UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

VHDL : HARDWARE DESCRIPTION LANGUAGE FOR DIGITAL SYSTEM DESIGN USING SYNTHESIS TOOLS

PHILIP A. MILLS

Las Palmas de Gran Canaria, Septiembre de 1995

Título del Proyecto Fin de Carrera:

VHDL : HARDWARE DESCRIPTION LANGUAGE FOR DIGITAL SYSTEM DESIGN USING SYNTHESIS TOOLS

VHDL : Hardware Description Language for digital system design using synthesis tools

A Dissertation

submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in Computer Science and Applications

Faculty of Science

The Queen's University of Belfast

in conjunction with

Escuela Tècnica Superior de Ingenieros de Telecomunicaciòn

Universidad de Las Palmas de Gran Canaria

by

Philip A. Mills M.Eng. A.M.I.Mech.E.

September 1995

Tutors : Pedro P. Carballo & Tomàs Bautista

Acknowledgements

I would like to thank my supervisors, Pedro P. Carballo and Tomàs Bautista for all their help and encouragement while working on this project. I would also like to thank Alfonso Naranjo for his help in completing the VHDL design, simulation and synthesis. Thanks to Mike McKeag and Valentìn de Armas Sosa for co-ordinating this ERASMUS exchange, and also to all the staff of ULPGC whom I have met in the past four months, and who have helped in the completion of this project in many ways.

Contents

List of Figures

List of Tables

Abstract

VHDL is a Hardware Description Language used to desribe the behaviour and structure of electronic digital systems. From a VHDL description it is possible to simulate the behaviour of the system under design and also synthesize the structure (at gate level) of the final system. The objectives of this project are to learn VHDL, and get some familiarity with the Leapfrog/Synergy tools from Cadence. Also to run a complex design example, and to write a tutorial.

Chapter 1

INTRODUCTION

1.1 General Introduction

In this introduction there is to be found a discussion of the basic background concepts used throughout the project. First of all, the issue of design automation will be discussed, along with the advantages to be gained from automating the process. There follows a definition and discussion of the synthesis stage of the design process. Hardware Description Languages, which are one way of automating the design process, are discussed in the following section in very general terms. After this is an introduction to cryptography, which is the subject concerning the practical part of this project.

1.2 The need for design automation

In the early 1990's, Very Large Scale Integration (VLSI) technology reached-the level of over one million transistors per chip, making such items very difficult to design by handcrafting or by logic gate definition. However, VLSI was also very well understood, and the electronics industry therefore deemed it necessary to examine the entire design process from coneptualisation to manufacture, with a view to automating many stages of the process.

There were two basic concepts which helped to shorten the time-to-market cycle :

- The *first silicon* approach attempts to reduce the number of iterations through the fabrication process to just one, i.e. the first piece of silicon produced will be a correct one. This approach requires automated tools for verification of both the functionality and the design rules for the entire chip.
- The *first specification* approach has the goal of reducing the number of iterations over the product specification to only one. This approach requires accurate modelling of the design process, and also accurate estimation of the product quality measures including performance and cost. Moreover, these will be required early in the specification phase.

There are three main advantages offered by automating the design process, and moving the automation to higher levels :

- It assures a much shorter design cycle. This leads to cheaper and more economic chip manufacture.
- It allows quick exploration of different design styles. This enables different designs to be investigated and simulated, without the need for expensive prototype manufacture. This again leads to more economic chip manufacture.

- Design automation tools *may* out-perform human designers in generating high quality designs. This is dependant upon the simulation and synthesis algorithms being accurate and well-understood.

1.3 Definition of Synthesis

When considering a digital electronic design, there are three domains which may be referred to :

- The **Behavioral Domain** is concerned with what a design actually does, not how it is built. The design is treated as one or more black boxes, with a specified set of inputs and outputs, and a set of functions describing the output in terms of the input. Also included is a description of the external interface and constraints.
- The **Structural Domain** represents a set of components and connections under constraints such as cost, area and delay.
- The **Physical Domain** ignores as far as possible the function and behavior of the design, but binds its structure in space, or on silicon.

Synthesis can be defined as the translation process from the behavioral description of the design, to a structural description. This may be carried out at a number of different levels :

- **Circuit Synthesis** generates a transistor schematic from a set of I/O current, voltage and frequency characteristics or equations.
- **Logic Synthesis** translates Boolean expressions into a net-list of components from a given library of logic gates.
- **Register Transfer Synthesis** starts with a set of states and a set of register transfers in each state. It generates two parts :

  A data path, which is a structure of storage elements and functional units which perform the given register transfers.

  A control unit which controls the sequencing of states in the register-transfer description.

- **System Synthesis** starts with a set of processes communicating through either shared variables or message passing. It generates a structure of processors, memories, controllers and interface adapters, from a set of system components.

All of these types of synthesis generate structures which are not bound in space, i.e. they are logical structures rather than physical structures. At each level, further synthesis phases are required to add the physical information about the circuit.

1.4 Hardware Description Languages

A Hardware Description Language (HDL) can be used to describe the behavior or structure of systems either at board level, or at chip level. The HDL provides a way to describe a circuit by using familiar programming language constructs, such as conditional statements (IF..THEN..ELSE) and loop statements (WHILE..DO).

There are, however, a number of problems associated with the creation of such a language. The main problem is the changes which take place to a design description as it proceeds through the design process. At first the description is vague, with little or no implementation detail. As the design evolves, more detail is added, and the level of abstraction is lowered, until the design is ready to be manufactured. A language which spanned all the phases of the design process would be highly desireable, but would be much too cumbersome to be mastered by designers.

Also, the different members of a design team require different views of the design, for the tasks of verification, testing, reliability testing, manufacturing and maintenance. If all the information required by all these people were added to the language, the result would be a language which was very rich and powerful, but also in which information for one particular aspect of the design would be very difficult to find. Thus the development of HDLs requires some compromise to be made so that the language is sufficiently powerful, but also concise.

1.5 Introduction to Cryptography

Cryptography, or data encryption, is the transformation of data into a form unreadable by anyone without a secret decryption key. Its purpose is to ensure privacy by keeping the information hidden from anyone for whom it is not intended, even those who can see the encrypted data. For example, one may wish to encrypt files on a hard disk to prevent an intruder from reading them, or to encrypt files for transmission over telephone lines. As the Internet and the 'Information Superhighway' grow in stature, such data protection methods are becoming more and more important. In a multi-user setting, encryption allows secure communication over an insecure channel. There are two basic types of data encryption, *secret key cryptography*, and *public key cryptography*.

1.5.1 Secret Key Cryptography

Traditional cryptography is based on the sender and receiver of a message knowing and using the same secret key. The sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. This method is known as *secret-key cryptography* or *symmetric cryptography*.

The general scenario can be described as follows. Suppose Alice wishes to send a message to Bob so that no one else besides Bob can read it. Alice encrypts the message, which is called the *plaintext*, with an *encryption key*. The encrypted message, called the *ciphertext*, is sent to Bob. Bob decrypts the ciphertext with the *decryption key* and reads the message. An attacker, Charlie, may either try to obtain the secret key or to recover the plaintext without using the secret key. In a secure cryptosystem, the plaintext cannot be recovered from the ciphertext except by using the decryption key.

The main problem with symmetric cryptography is getting the sender and receiver to agree on the secret key without anyone else finding out. If they are in separate physical locations, they must trust a courier, or a phone system, or some other transmission system to not disclose the secret key being communicated. If this transmission system is secure, then there is surely no need for data encryption in the first place. Anyone who overhears or intercepts the key in transit can later read all messages encrypted using that key. The generation, transmission and storage of keys is called *key management*. All cryptosystems must deal with key management issues. Secret-key cryptography often has difficulty providing secure key management.

1.5.2 Public Key Cryptography

Public-key cryptography, or *non-symmetric cryptography* was invented in 1976 by Whitfield Diffie and Martin Hellman in order to solve the key management problem. In the new system, each person gets a pair of keys, called the *public key* and the *private key*. Each person's public key is published while the private key is kept secret. The need for sender and receiver to share secret information is eliminated. All communications involve only public keys, and no private key is ever transmitted or shared. No longer is it necessary to trust some communications channel to be secure against eavesdropping or betrayal. Anyone can send a confidential message just using public information, but it can only be decrypted with a private key that is in the sole possession of the intended recipient. Furthermore, public-key cryptography can be used for authentication (digital signatures) as well as for privacy (encryption).

For encryption, the procedure is as follows. When Alice wishes to send a message to Bob, she looks up Bob's public key in a directory, uses it to encrypt the message and sends it off. Bob then uses his private key to decrypt the message and read it. No one listening in can decrypt the message. Anyone can send an encrypted message to Bob but only Bob can read it. Clearly, one requirement is that no one can figure out the private key from the corresponding public key.

For authentication, the procedure is as follows. Alice, to sign a message, does a computation involving both her private key and the message itself. The output is called the *digital signature* and is attached to the message, which is then sent. Bob, to verify the signature, does some computation involving the message, the purported signature,

and Alice's public key. If the results properly hold in a simple mathematical relation, the signature is verified as genuine. Otherwise, the signature may be fraudulent or the message altered, and they are discarded.

1.5.3 Advantages and Disadvantages

The primary advantage of public-key cryptography is increased security: the private keys do not ever need to transmitted or revealed to anyone. In a secret-key system, by contrast, there is always a chance that an enemy could discover the secret key while it is being transmitted.

Another major advantage of public-key systems is that they can provide a method for digital signatures. Authentication via secret-key systems requires the sharing of some secret and sometimes requires trust of a third party as well. A sender can then repudiate a previously signed message by claiming that the shared secret was somehow compromised by one of the parties sharing the secret. For example, the Kerberos secret-key authentication system involves a central database that keeps copies of the secret keys of all users; a Kerberos-authenticated message would most likely not be held legally binding, since an attack on the database would allow widespread forgery. Public-key authentication, on the other hand, prevents this type of repudiation; each user has sole responsibility for protecting his or her private key. This property of public-key authentication is often called non-repudiation.

Furthermore, digitally signed messages can be proved authentic to a third party, such as a judge, thus allowing such messages to be legally binding. Secret-key authentication systems such as Kerberos were designed to authenticate access to network resources, rather than to authenticate documents, a task which is better achieved via digital signatures.

A disadvantage of using public-key cryptography for encryption is speed. There are popular secret-key encryption methods which are significantly faster than any currently available public-key encryption method. But public-key cryptography can share the burden with secret-key cryptography to get the best of both worlds.

For encryption, the best solution is to combine public- and secret-key systems in order to get both the security advantages of public-key systems and the speed advantages of secret-key systems. The public-key system can be used to encrypt a secret key which is then used to encrypt the bulk of a file or message. Public-key cryptography is not meant to replace secret-key cryptography, but rather to supplement it, to make it more secure. The first use of public-key techniques was for secure key exchange in an otherwise secret-key system; this is still one of its primary functions.

1.6 Project Aims and Objectives

The objectives of this project are several. Firstly to learn the concepts and command structure of VHDL, and to gain some experience of its use in straightforward examples. Then to gain some familiarity with the design automation tools from Cadence, especially the Leapfrog Simulation tool, and the Synergy Synthesis tool, and to learn how they may be used as part of the overall design process. Then to write a tutorial describing the integrated use of the tools, and finally to utilise the tools to process a more complicated design.

The VHDL language structure and features are introduced in Chapter 2, along with a very simple example of its use. The design process is introduced in Chapter 3 together with a description of where each of the automated tools may be used. Chapter 4 describes the Data Encryption Algorithm which is an international standard algorithm for data encryption and draws particular attention to the Key Generation Algorithm. Chapter 5 describes a VHDL model of the Key Generation Algorithm, while Chapter 6 illustrates how this part of the algorithm fits in to the overall design. Chapters 5 and 6 also contain simulation and synthesis results for the Key Generation Algorithm, and for the overall design. Conclusions and suggestions for further work follow in Chapter 7.

Chapter 2

VHDL

2.1 Introduction

Very high speed integrated circuits Hardware Description Language (VHDL) is a hardware description language used to describe digital electronic systems. It arose out of the United States Government's Very High Speed Integrated Circuits (VHSIC) program, which was initiated in 1980. It became obvious during this program that a standard language was required for the description of integrated circuits, and so the VHSIC Hardware Description Language (VHDL) was developed. It was subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE).

VHDL fulfills a number of requirements within the design process :

- It allows the structure of a system to be described, along with its decomposition into sub-designs, and how the sub-designs are interconnected.
- It allows the specification of the function of a design using familiar programming language sructures.
- It allows designs to be simulated prior to manufacture, which reduces the need for expensive hardware prototypes.

2.2 Model Organisation

A VHDL model of a digital system usually takes the form of a hierarchical collection of modules. An *entity* is such a module, which may be used as a component in a design, or which may itself be the top level. Having been specified in an *entity declaration*, one or more implementations of the entity can be described an *architecture declaration*. Each architecture body can describe a different view of the entity, normally either a *structural view*, or a *behavioral view*. The structural view describes how the entity is constructed using other lower-level entities, while the behavioral view simply describes how the module responds to inputs.

As a simple example of how the VHDL language is used, consider the block diagram shown in Figure 2.1.

The layout consists of three gates, an AND-gate, an OR-gate and a NAND-gate. Using VHDL, the system may be modelled using a single entity declaration, and one of two possible architecture bodies.

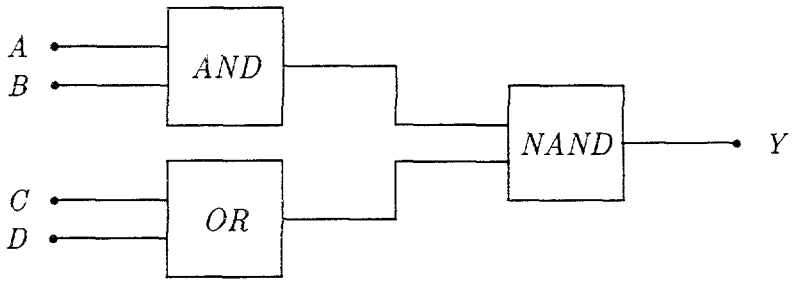


Figure 2.1: Schematic diagram of module EXAMPLE

2.2.1 Entity Declaration

The most basic unit used in VHDL is the *entity declaration.* This simply describes the inputs and outputs, or *ports* of the module being described, along with the type of data which will be carried by the ports.

The entity declaration for the module is shown in Figure 2.2. It declares that the module has four inputs, A, B, C and D, each of type *bit*, and one output, Y, also of type *bit.*

```
entity EXAMPLE is
  port ( A, B, C, D : IN bit;
         Y : OUT bit );
end EXAMPLE;
```

Figure 2.2: Entity declaration for module EXAMPLE

2.2.2 Describing Structure

One way of describing the function of an electronic circuit, or module, is to describe its structure, i.e. how it is constructed using smaller modules. Each of the sub-modules is an *instance* of some entity, and the entities are connected together using *signals.* This type of description is called a *structural* description and is shown in Figure 2.3.

2.2.3 Describing Behaviour

In many cases, it is not appropriate to describe a module by its structure. One example of this is where a module is at the bottom of a hierarchy of some other description, where the module may be a proprietary item, and hence the internal structure does not

```
architecture STRUCTURE of EXAMPLE is

  component AND_GATE
    port ( IN1, IN2 : IN bit;
           OUT1 : OUT bit );
  end component;

  component OR_GATE
    port ( IN1, IN2 : IN bit;
           OUT1 : OUT bit );
  end component;

  component NAND_GATE
    port ( IN1, IN2 : IN bit;
           OUT1 : OUT bit );
  end component;

  signal s1, s2 : bit;

begin
  A1 : AND_GATE port map ( IN1 => A, IN2 => B, OUT1 => s1 );
  O1 : OR_GATE  port map ( IN1 => C, IN2 => D, OUT1 => s2 );
  N1 : NAND_GATE port map ( IN1 => s1, IN2 => s2, OUT1 => Y );
end STRUCTURE;
```

Figure 2.3: Structural description of EXAMPLE

need to be described. In fact, the internal structure may not even be known. In such cases, it may be possible to describe the module using its *functional* or *behavioural* description. A behavioral architecture for the example module is shown in Figure 2.4.

2.2.4 Simulation

Once the system has been described in terms of structure and/or behaviour, it is possible to simulate the action of the module by running its behavioural description. This is done by simulating the passage of time in discrete steps. If the value on the input port of some module changes, then the module runs its behavioral description.

