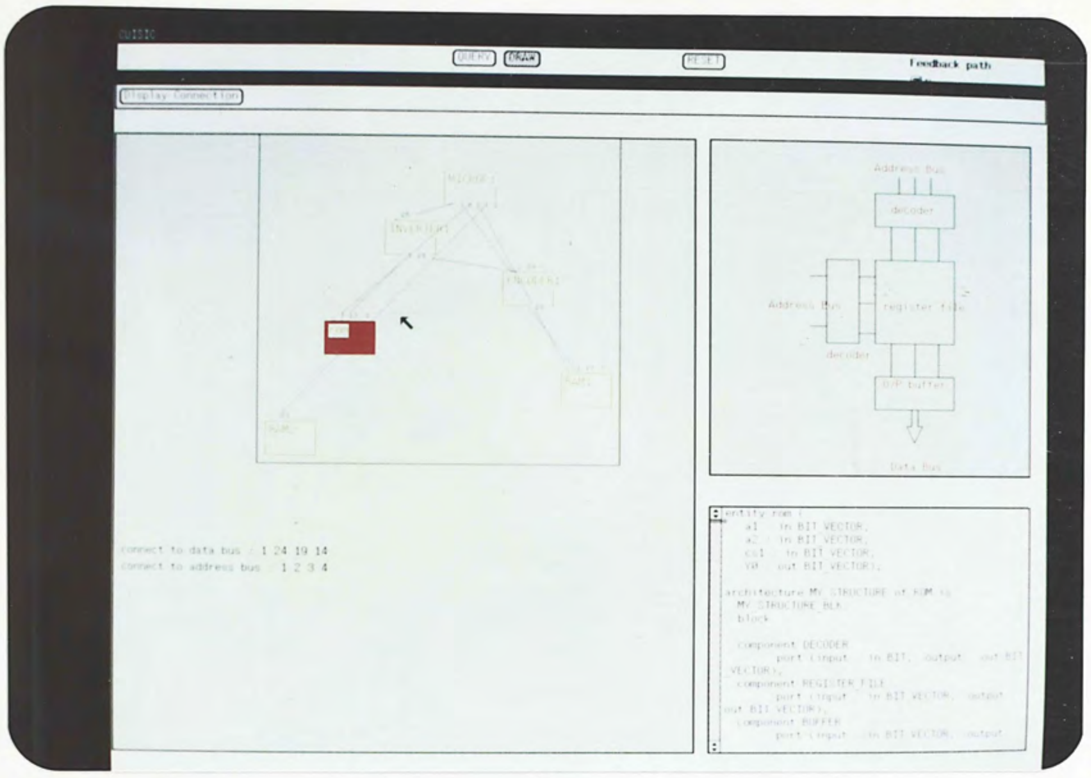


MICROPROGRAMS

Instructions	Mode	Address	r1in	r1out	yin	yout	zin	zout	mdrin	mdrout	add
add	direct-address		☐	✓	✓	☐	☐	☐	☐	☐	☐
			☐	☐	☐	☐	✓	☐	☐	✓	✓
			✓	☐	☐	☐	☐	✓	☐	☐	☐
			☐	☐	☐	☐	☐	☐	✓	☐	☐

<u>Figure 6.15 Table of Bit-Patterns for the Control Signals Assignment</u>

Since no further information is required by the system, the VHDL descriptions for the whole system as well as each component in the circuit under design will be generated. The final screen is shown in Figure 6.16. The information for each of the five components can be viewed in the subwindow by clicking on the specified item. The internal structure of each of the components can be seen on the upper left sub-window. This new design will be stored up in the module library with name 'sample2' and can be invoked later when the system is reset. Figure 6.17 shows the VHDL descriptions of this circuit.



```
connect to data bus : 1 24 19 14
connect to address bus : 1 2 3 4
```

```
entity rom (
    a1  : in BIT_VECTOR,
    a2  : in BIT_VECTOR,
    cs1 : in BIT_VECTOR,
    Y0  : out BIT_VECTOR);

architecture MY_STRUCTURE of ROM is
  MY_STRUCTURE_BLK
  block

  component DECODER
        port (input  : in BIT,  output : out BIT
_VECTOR),
  component REGISTER_FILE
        port (input  : in BIT_VECTOR,  output
out BIT_VECTOR),
  component BUFFER
        port (input  : in BIT_VECTOR,  output
```

Figure 6.16 The Final Display of the Block Diagram of the Design

```
component MICROPROCESSOR
    port (a1 : out BIT;
          a2 : out BIT;
          a3 : out BIT;
          r_w: out BIT;
          d1 : inout BIT);

component RAM
    port (cs1, cs2 : in BIT;
          r_w      : in BIT;
          a1       : in BIT_VECTOR;
```

```
        d1         : inout tristate);

component ROM
    port (cs : in BIT;
          a1 : in BIT_VECTOR;
          d1 : out tristate);

component DECODER
    port (a1 : in BIT;
          d1 : out BIT_VECTOR);

component INVERTER
    port (a1 : in BIT;
          d1 : out BIT);

signal address_bus : BIT_VECTOR (10 downto 1);
signal rw : BIT;
signal ram_cs : BIT_VECTOR(0 to 1);
signal rom_cs : BIT;
signal data_bus : tristate;

begin
    CPU : MICROPROCESSOR port (address_bus, rw, data_bus);
    SEL : DECODER port (address_bus(9), ram_cs);
    MO  : RAM port (ram_cs(0), address_bus(10), rw,
                   address(8 downto 1), data_bus);
    M1  : RAM port (ram_cs(1), address_bus(10), rw,
                    address_bus(8 downto 1), data_bus);
    INV : INVERTER port (address_bus(10), rom_cs);
    M2  : ROM port (rom_cs, address_bus(9 downto 1), data_bus);

end block;
```

Figure 6.17 The VHDL Description of the Design

7. CONCLUSIONS

This project has concentrated on the development of a VLSI design environment that assists the designer in the task of designing digital VLSI circuits. It is part of the CUISIC system which has been designed to satisfy its primary goal : creation of an ideal silicon compiler. Although the implementation of the whole system has not been finished, the structure of the system has been well developed. The idea behind the project is that a user who is not familiar with the techniques of digital and VLSI design should be able to create a special purpose chip. The implementation language is C, with Prolog as the underlying knowledge-representation language. Prolog is a language especially well-suited for AI development. No existing tool is adopted here for knowledge representation since it is too difficult to switch the data from one representation to another. The key part of the implemented system can be enhanced with ease, thus offering the possiblity of future extensions to the system.

The system structure is sufficiently general to allow a wide range of digital systems to be designed with relative ease. It allows the designer to enter into the system at different abstraction levels and styles. At the highest abstraction levels, the behaviour of the VLSI chip can be described using the form entry format.

The system has the ability to translate the high-level description given by the user (specifying the property lists of the design item by filling in its respective entry form) to the VHDL description. These procedural hardware descriptions are portable, thus giving the flexibility of the system.

The system adopts the knowledge-based heuristic approach for automating the design process. From the perspective of physical design, hierarchical design provides the framework that makes automation possible and enables design styles. Furthermore, this mimic expert designer's approach can enforce design discipline.

Until now, the major attention of the research has been concentrated on the design environment and the simple logic synthesis methods that translate a high-level description to a simple VHDL description. There are several limitations inherent in the system. The connection form cannot be checked interactively so that errors in filling in the pin number (an output pin instead of an input pin) can be caught at once. The system does not have the ability to do the optimization process for the circuit under designed. However, since all the necessary information has been stored in the data base, if time allows, it is not too difficult to overcome the problem by enhancing the knowlege-base of the system. The present system illustrates the new approaches to design entry and VHDL generation. The user interface is designed to minimize human error, the amount of information the user needs to remember, and it automatically invokes data conversion software when moving from one part of the design process to another. The idea of the VHDL generator is designed to automatically generate the VHDL description which, by far, has not been discussed in the literature. One point to be emphasized is that rules have been embedded so that errors in entering the specification has been kept to a minimum.