The simulation starts with an *initialisation phase*, and then proceeds by running a two-phase *simulation cycle*, which are as follows:

- Intialisation Phase - all signals are assigned initial values, the simulation time is set to zero, and each modules behavioural description is executed. This usually results in some transactions being scheduled on output signals at some future time.

```
architecture BEHAVIOUR of EXAMPLE is

begin
  RUN : process
  begin
    Y <= (A AND B) NAND (C OR D) after 12 ns;
  end process;
end BEHAVIOUR;
```

Figure 2.4: Behavioural description of module EXAMPLE

- Simulation Cycle Phase 1 - The simulation time advances to that of the earliest transaction. All transactions scheduled for that time are run.
- Simulation Cycle Phase 2 - All reacting modules have their behavioural programs executed. The simulation cycle then repeats. If no further transactions remain, then the simulation is complete.

The purpose of simulation is to gather information about the changes which take place in the system over time. A *simulation monitor* will allow signals and other state information to be viewed during the simulation, and often allows for interactive stepping of the simulation process.

2.3 Language Features of VHDL

One of the primary advantages of VHDL is that most of its command language is very similar to many comman software languages. In particular, VHDL bears a strong resemblance to the Ada programming language, although the facilities are not quite so comprehensive.

2.3.1 Lexical Elements

VHDL incorporates all the normal lexical elements used by programming languages, including the following :

- **Comments** - represented in VHDL by two adjacent hyphens ('−−').
- **Identifiers** - in VHDL, identifiers must begin with a letter, and must contain only letters, underscores, and digits. Identifiers are not case-sensitive.
- **Numbers** - may be expressed in decimal, or in any base between 2 and 16.

- **Characters** - single literal characters are formed by placing an ASCII character within a pair of single quote marks, e.g. 'A'.

- **Strings** - literal strings of characters are formed by enclosing the characters in a pair of double-quote marks, e.g. "A string".

- **Bit Strings** - VHDL also allows strings to be made up of nbinary characters, octal characters or hexadecimal characters, e.g. O"126" is equivalent to binary 001002110.

2.3.2 Data Types and Objects

VHDL provides a number of basic, or *scalar* data types, and also a means of forming *composite* types. Scalar types include numbers and physical quantities, while composite types include arrays and records.

- **Integer Types** - a range of integer values within a specified range. For example,

```
type byte_int is range 0 to 255;
type bit_index is range 31 downto 0;
```

- **Physical Types** - used for representing some physical quantity, such as mass, length, time or voltage. Particularly important in VHDL is the predefined type *time*, as it is used extensively in simulations.

- **Floating Point Types** - this is a discrete approximation to the set of real numbers within a specified range. The precision is implementation defined, but must be at least six decimal digits.

- **Enumeration Types** - an ordered set of identifiers or characters. Within a single enumeration type, the identifiers and characters must be distinct.

- **Arrays** - an indexed collection of elements, all of the same data type. The arrays may be one-dimensional or multi-dimensional. The array type may be *constrained*, where the bounds for the index are established upon definition, or *unconstrained*, where the bounds may defined later.

- **Records** - a collection of named elements, possibly of different types.

2.3.3 Expressions and Operators

Expressions in VHDL are very similar to expressions in other programming languages. An expression is a formula which combines a number of identifiers, using a number of operators, which may be of the following type :

- **Logical operators** include *and, or, nand, nor, xor* and *not*, and may be used on values of type bit.
- **Relational operators** include =, /=, <, <=, > and >=. They must have both operands of the same type, and yield a Boolean result.
- **Sign operators**, + and -, have the usual meaning of addition and subtraction.
- **Concatenation operator**, &, operates on one-dimensional arrays, to produce a new array, consisting of the left operand followed by the right.
- **Mathematical operators** include multiplication (*), division (/), modulus (**mod**), remainder (**rem**), absolute (**abs**), and exponentiation (**).

2.3.4 Sequential Statements

VHDL contains a number of facilities for modifying the state of objects, and for controlling the flow of execution of models. Many of these are very similar to those found in common programming languages.

- **Variable Assignment** - as in other programming languages, variables may be given new values using variable assignment statements. The object and value must always be of the same base type.
- **If Statement** - allows selection of statements to be executed depending on the result of a condition. For example,

```
if x_in = 1
    even <= '0';
    small <= '1';
elsif x = 2 then
    even <= '1';
    small <= '1';
else
    even <= '0';
    small <= '0';
end if;
```

- **Case Statement** - allows selection of statements to be executed depending on the result of a selection expression. For example,

```
case control is
  when 1 =>
      y_out <= x_in1;
  when 2 =>
      y_out <= x_in2;
  when others =>
      y_out <= '0';
end case;
```

- **Loop Statements** - VHDL allows for the creation of the usual while and for loops found in other programming languages. There are two additional commands which may be used inside the loop body. The **next** statement, causes execution to move immediately to the next iteration of the loop, without executing any following code. The **exit** statement causes the loop to terminate, and jumps to the first statement after the end of the loop.

- **Null Statement** - this has no effect whatsoever. It is most often used to show that in a particular case, no action is required.

- **Assertions** - are used to verify a particular condition, and to report if the condition is violated.

2.3.5 Subprograms and Packages

In common with other programming languages, VHDL provides subprogram facilities in the form of procedures and functions. VHDL also provides a package facility, whereby declarations and objects may be gathered together into modular units. Packages also provide a measure of data abstraction and information hiding.

- **Procedures and Functions** - these are declared by giving the procedure or function name, together with the parameters require and, in the case of a function, the return type of the result.

- **Overloading** - two subprograms are allowed to have the same name, as long as the number or type of the parameters are different. The subprogram is then said to be *overloaded.*

- **Packages** - a package is a collection of types, constants, subprograms and possibly other things, usually intended to isolate a group of related items. In particular, details of constants and subprogram bodies may be hidden from the user within a package. The package itself may be split into two parts, the *package declaration* and the *package body*. The declaration defines things which are visible to the user and to the package body, like the procedure and function headers, and constant names and types. Procedure and function bodies, along with constant values, may then be provided in the package body.

2.4 Describing Structure using VHDL

When a designer wishes to describe a system in terms of the way in which it is made up from other components, he uses a structural architecture. The structural architecture is made up of a number of components, connected together using signals.

- **Signals** - these are used within a structural architecture to connect the various components. Signals must be declared before they are used, and may be given a default value.

- **Component Declarations** - a stuctural architecture normally makes use of other entities which are described separately and placed in design libraries. In order to do this, an architecture must declare a component, which can be thought of as a template, and which is then instantiated within the architecture.

- **Component Instantiation** - A component declared within an architecture may then be instantiated as many times as required, with each instance being given a unique name. The ports of the instance will then be mapped on to internalk signals, or the ports of the top-level entity.

2.5 Describing Behaviour using VHDL

The primary unit of behavioral description in VHDL is the **process**. A process is a sequential body of code which is activated in response to a change in state. When more than one process is activated at the same time, they execute concurrently. A process usually contains a number of signal assignment statements.

- **Wait Statement** - this may be used to suspend the execution of a process, either unconditionally, for a specified period of time, or pending a certain condition.

- **Sensitivity List** - this is a list of signals provided in the process header. The process will activate when any one of these signals changes value, and so is said to be *sensitive* to these signals.

- **Conditional Signal Assignment** - this is a shorthand way of describing a process containg signal assignments in an if statement. For example the statement

```
if x < 20 then s <= '01';
elsif x > 19 and x < 30 then s <= '10';
elsif ...
else s <= '00';
end if;
```

could be replaced by the conditional signal assignment

```
s <= '01' when x < 20 else
     '10' when x > 19 and x < 30 else...
     '00';
```

- **Selected signal Assignment** - this is a shorthand way of describing a process containing signal assignments in a case-statement. For example, the statement

```
case base is
   when 1 =>
          s <= '001';
   when 8 =>
          s <= '010;
   ...
   when 16 =>
          s <= '100';
end case;
```

may be replaced by the selected signal statement

```
with base select
   s <= '001' when 1,
        '010' when 8,
        ...
        '100' when 16;
```

Chapter 3

THE DESIGN FRAMEWORK

3.1 Overview

The electronic design process consists of a number of steps, which can be summed up in the flowchart in Figure 3.1.

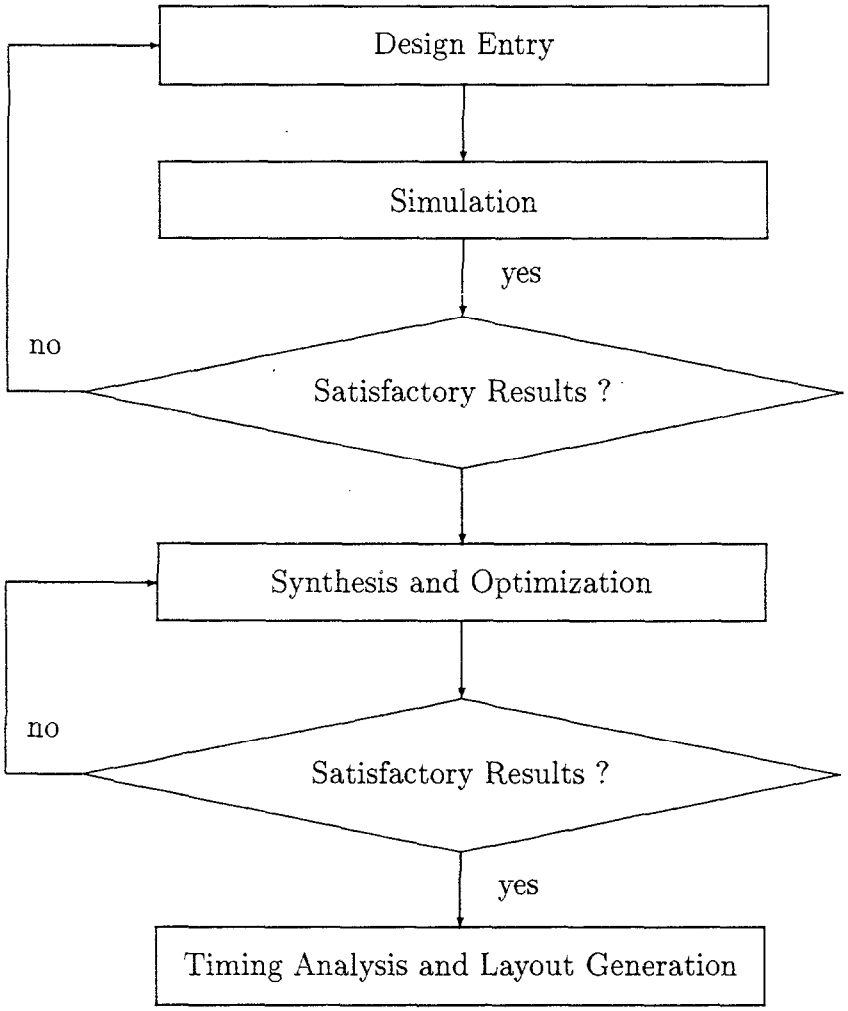


Figure 3.1: Typical Top-down Design Process

The first stage in the design process is the entry of the design into the system being used. Depending on the nature of the system, the design input may come in a number of different forms, including VHDL code, schematic diagram, state digrams, truth tables, etc. When the entire design has been input, the design hierarchy can be checked to ensure that it is error-free.

Having verified the integrity of the design, the next stage is to run a simulation. This process provides the model with some form of input, and the simulation tool should allow the designer to observe the reaction of the model to the input data, and also to note how the state of the model changes with respect to time. Very often the changes to inputs and outputs over time will be viewed as waveforms.

If the simulation results are not as expected, or the design does not behave as it should, the designer must return to the data entry stage, in order to make any changes which are required. This iterative process is followed until the simulation produces the required results.

Having established that the design behaves correctly, the designer may now turn his attention to the synthesis process. This takes the design, and produces a logical layout description of the design, often in the form of a netlist or schematic diagram, describing all the components or gates which make up the system, and how they are interconnected. The designer usually has the opportunity to decide what type of technology is to be used and from what type of components the system will be made up. There is also the opportunity to place constraints on the layout of the system, particularly in terms of area, cost and timing.

If the synthesis results are not as required, then it is necessary to run the synthesis step again, possibly with different constraints, or perhaps using a different technology for the component library. In extreme cases, it may be necessary to go all the way back to the design entry stage, to make small changes to the model, in order to aid the synthesis process. In such cases, the simulation step should also be run again to ensure the continuing integrity of the design.

Once the results of the synthesis run are satisfactory, the final stage is to check the timing properties of the design, and to produce a physical layout. The circuit is then ready for manufacture.

3.2 The VHDL Tool Box

The VHDL Tool Box of the Cadence System provides an integrated environment within which VHDL designs may be created, edited, simulated and synthesized. The Tool Box manages all of the following interfaces :

- Schematic Editor for composing a graphical representation of the design structure.
- Language Sensitive Editor (LSE) for creating and editing the VHDL source code.
- VHDL netlister.
- Simulation tools for carrying out simulation and testing of the design.

- Synthesis tools for producing a circuit layout from the design.
- Automatic Test Bench creation for the simulation run.
- VHDL design import for reading in design files which have been created previously.
- VHDL translation for mapping a design from one type of technology to another.

The VHDL Tool Box supports all phases in the design of digital components, regardless of the design process being used, or the implementation technology.

3.3 The Leapfrog Simulation Tool

The Leapfrog VHDL Simulator is a suite of programs that provide VHDL analysis, elaboration and simulation. Within the Leapfrog system, the following tasks may be carried out :

- Analyze VHDL source files created either with an editor, or by software that generates VHDL, and store the results of the analysis in VHDL design libraries.
- Examine and manipulate the contents of VHDL design libraries.
- Elaborate a design prior to simulation. This step links previously analyzed VHDL models as a function of the hierarchical instances and the signals which interconnect them.
- Simulate a previously elaborated design. Simulation can be carried out using a batch interface, and interactive text interface, or an interactive window-based interface. Full source debugging of VHDL models is supported with the window-based interface.
- Collect simulation information in a database file.
- Save the state of a simulation, so tha it can be continued at a later time.
- Obtain online help information for all Leapfrog error messages.

3.4 The Synergy Synthesis Tool

The Synergy VHDL Synthesizer and Optimizer allows the user to perform synthesis and optimization tasks in a top-down design process. A Register Transfer Level (RTL) or mixed-level design whose functionality has been verified using a simulation tool (like Leapfrog) may then be synthesized. The results of synthesis are then optimized while attempting to meet the user-specified area and timing goals, and the optimized design is then mapped to a technology-specific library of components.

The VHDL Synthesizer also allows the user to make a number of diffrent types of synthesis run. For example, a graphical trade-off curve may be generated, which shows a range of possible circuits which may be synthesized from a particular design.

When the synthesis run is complete, the user may view the optimized schematic layout and netlist, as well as a number of reports which summarize the cost and timing attributes of the synthesized design.

The Synergy synthesis tool is particularly useful for Application Specific Integrated Circuit (ASIC) design.

3.5 The Synopsys Design Compiler

The Synopsys software system provides a full suite of tools to support the high-level design process. In particular, the Synopsys synthesis tools translate VHDL code and other high-level circuit descriptions into optimized gate-level netlists. The optimization process trades off timing, area and testability goals which are set by the user. The tools include the VHDL Compiler, the Design Compiler and the Test Compiler. A graphical user interface for all the synthesis tools is provided by the Design Analyzer program.

Synopsys is most often used for the synthesis using Field Programmable Gate Array (FPGA) technology and ASICs.

The use of the different tools at various stages of the design process can be summed up in Table 3.1.

3.6 Integrated Tools Tutorial

The Cadence software provides comprehensive on-line help information, and also extensive tutorials for each of the individual tools. However, there is little information provided for the use of these tools in an integrated fashion, for the VHDL design process. One of the objectives of this project was to write such a tutorial, and this can be found in a separate document.

Design Step	*Tool*	*Purpose*
Design Entry	Schematic Editor	Creation and editing of schematic layout
	Language Sensitive Editor	Creation and editing of VHDL source code
Simulation	Language Sensitive Editor	Creation and editing of Test Bench code
	Leapfrog	Simulation of design
Synthesis and Optimization	Synergy	Optimization and Layout
	Synopsis	Optimization and Layout

Table 3.1: Use of automated tools in the design process

The tutorial uses a very simple VHDL design, and emphasizes the use of all the tools together at various stages of the design. The central tool used is the VHDL Tool Box, from which the editors, simulation and synthesis tools are launched.

Chapter 4

CRYPTOGRAPHY AND DATA ENCRYPTION

4.1 Secret Key Cryptography Algorithms

Secret-key cryptography remains extremely important and is the subject of much ongoing study and research. One important algorithm for secret-key encryption is the Data Encryption Algorithm (DEA) which will be described later.