Besides the limitations described above, there are still several directiions that further research is needed :

a. As ASICS become very complex and are being designed by system designers it becomes imperative that all logical design be done automatically. These rely on the efficient logic synthesis algorithms that translate the behavioral description of the circuit under design into the one using functional description. This functional description is quite useful since it can be used to identify the existing circuit in the library that performs the same task as the new circuit under design. Then comparison can be carried out to determine which of them is the most efficient.

b. If errors can be eliminated before going to the physical design process, a lot of design time can be saved. Therefore the evaluation task is very important in determining the design quality, and in indicating whether the given constraints have been met. The verifiers discussed in Chapter 5 serve part of this purpose. It is understood that without the timing verifier the evaluation task would not be completed. Circuit performance is related to the worst-case propagation delay of signals between two register boundaries, because the system clock must be adjusted to allow the arrival of each signal at the destination registers within the clock cycle. This timing verification process should be done interatively in the following two steps :

(i) evaluate the critical path delay and then

(ii) modify the circuit if necessary

c. Future work should also be concentrated on the development of the VHDL generator. This generator transforms the behavioral description of the circuit under design into a program-like description langauge which can be simulated and analysed through the VHDL Support Environment.

d. In order to complete the CUISIC system, it is required to build a translator which translates the VHDL description to a suitable form for the input to the back end part of the system developed by the group of the undergraduate students.

REFERENCES

Bendas J.B. "Design Through Transformation", Proceedings of the 20th Design automation Conference Miami Beach, FL., 1983.

Cleemput W.M. *Computer Aided Design of Digital Systems - A Bibliography.* Woodland Hills, CS: Computer Sci. Press, 1976, pp. 231-236.

Cleemput, W.M. "An Hierarchical Language for the Structured Description of Digital Systems", Proc. of 14th DA. Conf., pp. 377-385, June, 1977.

Collett R. "silicon compilation: A Revolution in VLSI Design", Digital Design, August, 1984.

COMPUTER IEEE (special issue on new VLSI tools), vol. 16, no. 12, 1983.

COMPUTER IEEE (special issue on Design Automation), Vol. 19, No. 4, 1986.

Costanzo, S. Stefano A. and Faro A. "Towards an Expert System for Logic Circuit Synthesis", 1987.

DeMan H., Rabaey J., Siz P. "CATHDERAL II: A Synthesis and Module Generation System for Multiprocessor Systems on a Chip", NATO Study Institute of Logic Synthesis and Silicon Compilation for VLSI Design, L'Aqulia, Italy, 7-18 July 1986.

Den Yer et_al. "A Silicon Compiler for VLSI signal processor", Procedding ESSCIRC'82, 1982.

Duke K.A. and Maling K. "ALEX: A Conversational Hierarchical Logic Design System", Proc. of 17th DA. Conf., pp. 318-327, 1980.

Gray J.P. "INTRODUCTION TO SILICON COMPILATION", 16th Design Automation Conference, pp.219, June 1979

Jerraya A., Varniot P., Jamier R., Curtois B. "Principles of the SYCOcompiler", 23rd Design Automation Conference, June 1986.

Johannsen D. "Bristle Blocks - A Silicon Compiler", 16th Design Automation Conference, pp.214, 1979.

Johannsen D.L. et_al "Intelligent Compilation", VLSI SYSTEMS DESIGN, pp. 32-38, March, 1987.

Johnson S.C. "Silicon Compiler lets system makers design their own VLSI chips", Electronic Design, pp. 167-181, October 4, 1987.

Kawato N., Saito T. and Uehara T. "A CAD System for Logic Design Based on Frames and Demons", Proc. 18th DA Conf., pp. 451-456, 1981.

Knapp D.W., Parker A.C. "A Design Utility Manager: the ADAM Planning Engine", 23rd Design Automation Conference, 1986.

Koford J.S., et_al "Using a graphic data processing system to design artwork for manufacturing hybrid integrated circuits", in *Proc. Fall Computer Conf.*, pp.229-246, 1966.

Kowalski T.J., et_al "Automatic Data Path Synthesis", IEEE COMPUTER, pp. 59-69, December, 1983.