The secret key, or *product cipher, E* is the composition of a number of functions, each of which may involve a substitution or transposition.

The famous ENIGMA machine used by Germany and Japan were of the secret key type, and were multiple rotor type machines, where each rotor represented one of the functions mentioned above.

4.1.1 Substitution - Permutation ciphers

One proposed method of composing functions is to apply a transposition, followed by an alternating sequence of substitution and simple linear operations. The example in Figure 4.1 shows how this principle is applied to small blocks, although in practice much larger ones would be used.

Initial list	A	B	C	D	E	F
After substitution S1	B	C	D	F	G	H
After permutation P1	H	F	G	C	D	B
After a further substition S1	I	G	H	E	F	D
After permutation P1	G	I	E	H	D	F

Figure 4.1: Example of function composition

The first three letters are substituted by moving each letter one place to the right in the alphabet, and the second three by moving each letter two places right. The first permutation, P1, is (16)(24)(35). The second, P2, is (12)(34)(56). These permutations and their inverses are identical.

4.1.2 The Data Encryption Standard (DES)

In 1977, the American National Bureau of Standards announced a Data Encryption Standard to be used in unclassified United States Government applications.

The algorithms described in this standard formed the basis for that adopted by Australia and almost all other governments in the world, with the intention that the *Data Encryption Algorithm (DEA)* should be available to the worldwide banking and commercial networks as an ISO standard. However, this was not to be, because Diffie and

Hellmann, the inventors of public key cryptography, predicted that the algorithm would be vulnerable to attack by a special purpose machine with one million chips, which could be built for around $20 million, and which could break the code in about one day. They also predicted that by 1990, hardware speeds would have improved so much that a 56-bit key would no longer be secure.

As they predicted, machine speeds have indeed caught up with the theory, and now a 56-bit key is almost certainly too short.

Nevertheless, the DEA algorithm is still used heavily in today's technology.

The encryption algorithm was developed at IBM, and enciphers 64-bit blocks of data, using a 56-bit key. The algorithm is as follows.

An input block T is first transposed under an initial permutation IP, giving $T_0 = IP(T)$. After it has passed through 16 iterations of a function f, it is transposed under the inverse permutation IP_{-1} to give the final result. The permutations IP and IP-1 are given by lookup tables.

Between the initial and final transpositions, the algorithm performs 16 iterations of a function f, which combines substitution and transposition. Let T_i denote the result of the i-th iteration and let L_i and R_i denote the left and right halves of T_i respectively, that is, $T_i = L_i\ R_i$, where,

$$L_i = t_1 \ \ldots \ t_{32}$$
$$R_i = t_{33} \ \ldots \ t_{64}$$

Then,

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \text{ EOR } f(R_{i-1}, K_i)$$

The value K_i is a 48-bit key, which is different for each iteration. It is the algorithm for the generation of these 16 key values which will be the subject of this project.

4.1.3 The Key Generation Algorithm

Each iteration, i, uses a different 48-bit key K_i which is derived from the initial key K. The algorithm for the generation of the 16 keys is shown in Figure 4.2, and will now be described in detail.

K is input as a 64-bit block, with 8 parity bits in positions 8, 16, ..., 64. The permutation PC-1 discards the eight parity bits, and then transposes the remaining 56 bits as shown in Table 4.1, which should be read from left-to-right and from top-to-bottom.

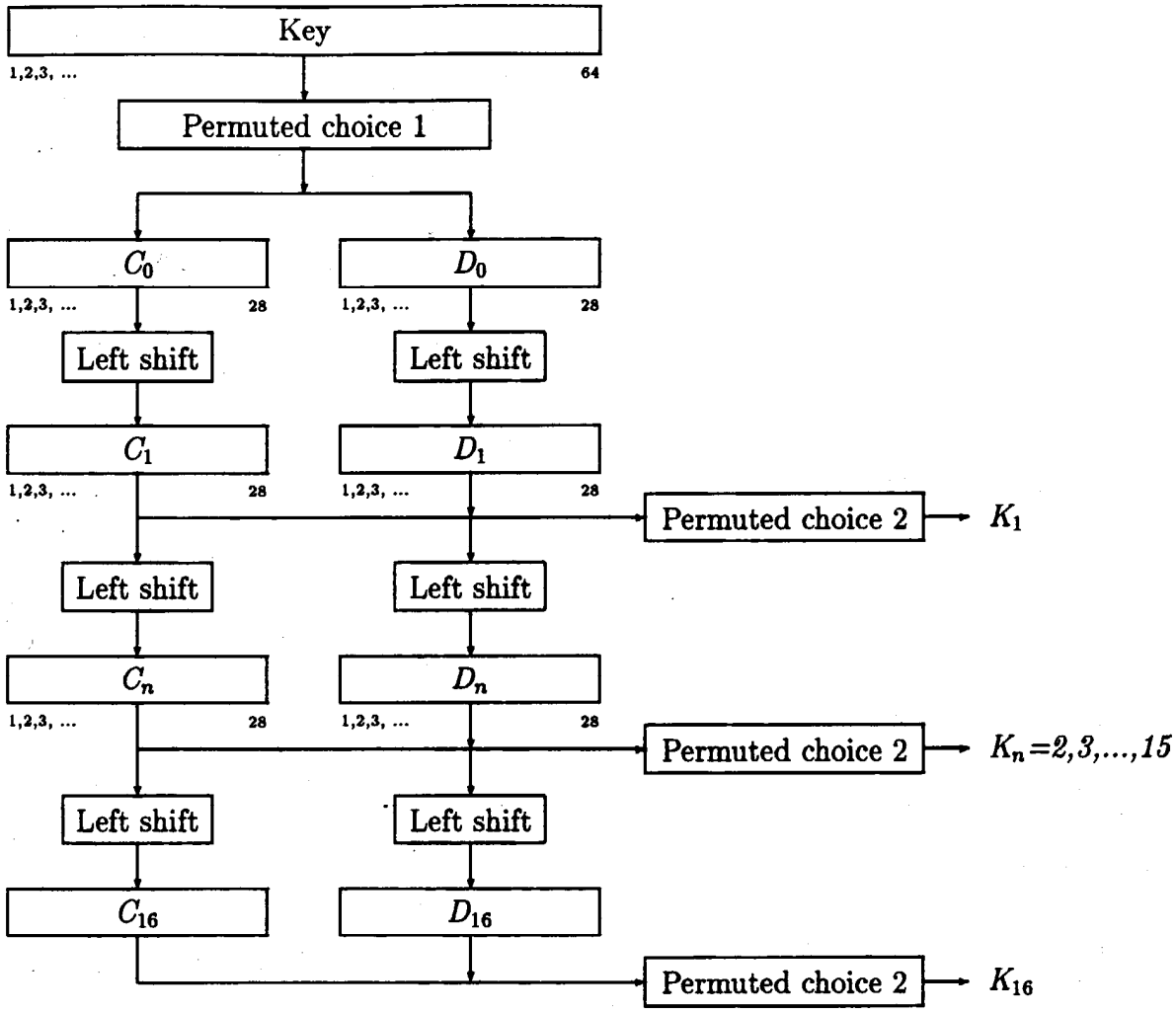


Figure 4.2: Key Generation Algorithm for Data Encryption Algorithm

The resulting 56 bits, PC-1(K), are then split into two halves, C and D, each of 28 bits. These are used to derive each key K_i.

Letting C_i and D_i denote the values used to derive K_i, we have,

$$C_i = \mathrm{LS}_i(C_{i-1}),$$
$$D_i = \mathrm{LS}_i(D_{i-1}),$$

where LS_i is a left circular shift. The number of places to be shifted varies from iteration to teration, as shown in Table 4.2.

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Table 4.1: Key permutation PC-1

Iteration	*Number of i*	*Iteration*	*Number of i*
1	1	9	1
2	1	10	2
3	2	11	2
4	2	12	2
5	2	13	2
6	2	14	2
7	2	15	2
8	2	16	1

Table 4.2: Key schedule of left shifts LS

The resulting values of C_i and D_i are joined, and subjected to a permutation PC-2, which removes the bits at positions 9, 18, 22, 25, 35, 38 43 and 54, and permutes the remaining bits according to Table 4.3.

This means that the key K_i is given by,

$$K_i = \text{PC-2}(C_i D_i)$$

4.1.4 'C' implementation of Data Encryption Algorithm

An implementation of the Data Encryption Algorithm written in the 'C' programming language by Lawrence Brown at the University of New South Wales, Australia, was used as a guide to how the algorithm could be implemented, and to observe it in action. Complete listings of the 'C' source code are provided in Appendix A.

The program takes as its input parameters the following :

- **-e|-d** - this parameter indicates whether the program is being run in encryption or decryption mode. One, and only one, of these two must be provided.

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Table 4.3: Key permutation PC-2

- **-h** - this parameter indicates that the master key will be provided in hexadecimal mode, and must be 16 digits long. If this parameter is omitted, the key will be in ASCII, and will be 8 characters long.
- **-k key** - this parameter is mandatory, and denotes the master key to be used for the algorithm.
- **-b** - indicates that Electronic Code Book (ECB) mode is to be used. If it is omitted, the Cipher Block Chaining (CBC) mode is used.
- < **filename** - indicates that the data to be encrypted or dcrypted is contained in the file 'filename'.
- > **filename** - indicates that the output from the program should be directed to the file 'filename'.

The program first of all takes all the command line parameters, and verifies that they are valid, checking, for example, that the input file and output file are different. The master key is then sent to the key generation function, which is executed once. The main encryption or decryption process is continually run, using data blocks of 64 bits, until no more bits remain. Each 64-bit input block will result in an output block of 64 bits also.

The program was tested with the simple text file shown in Figure 4.3, which was called *testfile*.

When the program is run with the command line **esd -e -k 11111111 < testfile > fu**, the output to the file *fu* contained a large number of non-printable characters, but was completely different from the original.

This file was then passed through the program in deciphering mode, using the command line **esd -d -k 11111111 < fu ¿ result**, with the output being sent to the file *result*. As should be the case, the two files *testfile* and *result* were identical, proving the functionality of the program.

```
Severiano Ballesteros     37    Spain    -8
Jose Maria Olazabal       26    Spain    -7
Darren Clarke             24    GB       -5
Howard Clarke             35    GB       -5
Frank Nobilo              29    NZ       -3
Fred Couples              31    USA      -3
John Daly                 27    USA      -1
Christy O'Connor Jr.      45    Ire      -1
```

Figure 4.3: Input file *testfile*

Chapter 5

IMPLEMENTATION OF KEY GENERATION ALGORITHM

5.1 VHDL Description of Key Generation Algorithm

5.1.1 Top level VHDL entity model

The input to the key generation algorithm consists of a master key of 64 bits. There are 16 outputs, one for each of the generated subkeys, each consisting of 48 bits. The algorithm can therefore be modelled at the top level by a module containing a single 64-bit input, and 16 outputs, each of 48 bits, as shown in Figure 5.1.

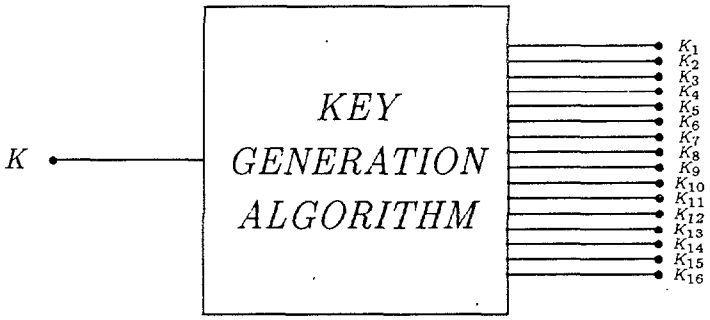


Figure 5.1: Black Box model of Key Generation Algorithm

The 64-bit input may be modelled using a 64-element array, where each element is of type *std_ulogic*. This differs slightly from the data type *bit* in that, instead of having only two logic states, 0 and 1, this data type offers 9 states, which are shown in Table 5.1.

Logic Value	*Logic State*
U	uninitialized
X	forced unknown
0	forced 0
1	forced 1
Z	high impedance
W	weak unknown
L	weak 0
H	weak 1
-	don't care

Table 5.1: Logic states available with the *std_ulogic* data type

The data type *std_ulogic_vector*, which represents an array of *std_ulogic* elements, will be used for the input array. Each of the outputs may be represented by a 48-element array, where each element is again of type *std_ulogic*. The VHDL entity description for the top-level module is shown in Figure 5.2.

```
ENTITY keygen IS
    PORT(
        K : IN  std_ulogic_vector(0 to 63);
        K1 : OUT  std_ulogic_vector(0 to 47);
        K2 : OUT  std_ulogic_vector(0 to 47);
        K3 : OUT  std_ulogic_vector(0 to 47);
        K4 : OUT  std_ulogic_vector(0 to 47);
        K5 : OUT  std_ulogic_vector(0 to 47);
        K6 : OUT  std_ulogic_vector(0 to 47);
        K7 : OUT  std_ulogic_vector(0 to 47);
        K8 : OUT  std_ulogic_vector(0 to 47);
        K9 : OUT  std_ulogic_vector(0 to 47);
        K10 : OUT  std_ulogic_vector(0 to 47);
        K11 : OUT  std_ulogic_vector(0 to 47);
        K12 : OUT  std_ulogic_vector(0 to 47);
        K13 : OUT  std_ulogic_vector(0 to 47);
        K14 : OUT  std_ulogic_vector(0 to 47);
        K15 : OUT  std_ulogic_vector(0 to 47);
        K16 : OUT  std_ulogic_vector(0 to 47));
END keygen;
```

Figure 5.2: VHDL code for top level entity

5.1.2 Structure of Top Level Entity

The key generation algorithm may be split into three distinct units. The first is PERM1 wich takes the initial master key of 64 bits, and performs the permutation PC-1 on them, to give a 56-bit output. The second module, GENKEYS takes this 56-bit array, and by repeatedly splitting, shifting and concatenating, it produces 16 outputs, each of 48 bits. Each of these outputs is then pased to an instance of the third module PERM2 which transposes the bits according to the permutation PC-2, to give 16 48-bit outputs, which are the final subkeys. This structure is shown in Figure 5.3.

The design for each individual module will be described first, and then the VHDL design for the overall structure will follow.

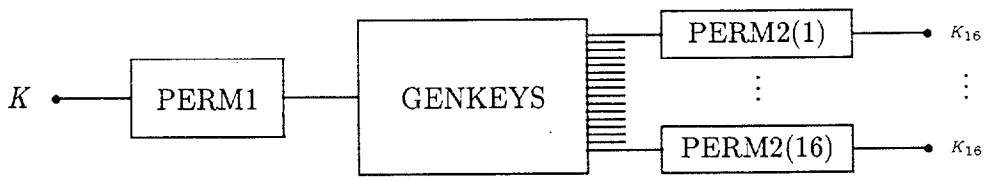


Figure 5.3: Structural diagram of Key Generation Algorithm

5.1.3 Permutation Algorithm PC-1

The permutation algorithm for PC-1 takes the supplied master key consisting of 64 bits, and transposes this into an output consisting of 56 bits. The bits at positions 8, 16, 24, 32, 40, 48, 56 and 64 are dropped at this stage. Both the input and output to this module are modelled as being of type *std_ulogic_vector*, with the input having 64 eleents, and the output 56. The entity declaration for PC-1 is shown in Figure 5.4.

```
ENTITY perm1 IS
    PORT(
        k_master : IN  std_ulogic_vector(0 to 63);
        permed   : OUT std_ulogic_vector(0 to 55);
END perm1;
```

Figure 5.4: VHDL entity declaration for permutation PC-1

The transposition is carried out according to Table 4.1 described in the previous section, the contents of which may be placed in an array constant. The behavioral description of permutation PC-1 is shown in Figure 5.5.

```
ARCHITECTURE behavior OF perm1 IS

    CONSTANT PC1 : array(0 to 55) OF integer :=
        ( 56, 48, 40, 32, 24, 16,  8,
           0, 57, 49, 41, 33, 25, 17,
           9,  1, 58, 50, 42, 34, 26,
          18, 10,  2, 59, 51, 43, 35,
          62, 54, 46, 38, 30, 22, 14,
           6, 61, 53, 45, 37, 29, 21,
          13,  5, 60, 52, 44, 36, 28,
          20, 12,  4, 27, 19, 11,  3  );
    VARIABLE temp_in : array(0 to 63) of std_ulogic_vector;
    VARIABLE temp_out : array(0 to 55) of std_ulogic_vector;

    BEGIN
        temp_in := k_master;
        for i in 0 to 55 do loop
            temp_out(i) := temp_in(PC1(i));
        end loop;
        permed <= temp_out;
    END behavior;
```

Figure 5.5: VHDL behavioral description of permutation PC-1

Note that each figure in the array is one less than that which appeared in the table

in the preceding section. This is because the algorithm numbers the bits starting from 1, whereas in the VHDL code the numbering of the elements starts from zero.