Kowalski T.J. "The VLSI Design Automation Assistant: What's in a Knowledge Base", 22nd Design Automation conference, pp. 252-258, 1983.

Krekelberg et_al "YET ANOTHER SILICON COMPILER", 22nc Design automation conference, pp. 176-182, 1985.

Lee B. and Ritzman D. "Silicon compiler teams with VLSI workstation to customize CMOS ICs", Electronic Design, pp. 149-162, November 15, 1984.

McWilliams T.M. and Widdoes L.C. "SCALD: Structured Computer-Aided Logic Design", Proc. of 15th DA. Conf., pp.271-277, June, 1978.

Ng K.W. "DISCDA - A Digital System Controller Designer Assistant", pp. 57-62, Microprocessing and Microprogramming 23, North Holland, 1988.

Noyce R.N.(Moore's Law), "Microelectronics", *Sci. Amer.*, vol. 237, no. 3, pp. 65, September, 1977.

Peskin A.M. "Toward a Silicon Compiler", Proceedings of the 1982 Custom Integrated Circuits Conference, Rochester, NY.

Proc. of the IEEE (special issue on VLSI Design: Problems and Tools), Vol. 71, No. 1, 1983.

Rawson J. and Trimberger S. "Second-generation compilers optimize semicustom circuits", Electron Design, pp. 92-96, 1987.

Saito T., Uehara T. and Kawato N. "A CAD System for Logic Design Based on Frames and Demons", 18th Design Automation Conference, 1981.

Schindler M. "Silicon compilers travel rough road to acceptance", Electronic Design, pp. 156-166, May 1 1986.

Six P., L. Claesen, J. Rabaey, and H. De Man "An Intelligent Module Generator Environment", 23rd Design Automation Conference, pp. 730-735, 1986.

Southard J.R. "MacPitts: An Approach to Silicon Compilation", IEEE Computer, vol. 16, no. 12, Dec 1983.

Subrahmanyan P.A. "Synapse: An Expert System for VLSI Design", COMPUTER, PP. 78-89, July 1986.

Uehara T et_al "An Interactive Logic Synthesis System Based Upon AI Techniques", 1982.

VLSI DESIGN Staff "Silion Compilers Part 1: Drawing a Blank", VLSI DESIGN, pp. 54-58, September 1984.

Werner J. "Progress Toward the "Ideal" Silicon Compiler", VLSI DESIGN, pp.38-41, September, 1983.

Winston P.H. , Artificial Intelligence. Addison-Wesley, 1977.

Wong Y.C, Chan H.K. and Lam C.H. "CAD Tools for VLSI Design", internal paper of the Chinese University of Hong Kong, 1988.

PUBLICATION

Ng W.M.T. "Synthesizing VLSI Circuits from Behavioral Specifications : an Overview of CUISIC", Proc. CADEMAT '88.

APPENDIX SUMMARY OF SUB-SYSTEMS

The following is a summary of each of the sub-systems. Since there is no interfaces between NU-Prolog and C, data obtained from C are stored in files. Rules that govern the user specifications are written in C.

1. Strategy Rule-Base

There are about 16 set of rules which govern the design of the items. The number of rules for each set depends on the design item, each of them is around 50 lines. The global data retrieved from the user specification is stored in a file 'datstart'. These data include :

name of the design item,
name of the file to be used for storing the new I/O pins for the circuit under designed,
name of the data file for storing information concerning the new design
name of the new task frame that the rules for the new design are stored
and whether the design contains any feedback path

These names have to be determined before passing to Prolog since Prolog cannot handle string manipulation.

The name of the data file for each of the modules is the same as its node number which appears in a tree structure (denoted in a depth-first manner). When the prolog program is invoked, the file 'datstart' is read by the predicate 'get_select(Noarg)'. This contains rules that :

determines which set of rules that is going to be invoked,
invokes the connection verifier and
finally saves the parameters.

For example, the design item is an 'adder' with cell name = add_4, and no module is connected to it. Then there will be two files passed to prolog : 'datstart' and '1'. The contents of both of the files are shown below :

```
design(item, cuisic_p).
subdesign(item, adder).
portfile(oiadd_4).
collect_dat(datadd_4).
rulename(../nuprog/ruleadd_4.nl).
feedback(no).
```

<u>Figure A-1 Data for datstart</u>

```
iport(adder,[[1,in1,0,1],[2,in2,0,1],[3,cin,0,0]]).
oport(adder,[[4,sum,0,3],[5,cout,0,0]]).
data(data_bus, Nbits).
choice(adder, cla).
cell_name(add_4).
```

Figure A-2 Data for '1'

Then a set of rules governing the design of an adder will be invoked :

```
selection([cuisic_p,Count],C,Tab) :-
        subdesign(item,S),
        selection([S,0],C,Tab).

selection([adder,Count],C,Tab) :-
        plus(Count,1,Count2),
        retract(count(adder,_)),
        assert(count(adder,Count2)),
        asserta(cell_no(adder,Count2)),
        consult('/uac/gds/ng047/thesis/nuprog/ruleadd.nl').
        adder(C,Tab),
        complist(adder,C,Tab).
```

For each of the design item, there is a counter counting the occurence of the item in the circuit under design. Data concerning a particular item can be referenced easily. Each set of rules is stored in a file with name starting with 'rule'. When the data concerning an 'adder' is examined, its component list will be searched. For each of the component, if any, the above procedure is repeated.

the rules for the adder determine :

the choice for the adder, whether it is CLA or full adder design

the validation of the input and output connections, if any

the new input port and output port lists for the new item

the component list for the adder

the parameters for the VHDL description

These data will be stored in the file 'datadd_4'.

2. Enquiry Rule-Base

This rule-base gives advice to the user if required. As stated above, it has not been finished yet. The rule-base only contains about 15 rules for illustrative purpose. The items under design that have been considered are :

micro-computer
adder
alu

micro-processor

addition

multiplication

These rules are written in C and they are embedded in the starting program.

3. **Information Data-Base**

All the data are stored as facts in the working area. Each fact includes a basic parameter 'counter' so that it can be easily referred to. The size of the working area for NU-PROLOG determines the size of the data base.

4. **Verifiers**

These verifiers check the validation of :

the data specified by the user,

the connection for the item under designed and

the feedback path if any

They contain about 45 rules and the size is around 200 lines. The connection of the circruit is stored in a fact 'connect_to' as illustrated below:

```
connect_to(Source,Cs,Dest,Cd,Ps,Pd,Start1,End1,Start2,End2).
```

where Source (Destination) = source (destination) item,

Cs (Cd) = counter for the source (destination)

Ps (Pd) = port number for the items,

Start1 (Start2) = starting line number for source (destination),

End1 (End2) = ending line number for source (destination),

These data will be used later to draw the block diagram for the circuit under design.

5. VHDL generator

The hardware frame is specified with name starting with 'v'. In this case, the VHDL frame that is going to be used is 'vadd'. The content is shown below :

```
architecture REGULAR-STRUCTURE of FOUR_BIT_ADDER is
    REGULAR_STRUCTURE_BLK :
    block
         signal C : BIT_VECTOR;
    component FULL_ADDER
         port (ADDEND, AUGEND, CARRY : in BIT;
               CARRY_OUT, SUM : out BIT);

    begin
         L0 : FULL_ADDER port(A(0), B(0), CIN, C(0), SUM(0));
         L1 : FULL_ADDER port(A(1), B(1), C(0), C(1), SUM(1));
         L2 : FULL_ADDER port(A(2), B(2), C(1), C(2), SUM(2));
         L3 : FULL_ADDER port(A(3), B(3), C(2), C(3), SUM(3));
    end block;
end REGULAR_STRUCTURE;
```

This is the fixed part stored in the frame. The entity of this adder is determined by the parameters passing in from the data-base. An example below shows a typical entity using the parameters in the input and port lists:

```
entity FOUR_BIT_ADDER
    (in1, in2: in BIT_VECTOR;
     Cin: out BIT_VECTOR;
     sum : out BIT_VECTOR;
     cout : out BIT) is
end FOUR_BIT_ADDER;
```

This new frame will be stored under the name 'vadd_4'.