5.1.4 Permutation Algorithm PC-2

The permutation algorithm PC-2 is very similar to the one for the permuation PC-1. However, this time the input is an array of 56 bits, which is then transposed to give an output array of 48 bits. Both arrays may be modelled using the *std_ulogic_vector* data type.

The entity declaration and behavioral description of PC-2 are shown in Figure 5.6.

```
ENTITY perm2 IS
    PORT(
        k_in   : IN  BIT_VECTOR(0 to 55);
        k_out   : OUT BIT_VECTOR(0 to 47);
END perm2;

ARCHITECTURE behavior OF perm2 IS

    CONSTANT PC2 : array(0 to 47) OF integer :=
        ( 14, 17, 11, 24,  1,  5,
           3, 28, 15,  6, 21, 10,
          23, 19, 12,  4, 26,  8,
          16,  7, 27, 20, 13,  2,
          41, 52, 31, 37, 47, 55,
          30, 40, 51, 45, 33, 48,
          44, 49, 39, 56, 34, 53,
          46, 42, 50, 36, 29, 32  );

    VARIABLE temp_in : array(0 to 55) of std_ulogic_vector;
    VARIABLE temp_out : array(0 to 47) of std_ulogic_vector;

    BEGIN
        temp_in := k_in;
        for i in 0 to 47 do loop
            temp_out(i) := temp_in(PC2(i));
        end loop;
        k_out <= temp_out;
    END behavior;
```

Figure 5.6: VHDL entity and behavioral description of permutation PC-2

5.1.5 Subkey Generation Algorithm

The subkey generation algorithm involves taking the 56-bit output from PC-1, and firstly splitting it into two halves, C and D, each of 28 bits. These halves are repeatedly shifted and concatenated, to give sixteen outputs. Each of these will consist of 56 bits, and each will be used as input to an instance of PC-2. The entity declaration for the generation algorithm is shown in Figure 5.7, and the behavioral description in Figure 5.8.

```
ENTITY genkeys IS
    PORT ( inkey    : IN std_ulogic_vector(0 to 55);
           outkey1  : OUT std_ulogic_vector(0 to 55);
           outkey2  : OUT std_ulogic_vector(0 to 55);
           outkey3  : OUT std_ulogic_vector(0 to 55);
           outkey4  : OUT std_ulogic_vector(0 to 55);
           outkey5  : OUT std_ulogic_vector(0 to 55);
           outkey6  : OUT std_ulogic_vector(0 to 55);
           outkey7  : OUT std_ulogic_vector(0 to 55);
           outkey8  : OUT std_ulogic_vector(0 to 55);
           outkey9  : OUT std_ulogic_vector(0 to 55);
           outkey10 : OUT std_ulogic_vector(0 to 55);
           outkey11 : OUT std_ulogic_vector(0 to 55);
           outkey12 : OUT std_ulogic_vector(0 to 55);
           outkey13 : OUT std_ulogic_vector(0 to 55);
           outkey14 : OUT std_ulogic_vector(0 to 55);
           outkey15 : OUT std_ulogic_vector(0 to 55);
           outkey16 : OUT std_ulogic_vector(0 to 55);
END genkeys ;
```

Figure 5.7: VHDL entity declaration for key generation

5.1.6 VHDL structural code for Top Level Structure

The top level VHDL structural code firstly declares the three types of components described previously, and then declares one instance of each of PERM1 and GENKEYS, and then 16 instances of PERM2. The individual components are connected together using a number of internal signals.

The VHDL code for the structural architecture of the key generation algorithm is shown in Figure 5.9.

```
ARCHITECTURE behavior OF genkeys IS

  TYPE rotarray IS array ( 1 to 16 ) OF integer ;
  CONSTANT keyrot: rotarray := ( 1, 1, 2, 2, 2, 2, 2, 2,
                                 1, 2, 2, 2, 2, 2, 2, 1 );

  PROCEDURE shift ( variable cd : inout std_ulogic_vector(1 to 56) ;
                    constant places : in integer ) is

            VARIABLE c, d, ctemp, dtemp: std_ulogic_vector(1 to 28);
            VARIABLE newpos : integer;

            BEGIN
               c := cd(1 to 28);
               d := cd(29 to 56);
               for i in 1 to 26 loop
                  ctemp(i) := c(places + i);
                  dtemp(i) := d(places + i);
               end if;
               if places = 1 then
                 ctemp(27) := c(places + 27);
                 dtemp(27) := d(places + 27);
               else
                 ctemp(27) := c(places - 1);
                 dtemp(27) := d(places - 1);
               end if;
               ctemp(28) := c(places);
               dtemp(28) := d(places);
               cd(1 to 28) := c;
               cd(29 to 56) := d;
            END;

VARIABLE cd: vec56;

BEGIN
  cd := inkey;
  outkey1 <= shift ( cd, keyrot(1) ) ;
  outkey2 <= shift ( cd, keyrot(2) ) ;
  .
  .
  outkey16 <= shift ( cd, keyrot(16) ) ;
END behavior ;
```

Figure 5.8: VHDL behavioral description of key generation algorithm

```
ARCHITECTURE structure OF key_generation IS

COMPONENT perm1
  PORT ( k_master : IN  std_ulogic_vector(0 to 63);
         permed   : OUT std_ulogic_vector(0 to 55)  );
END COMPONENT;
COMPONENT genkeys
  PORT ( inkey  : IN  std_ulogic_vector(0 to 55);
         outkey1,  outkey2,  outkey3,  outkey4,
         outkey5,  outkey6,  outkey7,  outkey8,
         outkey9,  outkey10, outkey11, outkey12,
         outkey13, outkey14, outkey15, outkey16
                : OUT std_ulogic_vector(0 to 55)  );
END COMPONENT;
COMPONENT perm2
  PORT ( k_in  : IN  std_ulogic_vector(0 to 55);
         k_out : OUT std_ulogic_vector(0 to 47)  );
END COMPONENT;

SIGNAL s0, s1, s2, s3, s4, s5, s6 : std_ulogic_vector(0 to 55);
SIGNAL s7, s8, s9, s10, s11, s12 : std_ulogic_vector(0 to 55);
SIGNAL s13, s14, s15, s16 : std_ulogic_vector(0 to 55);

BEGIN
  P1 : perm1 PORT MAP ( k_master => K, permed   => s0  );
  GK : genkeys PORT MAP ( inkey   => s0, outkey1 => s1,
                          outkey1 => s2, outkey1 => s3,
                          outkey1 => s4, outkey1 => s5,
                          outkey1 => s6, outkey1 => s7,
                          outkey1 => s8, outkey1 => s9,
                          outkey1 => s10, outkey1 => s11,
                          outkey1 => s12, outkey1 => s13,
                          outkey1 => s14, outkey1 => s15,
                          outkey1 => s16  );
  P2_1 : perm2 PORT MAP (   k_in  => s1, k_out => K1  );
  P2_2 : perm2 PORT MAP (   k_in  => s2, k_out => K2  );
  .
  .
  P2_16 : perm2 PORT MAP (   k_in  => s16, k_out => K16  );
END structure;
```

Figure 5.9: VHDL code for KEY_GENERATION structure

5.2 Simulation

The simulation of the key generation module was carried out using the Leapfrog Simulation Tool in the Cadence Design Framework. From within the VHDL Tool Box, it is possible to create a test bench, which declares a single instance of the device to be tested, and then allows the user to provide the code which will act as a stimulus to the simulation. The key generation algorithm is a fairly straightforward algorithm, involving shift operations, and permutations. It was therefore possible to take a sample input value, for example, a key consisting of a '1' followed by sixty-three '0's, and using a manual method, to calculate the location of the '1' in each of the sixteen sub-keys.

It should also be noted at this stage, that the master key can take on a number of 'critical' values, where the generated subkeys will not be unique. For example, if the master key consists only of zero's, or has only a single '1' appearing in position 8, 16, 24, 32, 40, 48, 56, or 64 of the pattern, then it will be dropped during the first permutation, PC-1. All sixteen of the generated sub-keys will then consist entirely of zero's. While the data provided to the Data Encryption Algorithm will still be encrypted, the fact that all sixteen keys are the same may reduce the security of the encryption.

The stimulus code used to test the key generation module consists of a process which assigns an initial value of all zero's to the master key. After 50 ns, the value is changed so that the first bit is now a '1'. The process then stops.

The VHDL code for the process and its associated constants is shown in Figure 5.10.

```
constant ALLZERO : std_ulogic(0 to 63) := (others => '0');
constant CON1 : std_ulogic(0 to 63) := ('1', others => '0');

process
  begin
    K <= ALLZERO after 10 ns;
    wait for 50 ns;
    K <= CON1 after 10 ns;
    wait
  end process;
```

Figure 5.10: VHDL stimulus for key generation stimulus

The expected output from the simulation, in terms of the output values, is shown in Table 5.2.

Subkey Number	*Position of '1'*
1	20
2	10
3	16
4	24
5	8
6	17
7	4
8	none
9	11
10	14
11	2
12	9
13	23
14	3
15	none
16	18

Table 5.2: Predicted results from simulation of key generation module

5.3 Synthesis

The synthesis tool initially used for synthesizing the key generation design was the Synergy Synthesizer and Optimizer in the Cadence Design Framework. Using this tool, the design for the Key Generation Algorithm was analysed and elaborated to determine whether or not the design could be synthesized. During this process, a number of problems were encountered.

The first problem encountered was that the synthesis run took an inordinately long time. After a number of hours of processing, there were still no results, so it was decided to abandon this run, and to re-examine the design to see if it could be simplified in any way. This would make the resulting circuit less complex, and should reduce the synthesis time.

Each module was examined in turn, and the modifications are described in the following sections.

5.3.1 Modifications to PERM1

When the VHDL code for perm1 was examined, it appeared very straightforward. However, using an array to store the lookup table values, causes the synthesizer to create

some type of memory device to store this information. This would make the synthesis run much more complicated, and also cause the resulting circuit to be larger and more expensive. It was decided that the data in the lookup table could be incorporated into the body of the VHDL code for the module, and the loop replaced by a number of signal assignment statements. Although this would make the VHDL code appear longer and more cumbersome, it would in fact lead to a much simpler circuit.

A snippet from the modified VHDL code for the module PERM1 is shown in Figure 5.11.

```
ARCHITECTURE behavior OF perm1 IS

    BEGIN
        permed(0)  <= k_master(56);
        permed(1)  <= k_master(48);
        permed(2)  <= k_master(40);
        .
        .
        .
        permed(53) <= k_master(19);
        permed(54) <= k_master(11);
        permed(55) <= k_master(3);
    END behavior;
```

Figure 5.11: Modified VHDL behavioral description of permutation PC-1

5.3.2 Modifications to PERM2

The same philosophy could be applied to PERM2 as was applied to PERM1. The array was removed, and the loop replaced by a series of signal assignment statements, a snippet of which is shown in Figure 5.12.

5.3.3 Modifications to GENKEYS

Within the GENKEYS module, a major problem was identified as being the use of variables to store a large number of intermediate values. As before, this would cause the synthesizer to attempt to generate a number of memory devices. The first stage of modification was to replace the procedure in the VHDL code with a new entity called LEFT_SHIFT. This required one 56-bit input, and one 56-bit output, where the output was equal to the input shifted by the required number of places. This value was to be provided by a generic which would be passed to the component upon instantiation. The

```
ARCHITECTURE behavior OF perm2 IS

    BEGIN
        k_out(0)  <=  k_in(14);
        k_out(1)  <=  k_in(17);
        k_out(2)  <=  k_in(11);
        .
        .
        .
        k_out(45) <=  k_in(36);
        k_out(46) <=  k_in(29);
        k_out(47) <=  k_in(32);
    END behavior;
```

Figure 5.12: Modified VHDL behavioral description of permutation PC-2

required output was then determined by using a WITH..SELECT statement, the action of which was dependant on the value of the generic.

The entity declaration and behavioral description of LEFT_SHIFT are shown in Figure 5.13.

```
ENTITY left_shift IS
    GENERIC ( places : IN integer := 1 );
    PORT    ( cd_in  : IN std_ulogic_vector(0 to 55);
              cd_out : OUT std_ulogic_vector(0 to 55)  );
END left_shift;

ARCHITECTURE behavior OF left_shift IS

    BEGIN
        WITH places SELECT
          cd_out <= cd_in(1 to 27) & cd_in(0) &
                      cd_in(29 to 55) & cd_in(28) WHEN 1,
                    cd_in(2 to 27) & cd_in(0 to 1) &
                      cd_in(30 to 55) & cd_in(28 to 29) WHEN 2,
                    cd_in WHEN others;
    END behavior;
```

Figure 5.13: VHDL entity declaration and behavioral description of LEFT_SHIFT

The GENKEYS module can now be described in terms of sixteen instances of this entity placed one after the other. The input to the first instance will be the output from the PERM1 entity. The output from the first instance will serve as input for the second, but will also be connected to the first output port. In each instance, the output will form

both the input for the next instance, and a link to one of the output ports. Part of the modified structural description for the GENKEYS module is shown in Figure 5.14.

```
ARCHITECTURE structure OF genkeys IS
    COMPONENT left_shift IS
        GENERIC ( places : IN integer := 1 );
        PORT    ( cd_in  : IN std_ulogic_vector(0 to 55);
                  cd_out : OUT std_ulogic_vector(0 to 55)  );
    END COMPONENT;

BEGIN
  LS1  : left_shift_comb
       generic map ( places => 1 )
       port map ( cd_in  => inkey,
                  cd_out => ik1 ) ;
  LS2  : left_shift_comb
       generic map ( places => 1 )
       port map ( cd_in  => ik1,
                  cd_out => ik2 );
  .
  .
  .
  LS15 : left_shift_comb
       generic map ( places => 2 )
       port map ( cd_in  => ik14,
                  cd_out => ik15 );
  LS16 : left_shift_comb
       generic map ( places => 1 )
       port map ( cd_in  => ik15,
                  cd_out => ik16 );
  outkey1  <=  ik1;
  outkey2  <=  ik2;
  .
  .
  .
  outkey16 <=  ik16;
END behavior ;
```

Figure 5.14: Modified description of key generation algorithm

It was also decided at this stage that it would be beneficial to use the Synopsys Design Analyzer for the synthesis run, rather than the Synergy package. The Synopsys system offers a more powerful synthesis tool, and also allows the use of FPGA technology.

5.4 Results

The Synopsys Design Compiler was set up to generate layouts using the X4000 component libraries from XiLinx. The VHDL source files were analysed and optimised from the lowest level up, the order being LEFT_SHIFT_COMB, GENKEYS_COMB, PERM1_COMB, PERM2_COMB and KEY_GEN_COMB. Any module which contained more than one instance of any particular lower level component, had to be *uniquified.* This enabled each of the instances to be treated as a component in its own right during the synthesis run, enabling each to be optimized in different ways, if required. After optimization, the next stage was to convert the FPGA cells to standard logic gates.

Of the five source files, two of them generated structural schematic diagrams, these being GENKEYS_COMB, and KEY_GEN_COMB. Their schematic block diagrams are shown in Figure 5.15 and Figure 5.16.

The remaining three files all produced gate-level schematic diagrams which can be seen in Figures 5.17, 5.18 and 5.19. The nature of these three lower-level entities is very similar, in that each of them performs only a permutation, or a shift procedure. This meant that the schematic diagrams in their simplest forms were made up of no components, but simply direct connections between input ports and output ports. However, for some technologies, this is not acceptable, and some sort of buffer must be introduced into the circuit betweem each output and its associated input. This took the form of a pair of inverters placed on each path. The area and timing attributes for the five modules are shown in Table 5.3.

Module	*Combinational Area*	*Non-combinational Area*	*Delay*
perm1_comb	0.00	0.00	3.60
perm2_comb	0.00	0.00	3.60
left_shift	0.00	0.00	3.60
genkeys	0.00	0.00	67.95
key_gen	0.00	0.00	75.15

Table 5.3: Area and timing attributes for key generation modules

All the modules are reported as having an area of zero. This is because, as stated before, the circuit can be made up entirely of direct connections between input and output pins.

The overall delay time of 75.15 us comes about due to the longest path having to pass through 16 instances of left_shift, and then also perm1 and perm2.

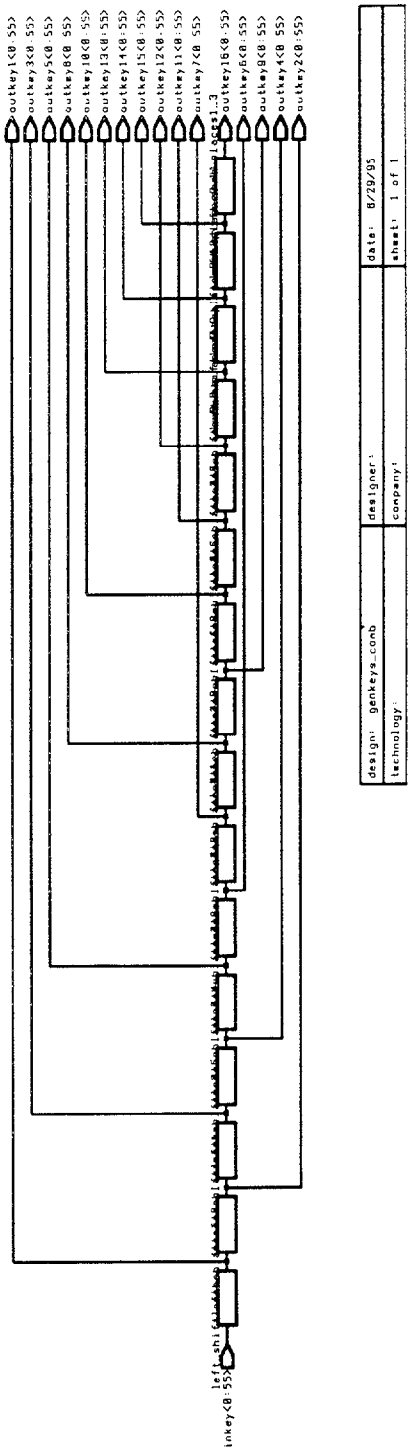


Figure 5.15: Schematic block layout of GENKEYS_COMB

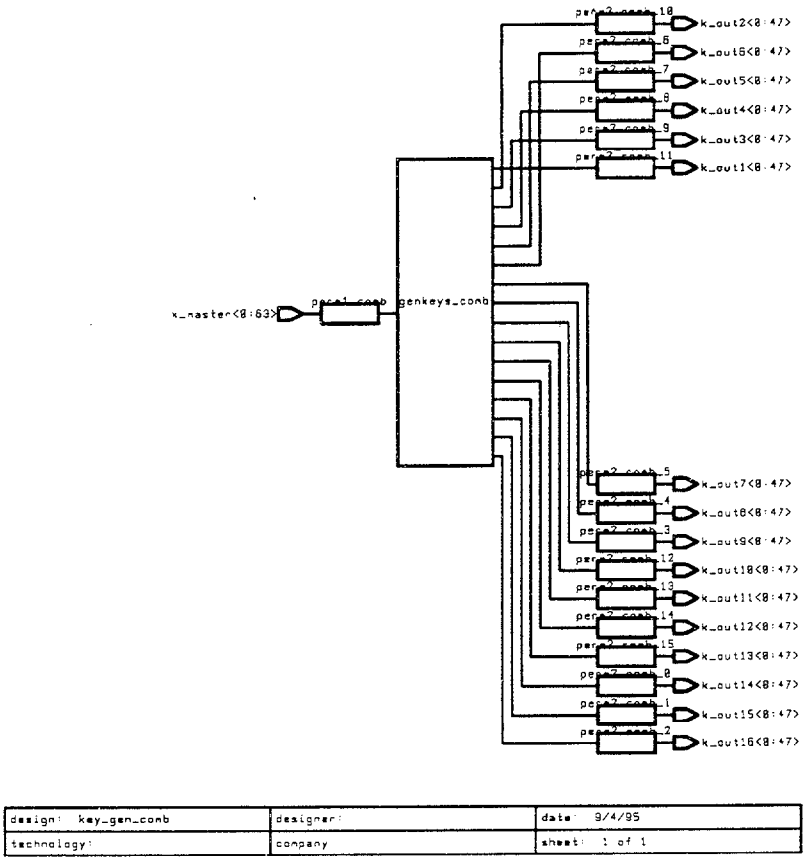


Figure 5.16: Schematic block layout of KEY_GEN_COMB

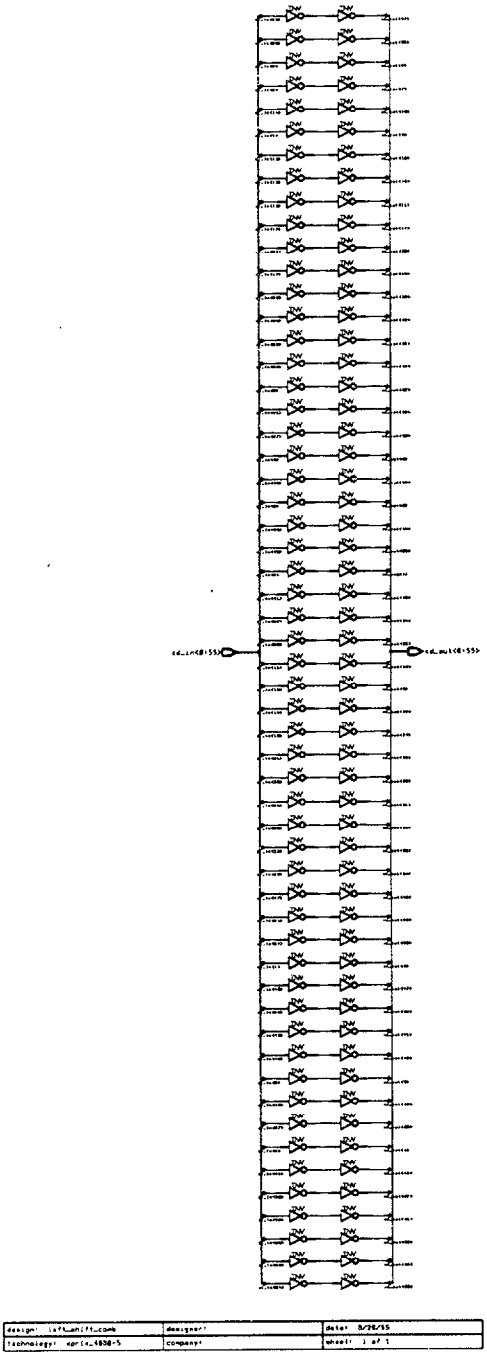


Figure 5.17: Schematic gate layout of LEFT_SHIFT_COMB

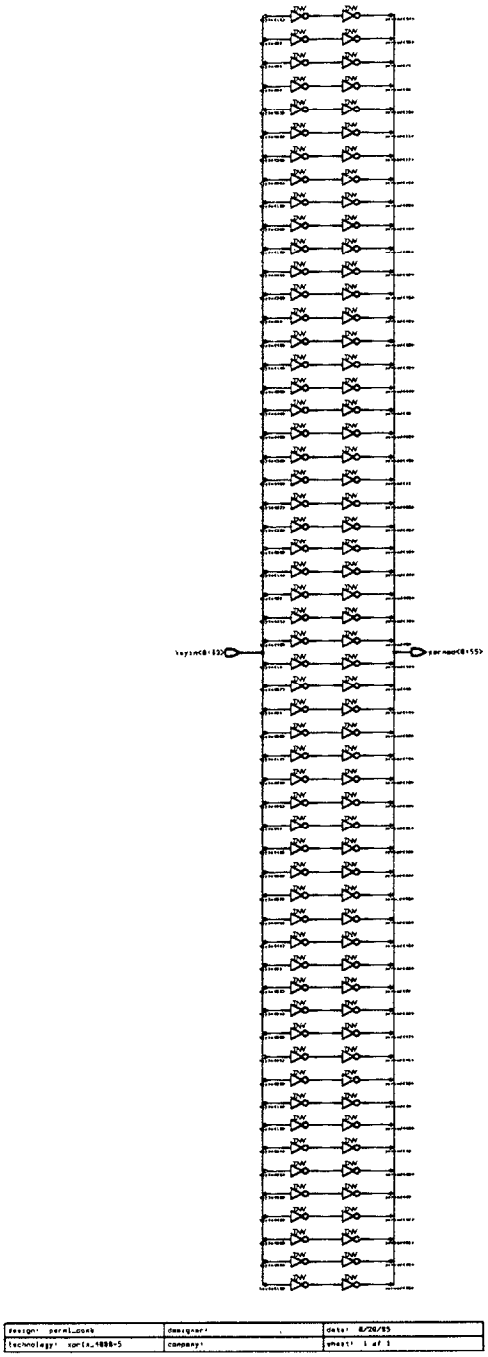


Figure 5.18: Schematic gate layout of PERM1_COMB

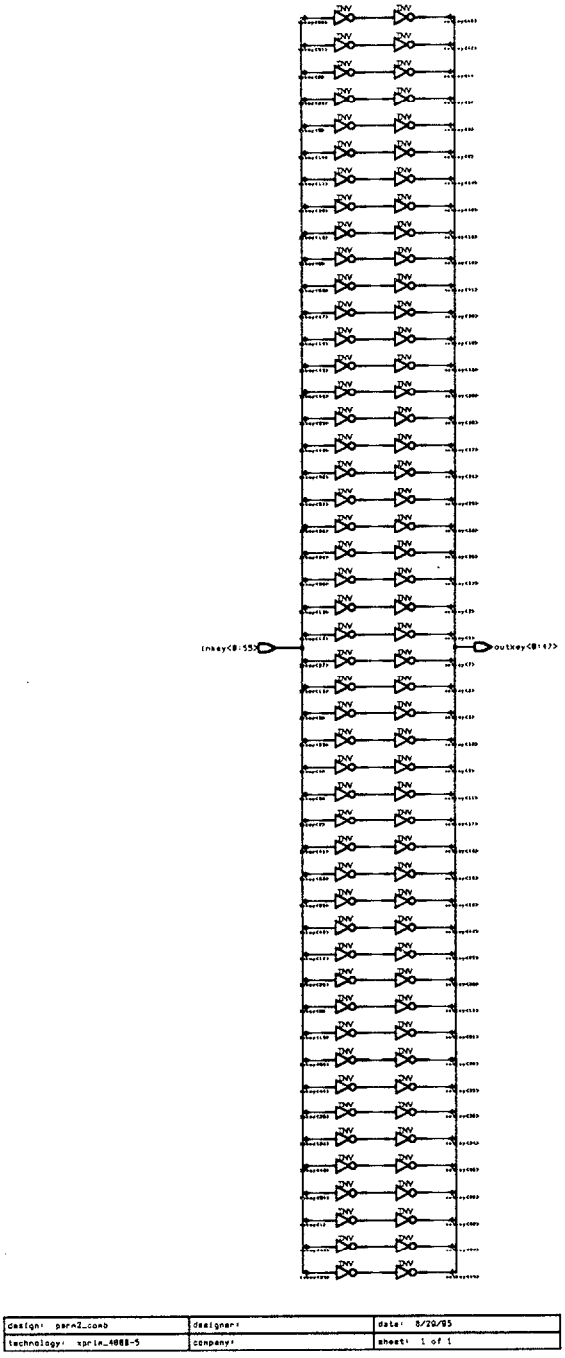


Figure 5.19: Schematic gate layout of PERM2_COMB

Chapter 6

IMPLEMENTATION OF DATA ENCRYPTION ALGORITHM

6.1 Circuit Structure

The sub-key generation algorithm described and modelled in the previous chapter forms only a part of the Data Encryption Algorithm. In order to satisfy the complete algorithm, the key generation module must be placed in a circuit along with a module which will carry out the encryption/decryption algorithm, and also with buffers which will allow the input and output of a stream of data, and convert it to the form required by the sub-module. A block diagram of such a circuit is shown in Figure 6.1.

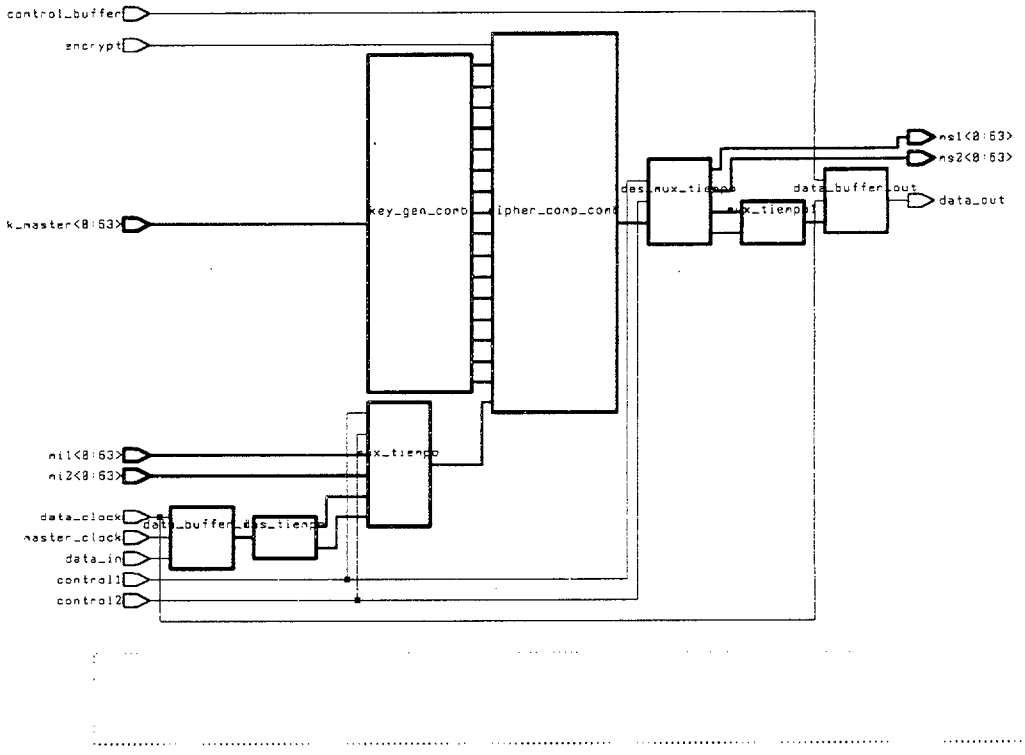


Figure 6.1: Schematic block layout of complete design

6.1.1 DATA_BUFFER_IN

The DATA_BUFFER_IN component reads in the input data at the rate of 160 bits per clock cycle (*master_clock*). The input data buffer stores each bit as part of a 160-bit array. At the end of the clock cycle, when the array is full, the entire array is passed to the output of the DATA_BUFFER_IN module.

6.1.2 DES_TIEMPO

The DES_TIEMPO component takes as input an array of 160 bits, which are grouped into blocks of ten. In each block of ten, bits 9 and 10 are discarded, leaving 128 bits in all. The first group of eight come from channel 1 and the second from channel 2, and so on alternately. The DES_TIEMPO module directs the groups of eight to each of two arrays in turn, and thus creates two 64-bit arrays. These are passed as output.

6.1.3 MUX_TIEMPO

The MUX_TIEMPO component has four 64-bit inputs and two single bit control inputs. The inputs mi1 and mi2 are 64-bit inputs from the microprocessor, while the inputs ri1 and ri2 are 64-bit inputs from the data stream. The first of the control signals, control1, indicaes whether the data is to be read from the microprocessor, or the data stream. Having established this, the second control bit, control2, switches between the required two inputs, and sends them to the output port, one after the other. These then comprise two blocks of data passed to the DES encryption module.

6.1.4 CIPHER_BLOCK

The CIPHER_BLOCK component, which is made up entirely of combinational logic takes the data input, along with the 16 sub-keys from the key generation moodule. The output of the CIPHER_BLOCK is the encrypted or decrypted data, depending on the value of the *encrypt* input.

6.1.5 DES_MUX_TIEMPO

The DES_MUX_TIEMPO module has a single 64-bit input, and two single-bit control outputs. The first control bit, control1, indicates whether the output should be sent to the microprocessor outputs, ms1 and ms2, or the data stream outputs, rs1 and rs2. The second control bit switches between the required output ports, and sends alternate blocks to each of the outputs.

6.1.6 MUX_TIEMPO1

The MUX_TIEMPO1 module takes as intput two 64-bit arrays, ri1 and ri2. From these it creates a 160-bit array comprising of 8 bits from ri1, followed by two '1's, then 8 bits from ri2, followed by two more '1's, and so on.

6.1.7 DATA_BUFFER_OUT

The DATA_BUFFER_OUT module takes as input a single array of 160 bits, the data_clock, and a single bit control input, control_buffer. The control_buffer input triggers the buffer to begin sending bits to the output stream, one bit at a time, which it does once every clock cycle of the data_clock input.

6.2 Simulation

The simulation of the complete Data Encryption Algorithm design was much more complex than that required for the key generation module. In all, five processes were required in the stimulus code.

6.2.1 Process *master_clock_driver*

This process was used to provide the waveform for the master clock, which had a period of 125 micro-seconds. The waveform, therefore was set to 1 at time t=0, chnaged to zero after 62.5 micro-seconds, and then back to '1' again after a further 62.5 micro-seconds. The VHDL code for this process is ahown in Figure 6.2.

```
master_clock_driver : PROCESS
  BEGIN
    master_clock <= '1', '0' after 62.5 us;
    WAIT FOR 125 us;
  END PROCESS
```

Figure 6.2: VHDL code for *master_clock_driver* process

6.2.2 Process *data_clock_driver*

The model was required to read 160 bits of data during every cycle of the master clock, so the period of the data clock was found by dividing the period of the master clock by 160, giving 0.78125 micro-seconds. The data_clock signal was therefore set to '1' at time t=0, changed to '0' after 0.390625 microseconds, and back to '1' again a further 0.390625 micro-seconds later. The VHDL code for this process is shown in Figure 6.3.

```
data_clock_driver : PROCESS
  BEGIN
    data_clock <= '1', '0' after 0.390625 us;
    WAIT FOR 0.78125 us;
  END PROCESS
```

Figure 6.3: VHDL code for *data_clock_driver* process

6.2.3 Process *data_driver*

With every cycle of the data clock, the data_in port had to be provided with the next input bit, which was read from an array. The simulation was run twice, the first time with a single block of 64 bits, the value of which was chosen arbitrarily. The output which was produced from this simulation run was then added as a second input block of 64 bits. If the design was correct, the second output should be the same as the first input, showing that the design worked in both encryption and decrytion modes. The VHDL code for this process is shown in Figure 6.4.

```
datadriver : PROCESS
  VARIABLE x : integer := 0;
  BEGIN
    WAIT UNTIL data_clock'event AND data_clock = '1';
    IF x < 320 THEN
      IF x < 160 THEN
        data_in <= DIN1(x);
      ELSE
        data_in <= DIN2(x-160);
      END IF;
      x := x + 1;
    END IF;
  END PROCESS
```

Figure 6.4: VHDL code for *data_driver* process

6.2.4 Process *control_driver*

The control driver was used to stimulate the control inputs used on the multiplexors. These ensured that data read in from channel 1 was also output on channel 1, and that from channel 2 was output to channel 2. Both control bits were initially set to the logic value 'U' for unknown. At the start of each clock cycle, when a block of data had been completely read, the value of the signal control2 was changed to '0', then '1' and then

back to 'U', all within the space of 2 nano-seconds. This was sufficient to allow the two blocks of data to be processed by the cipher block, and the two bocks of output to be sent to the data buffer. The control_buffer input was then switched to '1' and quickly back to zero, to enable this output to be sent to the output of the data buffer. The VHDL code for this process is shown in Figure 6.5.

```
control_driver : PROCESS
  BEGIN
    control_buffer <= '0';
    control1 <= '0';
    control2 <= '0';
    WAIT FOR 1 ns;
    control1 <= '0';
    control2 <= '1';
    WAIT FOR 1 ns;
    control1 <= 'U';
    control2 <= 'U';
    control_buffer <= '1';
    WAIT FOR 1 ns;
    control_buffer <= '0';
    WAIT FOR 124.998 us;
  END PROCESS
```

Figure 6.5: VHDL code for *control_driver* process

6.2.5 Main Process

The main process was used simply to control the encrypt/decrypt input, and also to assign a value to the master key, which remained unchanged throughout the simulation. The VHDL code is shown in Figure 6.6.

```
PROCESS
  BEGIN
    encrypt <= '1', '0' after 250 us;
    k_master <= ('1', '1', '1', others => '0');
    wait;
  END PROCESS
```

Figure 6.6: VHDL code for main process

The simulation was first of all run with a single 64-bit block of data. The output, which was this data in its encrypted form, was then added to the end of the input data for the second run, giving two 64-bit blocks. The results shown in Figures 6.7, 6.8 and 6.9

show that for the first 125 us, the first block is being read in, and the output is unknown. For the next period of 125 us, the second block is being read in, and the output is the encrypted form of the first data block. For the third period of 125 us, the input is finished, and may be ignored, and the output is the decrypted form of the second block of data. As predicted, the second output block was identical to the first input block, proving the functionality of the design.

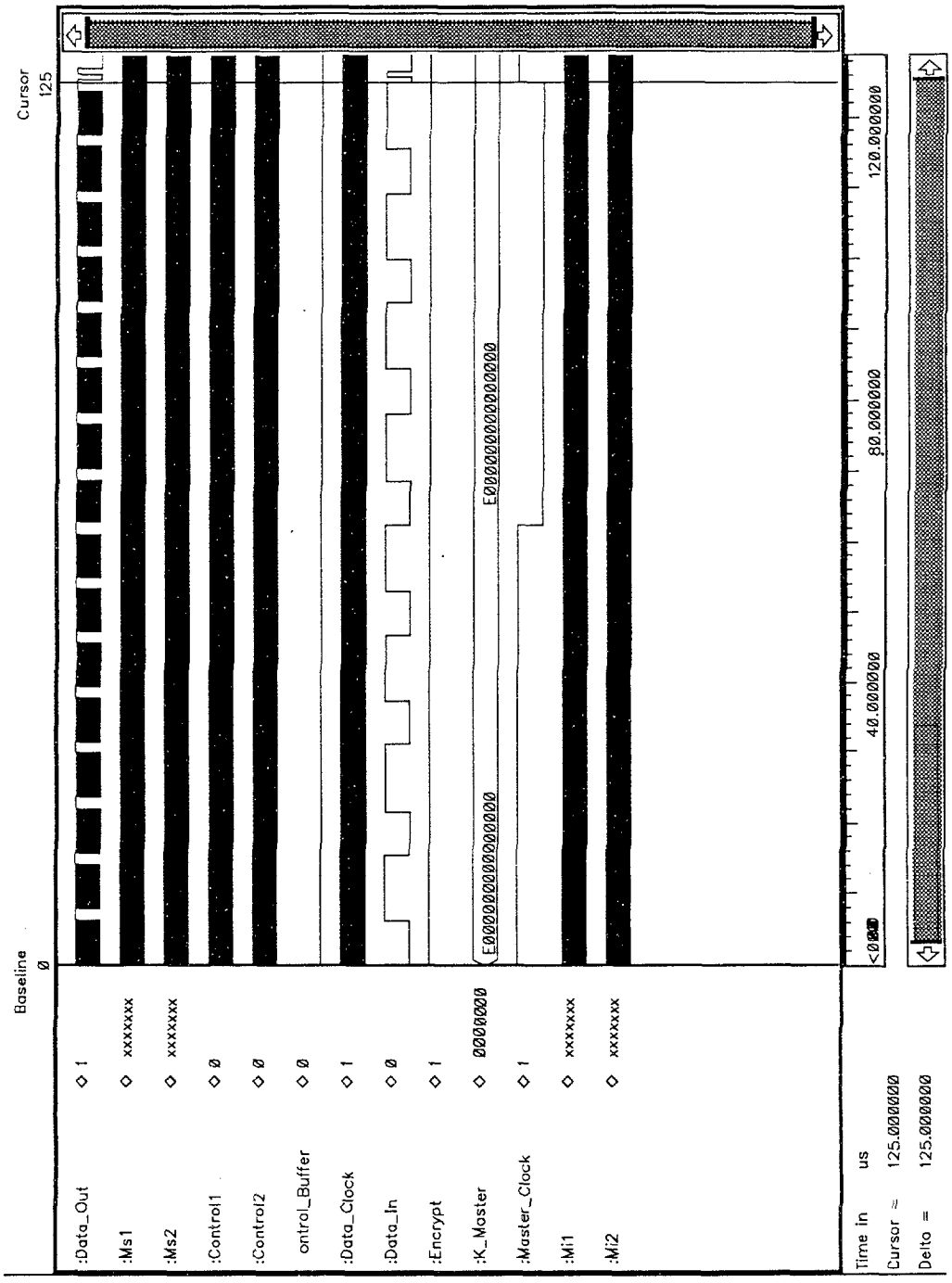


Figure 6.7: Simulation results for time t=0 to t=125us

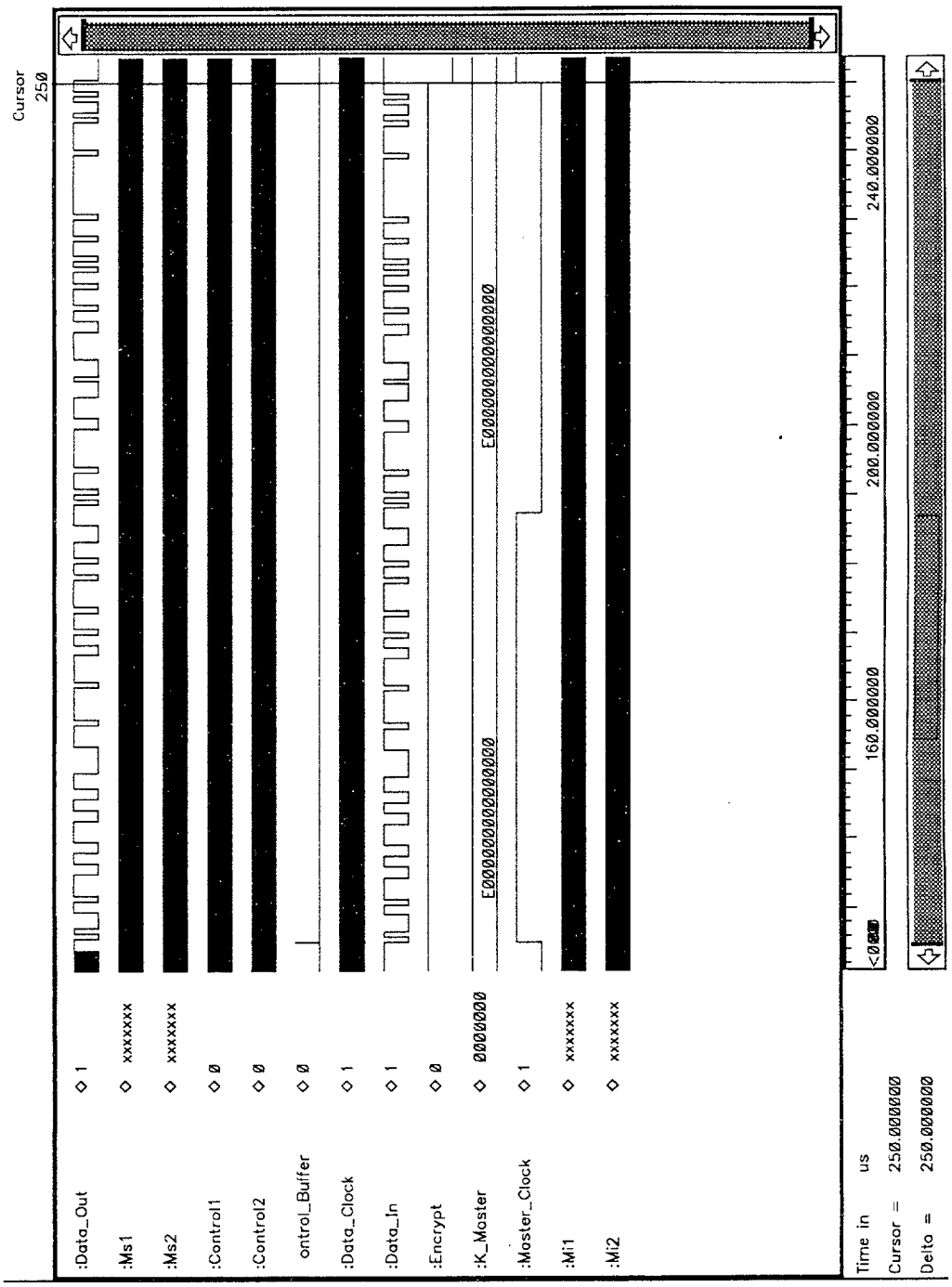


Figure 6.8: Simulation results for time t=125 to t=250us

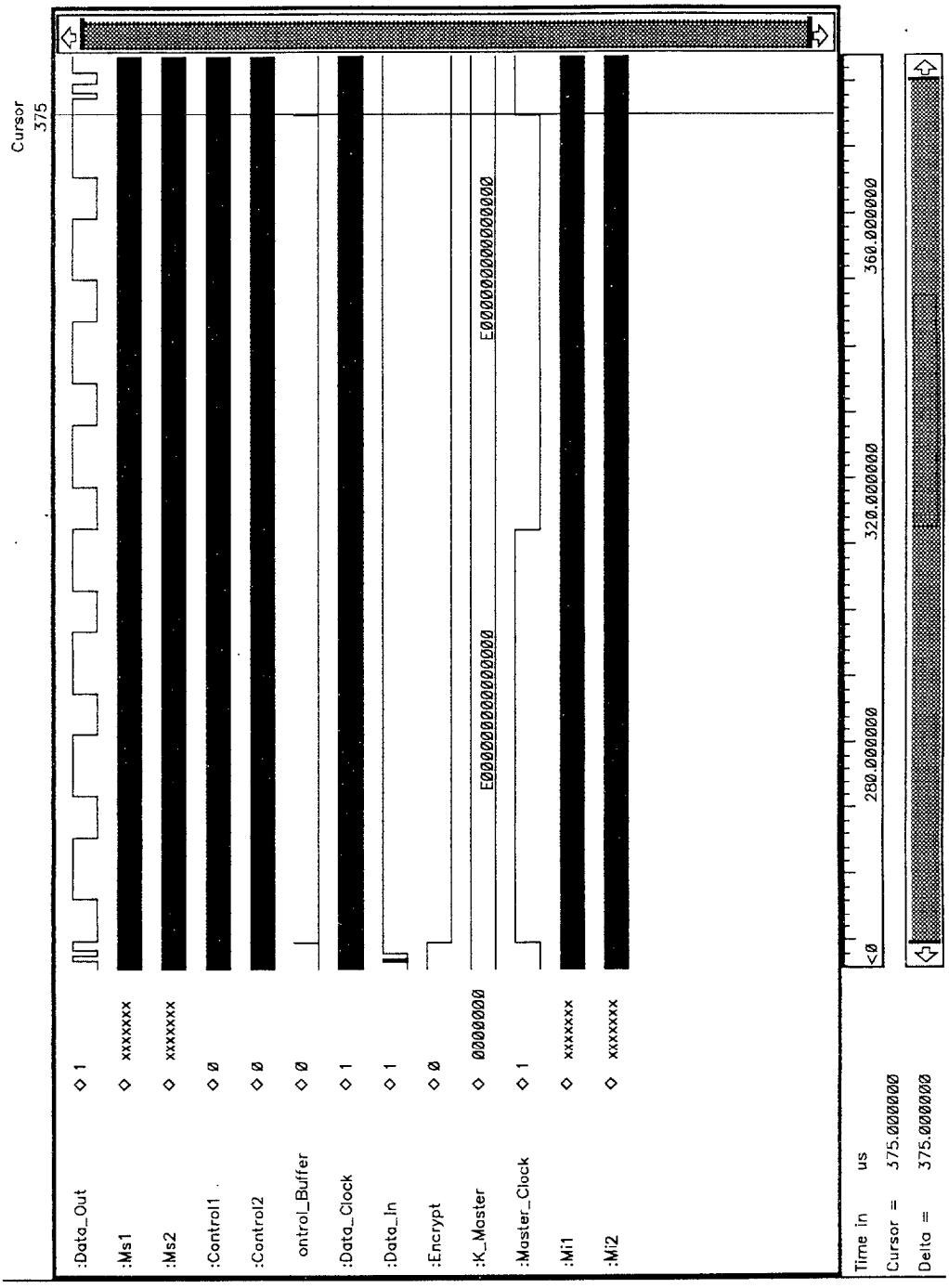


Figure 6.9: Simulation results for time t=250 to t=375us

6.3 Synthesis

The complete design was again to be synthesized using the Synopsys Design Analyzer, and the X4000 symbol libraries from XiLinx. The design was again optimized from the lowest level up, and in any design where a component was instantiated more than once, the uniquify operation carried out.

Having optimized the design, the FPGA components were translated into gates, and some area and timing analysis carried out, the results of which are shown in the following section.

6.4 Results

The area requirements and timing attributes for the various modules are shown in Table 6.1.

Module	*Combinational Area*	*Non-combinational Area*	*Total Area*	*Delay*
DATA_BUFFER_IN	440.00	258.00	698.00	39.08
DES_TIEMPO	0.00	0.00	0.00	3.60
MUX_TIEMPO	224.00	0.00	224.00	20.67
CIPHER	10576.00	0.00	10576.00	1862.08
KEY_GEN	0.00	0.00	0.00	75.15
DES_MUX_TIEMPO	2.00	0.00	2.00	6.29
MUX_TIEMPO1	0.00	0.00	0.00	3.60
DATA_BUFFER_OUT	228.00	129.50	357.50	39.08

Table 6.1: Area and timing attributes for key generation modules

As can be seen from the table, the most critical module, in terms of both area an delay time, is the cipher block. This is to be expected, since it is within this block that all the calculations take place, and the encryption and decryption of the data are carried out. In particular, one block within the cipher module stands out as being much larger and slower than all the others. This module is called POS_NEW1, and its optimized layout was found to contain some 1500 cells. It also had an area of over 500 microns, the sixteen instances of which comprise the relatively large area of the cipher block.

Chapter 7

CONCLUSIONS AND FURTHER WORK

7.1 Conclusions

VHDL provides a standard method of describing the structure and behavior of digital electronic circuits. One of its main advantages is that it contains many familiar programming language structures, and may therefore be easily understood by a wide range of people with even basic programming experience. The simulation and high-level synthesis properties of VHDL means that it is a very valuable tool as designers attempt to further reduce both the time and money spent during the design process.

One of the current problems with VHDL is that, while the language itself has been accepted as a standard, many of the tools which use it operate only on a subset of the language. This means that structures which are possible with one simulation or synthesis tool may not be available with anither. However, if such anomalies were eradicated, the language would prove to be a powerful tool indeed.

7.2 Suggestions for Further Work

As the design currently stands, it can operate in only encrypt *or* decrypt mode at any one time, and the nature of the operation is determined by an external input. It may be possible to modify the circuit so that, while it still operates in only one mode at a time, there is no external signal required, rather data is sent to a particular input port for encryption, and to a different port for decryption. This would reduce the effect of outside influence on the system.

One particular section of the design which could be further examined is the POS module described in the synthesis results. This accounts for a very large majority of both the area and timing characteristics of the design. Also, because it is instantiated sixteen times, any reduction in area or timing would be similarly multiplied.

Another suggestion for further study would be to investigate the effect of attempting to use different technology for the synthesis step. This could lead to changes in the area requirement and timing attributes of the circuit.

Bibliography

[1] Heinz Bonnenberg. *Secure Testing of VLSI Cryptographic Equipment.* PhD thesis, Swiss Federal Institute of Technology, 1993.

[2] Allen Wu Daniel Gajski, Nikil Dutt and Steve Lin. *High-level Synthesis - Introduction to Chip and System Design.* Kluwer Acedemic Publishers, 1992. .

[3] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall Software Series, 1988.

[4] Stanley Mazor and Patricia Langstraat. *A Guide to VHDL.* Kluwer Academic Publishers, 1992.

[5] Zainalabedin Navabi. *VHDL Analysis and Modelling of Digital Systems.* McGraw Hill, 1993.

Appendix A

'C' source code files for Data Encryption Algorithm

```
/*
 * des.h -specifies the interface to the DES encryption library.
 *            This library provides routines to install a key, encrypt and
 *            decrypt 64-bit data blocks. The DES Data Encryption Algorithm
 *            is a block cipher which ensures that its output is a complex
 *            function of its input and the key. A descryption of the
 *            algorithm may be found in:
 *
 *             Australian Standard AS2805.5-1985 Data Encryption Algorithm
 *
 * Descryption:
 *        The routines provided by the library are:
 *
 *        desinit()    - perform any data structure initialization needed.
 *                       Must be called before any other functions.
 *
 *        keyinit(key)- installs key for use in subsequent encryptions and
 *        Long key[2];  decryptions. A key must be installed before encryption
 *                      o4r decryption may be done.
 *
 *        endes(b)     - main DES encryption routine, this routine encrypts
 *        Long b[2];     one 64-bit block b with the current key
 *
 *        dedes(b)     - main DES decryption routine, this routine decrypts
 *        Long b[2];     one 64-bit block b with the current key
 *
 *        The 64-bit data blocks used in the algorithm are specified in two
 *              unsigned longwords (see the Long type specification below).
 *        For the purposes of implementing the DES algorithm, the bits
 *        are numbered as follows:
 *              [1 2 3 ... 32]     [33 34 35 ... 64]
 *         in         b[0]                b[1]
 *        The L (left) half is b[0], the R (right) half is b[1]
 *
 * The key is passed as a 64-bit value, of which the 56 non-parity
 *         bits are used. The parity bits are in DES bits
 *         8, 16, 24, 32, 40, 48, 56, 64 (nb: these do NOT correspond to
 *         the parity bits in ascii chars when packed in the usal way).
 */

typedef    unsigned long     Long; /* type specification used for DES data */

extern desinit();                  /* DES library intialization          */
extern keyinit();                  /* DES key installation               */
extern endes();                    /* DES Encryption                     */
extern dedes();                    /* DES Decryption                     */
```

```
/*
 *    des64.i - contains the fixed permutation and substitution tables
 *                    for a 64 bit DES implementation
 *      see "AS2805.5-1985 DEA" for table definitions and explanations
 *    Notes: Lawrence Brown <lpb@csaffa.oz>  12/87
 */

char IP[64]                          /* initial permutation P                */
={      58, 50, 42, 34, 26, 18, 10, 2,
        60, 52, 44, 36, 28, 20, 12, 4,
        62, 54, 46, 38, 30, 22, 14, 6,
        64, 56, 48, 40, 32, 24, 16, 8,
        57, 49, 41, 33, 25, 17,  9, 1,
        59, 51, 43, 35, 27, 19, 11, 3,
        61, 53, 45, 37, 29, 21, 13, 5,
        63, 55, 47, 39, 31, 23, 15, 7  };

char FP[64]                          /* final permutation F                  */
={      40, 8, 48, 16, 56, 24, 64, 32,
        39, 7, 47, 15, 55, 23, 63, 31,
        38, 6, 46, 14, 54, 22, 62, 30,
        37, 5, 45, 13, 53, 21, 61, 29,
        36, 4, 44, 12, 52, 20, 60, 28,
        35, 3, 43, 11, 51, 19, 59, 27,
        34, 2, 42, 10, 50, 18, 58, 26,
        33, 1, 41,  9, 49, 17, 57, 25  };

char E[48]={ 32,  1,  2,  3,  4,  5,     /* expansion operation matrix       */
              4,  5,  6,  7,  8,  9,     /*   nb: NOT USED                   */
              8,  9, 10, 11, 12, 13,     /*     expand() does this fn        */
             12, 13, 14, 15, 16, 17,
             16, 17, 18, 19, 20, 21,
             20, 21, 22, 23, 24, 25,
             24, 25, 26, 27, 28, 29,
             28, 29, 30, 31, 32,  1  };

char PC1[64] /* permuted choice table (key)                                   */
             /* Rewritten as a 64 bit permutation, where 4 parity bits are */
             /* permuted to DES bits 29-32 in each 32 bit longword         */
             /* and subsequently ignored                                   */
            ={  57, 49, 41, 33, 25, 17,  9,
                 1, 58, 50, 42, 34, 26, 18,
                10,  2, 59, 51, 43, 35, 27,
                19, 11,  3, 60, 52, 44, 36,
                 8, 16, 24, 32,            /* <- these are parity bits      */
                63, 55, 47, 39, 31, 23, 15,
                 7, 62, 54, 46, 38, 30, 22,
                14,  6, 61, 53, 45, 37, 29,
                21, 13,  5, 28, 20, 12,  4,
                40, 48, 56, 64 };          /* <- these are paraty bits      */
```

```
char keyrot[16]
={  1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1 };

char PC2[48]                              /* permuted choice key (table)      */
           ={  14, 17, 11, 24,  1,  5,
               3, 28, 15,  6, 21, 10,
              23, 19, 12,  4, 26,  8,
              16,  7, 27, 20, 13,  2,
              41, 52, 31, 37, 47, 55,
              30, 40, 51, 45, 33, 48,
              44, 49, 39, 56, 34, 53,
              46, 42, 50, 36, 29, 32  };

char S[8][64]           /* S-Boxes: 48-> 32 bit compression tables          */
                        /* nb: DES bits (16) select row in each table       */
                        /*     DES bits (2345) select column in row         */
={                              /* S[1]                                     */
     14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7,
      0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8,
      4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0,
     15, 12,  8,  2,  4,  9,  1,  7,  5, 11,  3, 14, 10,  0,  6, 13,
                                /* S[2]                                     */
     15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10,
      3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,  5,
      0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2, 15,
     13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,  9,
                                /* S[3]                                     */
     10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,  7, 11,  4,  2,  8,
     13,  7,  0,  9,  3,  4,  6, 10,  2,  8,  5, 14, 12, 11, 15,  1,
     13,  6,  4,  9,  8, 15,  3,  0, 11,  1,  2, 12,  5, 10, 14,  7,
      1, 10, 13,  0,  6,  9,  8,  7,  4, 15, 14,  3, 11,  5,  2, 12,
                                /* S[4]                                     */
      7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,  5, 11, 12,  4, 15,
     13,  8, 11,  5,  6, 15,  0,  3,  4,  7,  2, 12,  1, 10, 14,  9,
     10,  6,  9,  0, 12, 11,  7, 13, 15,  1,  3, 14,  5,  2,  8,  4,
      3, 15,  0,  6, 10,  1, 13,  8,  9,  4,  5, 11, 12,  7,  2, 14,
                                /* S[5]                                     */
      2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3, 15, 13,  0, 14,  9,
     14, 11,  2, 12,  4,  7, 13,  1,  5,  0, 15, 10,  3,  9,  8,  6,
      4,  2,  1, 11, 10, 13,  7,  8, 15,  9, 12,  5,  6,  3,  0, 14,
     11,  8, 12,  7,  1, 14,  2, 13,  6, 15,  0 , 9, 10,  4,  5,  3,
                                /* S[6]                                     */
     12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,  4, 14,  7,  5, 11,
     10, 15,  4,  2,  7, 12,  9,  5,  6,  1, 13, 14,  0, 11,  3,  8,
      9, 14, 15,  5,  2,  8, 12,  3,  7,  0,  4, 10,  1, 13, 11,  6,
      4,  3,  2, 12,  9,  5, 15, 10, 11, 14,  1,  7,  6,  0,  8, 13,
                                /* S[7]                                     */
      4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,  7,  5, 10,  6,  1,
     13,  0, 11,  7,  4,  9,  1, 10, 14,  3,  5, 12,  2, 15,  8,  6,
```

```
      1,  4, 11, 13, 12,  3,  7, 14, 10, 15,  6,  8,  0,  5,  9,  2,
      6, 11, 13,  8,  1,  4, 10,  7,  9,  5,  0, 15, 14,  2,  3, 12,
                                /* S[8]                                  */
     13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3, 14,  5,  0, 12,  7,
      1, 15, 13,  8, 10,  3,  7,  4, 12,  5,  6, 11,  0, 14,  9,  2,
      7, 11,  4,  1,  9, 12, 14,  2,  0,  6, 10, 13, 15,  3,  5,  8,
      2,  1, 14,  7,  4, 10,  8, 13, 15, 12,  9,  0,  3,  5,  6, 11  };

char P[32]                          /* 32-bit permutation function P       */
 ={  16,  7, 20, 21,
     29, 12, 28, 17,
      1, 15, 23, 26,
      5, 18, 31, 10,
      2,  8, 24, 14,
     32, 27,  3,  9,
     19, 13, 30,  6,
     22, 11,  4, 25  };
```

```
#include "des.h"      /* include Interface Specification header file        */
#include "des64.i"    /* include DES Substitution and Permutation tables    */
#include <stdio.h>
/*
 *             Define global data structures used byy DES routines
 */

 char subkey[16][8];  /* Storage for the 16 sub-keys used in the DES rounds */
        /* each 48-bit subkey is saved as eight 6-bit valvues, in each byte */
/*
 *    endes(b) - main DES encryption routine, this routine encrypts one 64-bit
 *               block b using the DES algorithm with the current key.
 *           The encryption operation involves permuting the input block
 *           using permutation IP, applying a DES round sixteen times
 *           (which ensures the output is a complex function of the input,
 *           and the key), and finally applying permutation FP (inverse IP)
 */
  endes(b)
  Long    b[2];
  {
         Long    work[2];                    /* 64-bit working store     */
         register int    i;
         perm64(work, b, IP);                /* Apply IP to input block  */
         for (i=0;i<=15;i++)                 /* Perform the 16 rounds    */
                 round(work,subkey[i]);      /* on the data              */
         swap(work);                         /* Unswap after final round */
         perm64(b, work, FP);                /* Perform Inverse IP       */
  }

/*
 *    dedes(b) - main DES decryption routine, this routine decrypts one 64-bit
 *                    block b using the DES algorithm with the current key
 *
 *           Decryption uses the same algorithm as encryption, except that
 *           the subkey are used in reverse order.
 */
  dedes(b)
  Long    b[2];
  {
         Long    work[2];
         register  int    i;
         perm64(work, b, IP);                /* Apply IP input block     */
         for (i=15; i>=0; i--)               /* Perform the 16 rounds    */
                 round(work, subkey[i]);
         swap(work);                         /* Unswap after final round */
         perm64(b, work, FP);                /* Perform Inverse IP       */
  }
/*
 *    desinit() - perform any data structure initialization needed
 */
```

```
 desinit()
 {                      /* No preinitialized tables used here          */
 }
/*
 *       round(d, k) - implements a single DES round on 64-bit block d,using
 *                     48-bit subkey k. Each round performs the following
 *                     calculation:
 *                                     L(i)= R(i-1)
 *                                     R(i)= L(i-1) XOR f(R(i-1),K(i))
 *
 *                     nb: L (left) component is in d[0], R (right) component
 *                     in d[1]
 */
 round(d, k)
 Long   d[2];
 char   *k;
 {
        Long    t;
        extern Long    f();
        t= d[0] ^ f(d[1], k);          /* Calc complex fn t= f(R(i-1), K(i)) */
                                       /* and XOR with L(i-1)                */
        d[0]= d[1];                    /* L(i)= R(i-1)                       */
        d[1]= t ;                      /* R(i)= L(i-1) XOR f(R(i-1), K(i))   */
 }
/*
 *      f(r, k) - is the complex non- linear DES function, whose output is a
 *                complex function of both input data and sub-key.
 *                The input data R(i-1) is expanded to 48-bits via expansion
 *                fn E, is XOR'd with the subkey K(i),substituted into the
 *                S-boxes, and finally permuted by P. ie the caculation is:
 *                    A = E(R(i-1)) XOR K(i)          ( a 48-bit value)
 *                    B = S(A)                        ( a 32-bit value)
 *                    f = P(B)                        ( a 32-bit value)
 *
 *       nb: the 48-bit values are stored as eight 6-bit values.
 *           In each byte, the bits are numbered [x x 1 2 3 4 5 6]
 *           Overall the bit numbering is:
 *           [x x 1 2 3 4 5 6] [x x 7 8 9 10 11 12]...[x x 43 44 45 46 47 48]
 *       nb: the 6-bit S-box input value [x x 1 2 3 4 5 6] is interpreted as:
 *           bits [1 6] select a row within each box,
 *           bits [2 3 4 5] then select a column within that row
 *           hence the input value is reordered to [x x 1 6 2 3 4 5] before
 *           indexing into the S-box tables.
 */
 Long
 f(r, k)
 Long    r;     /* Data value R(i-1) */
 char   *k;     /* Subkey     K(i)   */
 {
         Long   b = 0,   /* 32 bit S-box output block                      */
```

```
            out = 0; /* 32 bit output value                              */
        char   a[8];    /* store expanded input data as eight 6 bit value */
        register Long  s;  /* an S-box output                            */
        register Long  rc; /* an S-box row-col index                     */
        register int   j;
        expand(a, r);      /* expand input data R(i) to 48 bits using fn E*/
        for(j=0; j<8; j++){        /* Loopup S-boxes to get B = S(A)      */
                rc = a[j] ^ k[j];  /* A = E(R(i-1)) XOR K(i)              */
                rc = (rc & 0x20) | /* reorder S-box index so              */
                     ((rc << 4) & 0x10) | /* bits 1,6 form the row       */
                     ((rc >>1) & 0x0F);    /* bits 2-5 form the col       */
                s = S[j][rc];              /* S-box j output              */
                b = (b << 4) | s;      /* Concatenable S-box output to b  */
         }
         perm32(&out, &b, P);          /* Apply 32 bit permutation P to B */

#ifdef TRACE
           /* If Tracing, dump R(i-1), and f(R(i-1),K(i))                */
         fprintf(stderr," f(%081x, %02x %02x %02x %02x %02x %02x %02x %02x)
                 = %081x", r, k[0], k[1], k[2], k[3], k[4], k[5], %k[6], k[7], out);
#endif TRACE
           return(out);

 }
/*
 *  keyint(key) - installs key for use in subsequent encryptions and
 *                decryptions. A key must be installed before endes/dedes can
 *                be called.
 *                The key is passed as a 64 bit value, of which the 56 non
 *                parity bits are used. The parity bits are inDES bits
 *                8, 16, 24, 32, 40, 48, 56, 64, (nb: these do NOT correspond
 *                to the parity bits in ascii chars when packed in the usual
 *                way). The functon performs the key scheduling calculation,
 *                saving the resulting sixteen 48 bit subkeys for use in
 *                subsequent encryption/decryption calculations. These 48 bit
 *                values are saved as eight 6 bit values, as detailed
 *                previously.
 *                The key scheduling calculation involves permuting the input
 *                key by PC1 which selects 56 bits dividing the 56 bit value
 *                into two halves C, D sixteen times rotates each half left by
 *                1 or 2 places according to schedule in keyrot concatenates
 *                the two 28 bit values, and permutes with PC2 which selects
 *                48 bits to become the subkey
 *  nb: the two 28 bit halves are stored in two longwords, with bits numbered
 *      as [1 2 ... 28 x x x]...[29 30 ... 48 x x x] in longs n[0] n[1]
 *      (MSB to LSB) the bottom 4 bits in each longword are ingnored.
 *      This scheme is used in keyinit(), keysched(), and rot128()
 */

#define MASK28 0xFFFFFFF0L                  /* Mask DES key bits 1 to 28   */
```

```
  keyinit (key)
  Long     key[2];              /* Key to use, stored as an array of Longs*/
  {
          Long  cd[2];   /* Storage for the two 28 bit key halves C and D  */
          register  int   i;
          perm64(cd, key, PC1);      /* Permute key with PC1               */
          cd[0] &= MASK28;   /* form 28 bit value C by dropping 4 parity b.*/
          cd[1] &= MASK28;   /* form 28 bit value D by dropping 4 parity b.*/
          for(i=0; i<16; i++){                  /* Form sixteen subkeys  */
                  rot128(&cd[0], (int) keyrot[i]);/* Rotate by 1 or 2 bits */
                  rot128(&cd[1],(int) keyrot[i]); /* according to schedule */
                  keyperm(cd, i);            /* Apply PC2 to form subkey i*/
                              }
  }
/*
 *     keyperm(cd, i) - takes the two key halves in cd, permutes them
 *                      according to Pc2, and generates a 48 bit output value,
 *                      which is store as eight 6 bit values
 *
 *     nb: to set bits in the output word, as mask with a single 1 in it is
 *         used. On each step; the 1 is shifted into the next location
 */
#define KEYBIT1 0x20                      /* bit 1 of a 6 bit value       */
#include <stdio.h>
 keyperm(cd, i)
 Long    cd[2];                           /* The two 28 bit key halves     */
 int        i;                            /* subkey number i               */
 {
          register  int  j, k;
          register  char mask;            /* mask used to set bit in output */
          register  char *perm = PC2;     /* used to step through perm PC2  */
                                          /* Process half C                 */
          for(j=0; j<4; j++){             /* For each S-box input           */
                  subkey[i][j] = 0;       /* Clear output word              */
                  mask = KEYBIT1;         /* Reset mask to bit 1            */
                  for (k=0;k<6;k++){      /* For each S-box input bit       */
                           if (keybit(cd, (int)*perm++) ==1)/* if input  */
                                                          /* bit permut*/
                                   subkey[i][j]|= mask;/* to this loc is 1*/
                                                  /* set it          */
                            mask >>=1; /* shift mask to next bit         */
                                   }
                          }
                                     /* Process half D                 */
          for(j=4; j<8; j++){        /* For each S-box input           */
                  subkey[i][j] = 0;  /* Clear output word              */
                  mask = KEYBIT1;    /* Reset mask to bit 1            */
                  for(k=0; k<6; k++){ /* For each S-box input bit       */
                        if(keybit(cd, (int)*perm++) == 1)/*If input bit  */
                             subkey[i][j] |= mask;
```

```
                                        /* to this loc is 1, set it        */
                            mask >>=1;  /* Shift mask to next bit          */
                                    }
                                }
        }
/*
 *  keybit (n, pos) - return bit at position pos in n (1 <= pos <= 48)
 *                    when n is the two 24 bit halves used in the key
 *                    scheduling bits are numbered [1 2...27 28 x x x x]
 *                    [29 30... 48 x x x x] in Longs       n[0]
 *                             n[1]
 *                    nb: bottom 4 bits in each longword are ignored
 */
  int
  keybit(n, pos)
  Long      n[2];
  int       pos;
  {
            register int  b, d, o;
            d = (pos -1) / 28;          /* calculate which half bit is in  */
            o = 31 - ((pos - 1) % 28);  /* calculate offset within half    */
            b = ((n[d] >> o) & 01);     /* shift wanted bit to LSB, mask it*/
            return (b);                 /* and retund wanted wbit          */
  }

/*
 *   rot128(b, pos) - rotate block b (where b is a 28 bit key half) left by
 *                    pos bits, (0 <= pos <= 32)
 */
  rot128(b, pos)
  Long        *b;                       /* Block to rotate left          */
  int         pos;                      /* Number of bits to rotate left */
  {
            *b = ((*b << pos) | (*b >> (28-pos))) & MASK28;
  }
/*
 *   perm64(out, in, perm) is the general permutation of a 64 bit input
 *   block to a 64 bit output block, under the control of a permutation
 *   array perm. Each element of perm specifies which input bit is to
 *   be permuted to the output bit with the same index as the array
 *   element.
 *   nb: to set bits in the output word, a mask with a single 1 in it is
 *       used. On each step, the 1 is shifted into the next location
 */
#define DESBIT1  0x80000000L          /* DES bit 1, MSB of 32 bit word     */
  perm64(out, in, perm)
  Long   out[2];                      /* Output 64 bit block to be permuted */
  Long    in[2];                      /* Input 64 bit block after permutation*/
  char   perm[64];                    /* Permutation array                 */
  {
```

```
        Long   mask = DESBIT1;          /* mask used to set bit in output     */
        register int        i;
                                        /* Process left half out[0]           */
        out[0] = 0L;                    /* Clear output word                  */
        for(i=0; i<32; i++){            /* For each bit position              */
              if(bit(in, (int)*perm++) == 1) /* if the input bit permuted     */
                    out[0] |= mask;     /* to this loc is 1, set it           */
              mask >>= 1;               /* shift mask to next bit             */
                          }
        out[1] = 0L;                    /* Clear output word                  */
        mask = DESBIT1;
        for (i=0; i<32; i++){           /* For each bit position              */
                  if (bit(in, (int)*perm++) == 1)
                          out[1] |= mask;/*to this loc is 1, set  it          */
                  mask >>= 1;           /* Shift mask to next bit             */
                           }
  }
/*
 *  perm32(out, in, perm) is the general permutation of a 32 bit input block
 *         to a 32 bit output block, under the control of a permutation
 *         array perm. Each element of perm specifies which input bit is to
 *         be permuted to the output bit with the same index as the array
 *         element.
 *  nb: to set bits in the output word, as mask with a single 1 in it is used
 *      On each step, the 1 is shifted into the next location.
 */
  perm32(out, in, perm)
  Long     *out;                      /* output 32 bit block to be permuted */
  Long      *in;                      /* input 32 bit block after permutation*/
  char     perm[32];                  /* permutation array                  */
  {
        Long mask = DESBIT1;          /* mask used to set bit in output     */
        register int       i;
        for(i=0; i<32; i++){          /* for each bit position              */
             if (bit(in, (int)*perm++) ==1)/* if the input bit permuted     */
                  *out |= mask;       /* to this loc is 1, set it           */
             mask >>= 1;              /* shift mask to next bit             */
                          }
  }
/*
 *   expand(a, r) - expands the 32 bit value r into a 48 bit value a stored
 *                  as eight 6 bit value in a char array, with bits numbered
 *                  as specified previously.
 *                  Due to the regular nature of the expansion matrix E, this
 *                  function implements it directly for efficiency reasons.
 *                  It takes each 4 bit nybble of the input word, and
 *                  concatenates it with the adjacent bit on either side to
 *                  form a 6 bit value. (nb: bit 32 is assumed to be adjacent
 *                  to bit 1)
 */
```

```
#define MASK6  0x3f                    /* mask out all bar lower 6 bit       */
  expand(a, r)
  char     *a;                  /* output array to store the 48 bit val     */
  Long      r;                  /* 32 bit innput data block                 */
  {
       register Long  t;   /* temporary storage for values                  */
       t = (r<<5) | (r>>27);  a[0] = t & MASK6;/* bits 32   1   2   3   4   5*/
       t = r >> 23;   a[1] = t & MASK6;        /* bits  4   5   6   7   8   9*/
       t = r >> 19;   a[2] = t & MASK6;        /* bits  8   9  10  11  12  13*/
       t = r >> 15;   a[3] = t & MASK6;        /* bits 12  13  14  15  16  17*/
       t = r >> 11;   a[4] = t & MASK6;        /* bits 16  17  18  19  20  21*
```

```
/*
 *  des.c - DES file encryption/decryption program.
 *          This program reads its stdin, and encryts/decrypts using the DES
 *  Data Encryption Algorithm in either Cipher Block Chaining (CBC), or
 *  Electronic Codebook (ECB) Mode. In ECB mode, each block is treated
 *  separately, is encrypted and the ciphertext output. In CBC mode, the
 *  plaintext input is XOR'd with the ciphertext output from the previous
 *  block ( or an initialization vector on the fist block). Thus each block
 *  depends on all previous blocks. The DES encryption/decryption is performed
 *  by calling DES library routines. For a detailed descryption of the DES
 *  encryption/descryption and modes of use see:
 *
 *   Australian Standard AS2805.5-1985  The Data Encryption Algorithm
 *
 *  Usage: des -e|-d [-h] -k key [-b]
 *         Flags:
 *               -e        encrypt input (degault)
 *               -d        decrypt input
 *               -h        specify that key is supplied in hex
 *               -k key    specifies key as 8 ascii chars, or 16 hex digits
 *               -b        use ECB mode, otherwise CBC mode is used
 */
#include "des.h"    /* include DES Interface Specification header file     */
#include <stdio.h>
#include<strings.h>
/*
 * Define global data structures used by DES routines
 */
  Long  iv[2] = {0, 0};      /* Initialization Vector & previous block       */
  Long  key[2] = {0, 0};     /* DES key                                      */
  char  *usage = "des -e|-d [-h] -k key [-b]";/*program usage statement      */
  char  *Name = 0;           /* name of this program                         */
  int   ecb   = 0;           /* Use ECB mode flag (default false)            */
  int   encrypting = 1;      /* Encrypt input flag (default true)            */
  int   hexkey = 0;          /* Key suppied in hex (default false)           */
  main (argc, argv)
  int   argc;
  char  **argv;
  {
        int  errflag = 0;    /* Error detected in command line flags?        */
        int  c;              /* current flag found                           */
        char *keyinp = 0;    /* ascii key supplied                           */
        extern char    *optarg; /* current getopt argument pointer           */
        Name = argv[0];      /* save name of program for error messages      */
                             /* scan command line flags                      */
        while ((c = getopt(argc, argv, "edhk:b")) != EOF){
                switch(c){
                     case 'e':  encrypting = 1;     break;
                     case 'd':  encrypting = 0;     break;
                     case 'h':  hexkey = 1;         break;
```

```
                    case 'k':  keyinp = optarg;     break;
                    case 'b':  ecb = 1;             break;
                    default:   errflag++;           break;
                         }
                                                        }
        if (errflag || keyinp == 0) { /* error in flags, or no key given    */
                                      fprintf(stderr, "%s\n", usage);
                                      exit(1);
        }
        if (hexkey)          /* convert key from ascii to long               */
                gethex(keyinp, key);/* specified in hex                      */
        else
                getkey(keyinp, key);/*specified in ascii                     */
        desinit();           /* initialize des library routines              */
        keyinit(key);        /* specify key to use                           */
        if (encrypting)      /* Encrypt/decrypt input as specified           */
                 do_encrypt();
        else
                 do_decrypt();

  }
/*
 * do_encrypt() - the main encryption routine. This procedure reads 64 bits
 *                (8 byte) blocks of input, XOR's with chain vector if in
 *                Cbc modde encrypts the block, and writes it out, saving the
 *                output as the next chain vector. It swaps bytes around if
 *                necessary on a little-endian machine. The last byte of the
 *                last block contains the number of valid characters in that
 *                block (the remaining chars are garbage).
 */
  do_encrypt()
  {
        int  cnt = 0;        /* count of characters read in last read        */
        int  i,j;
        Long work[2];        /* 64 bits work value used by DES routines      */
        FILE *matriz;
        if ((matriz = fopen("mat","a")) == NULL){
                  printf(" No open of file ");
                  exit (1);
                                    }
        do{                  /* read next 8 byte block, if the last one      */
            if ((cnt = fread ((char *)work, 1, 8, stdin)) != 8)
                    ((char *)work)[7] = cnt;/* save cnt of valid bytes      */
#ifdef LITTLE_ENDIAN
                  swap((char *)work);/* swap bytes round if little_endian    */
#endif LITTLE_ENDIAN
                  if(!ecb){  /* if in CBC mode, chain previousciphertext     */
                      work[0] ^= iv[0];   /* work = work XOR iv              */
                      work[1] ^= iv[1];
                          }
```

```
                endes(work);/* encrypt data block                          */
                if (!ecb){  /* if in CBC mode, save ciphertext as next iv  */
                       iv[0] = work[0];
                       iv[1] = work[1];
                         }
#ifdef LITTLE_ENDIAN
                swap((char *)work, 1, 8, stdout); /* write ciphertext      */
#endif LITTLE_ENDIAN
                fwrite((char *)work, 1, 8, stdout);/*write ciphertext      */
          } while (cnt == 8); /* loop back until last block has been read   */

            for (i=0;i<16;i++){
                     fprintf (matriz,"\n");
                    for (j=0;j<8;j++)
                           fwrite(&subkey[i][j], 1, 1, matriz);
                 }
   }
/*
 *  do_encrypt() - the main decryption routine. This procedure reads 64 bit
 *  (8 byte) blocks of input, XOR's with chain vector if in CBC mode decrypts
 *  the block, and writes it out, saving the output as the next chain vector
 *  It swaps bytes around if necessary on a little_endian machine. The last
 *  byte of the last block contains the number of valid characters in that
 *  block. Hence must delay writting the plaintext out until have checked to
 *  see that it wasn't the last block.
 */
 do_decrypt()
 {
        FILE  *MATRICES;
        int  cnt = 0;          /* count of characters read in last read      */
        int i,j;
        Long work[2];          /* 64 bit work value used by DES routines     */
        Long prev[2];          /* previous 64 bit plaintext saved            */
        Long nextiv[2];        /* each 64 bit ciphertext block is next iv    */
        if (( MATRICES = fopen ("mat1", "a")) == NULL){
                       printf("No open file");
                       exit(1);
                  }
        cnt = fread((char *)work, 1, 8, stdin);/* read the first 8 bytes     */
        do{
#ifdef LITTLE_ENDIAN
                swap((char *)work);/* swap bytes round if little_endian      */
#endif LITTLE_ENDIAN
                if (!ecb){    /* if in CBC mode, save cipher as next iv      */
                          nextiv[0] = work[0];
                          nextiv[1] = work[1];
                         }
                dedes(work);  /* decrypt data block                          */
                if (!ecb){    /* if in CBC mode, chain previous iv to work   */
                      work[0] ^= iv[0]; /* work = work XOR iv                */
```

```
                    work[1] ^= iv[1];
                    iv[0] = nextiv[0];/* & move next iv to current        */
                    iv[1] = nextiv[1];
                       }
#ifdef LITTLE_ENDIAN
                swap((char *)work);/*swap bytes round if little_endian      */
#endif LITTLE_ENDIAN
                prev[0] = work[0]; /* save plaintext while do EOF check     */
                prev[1] = work[1]; /*read next 8 byte block, if the last one*/
                if((cnt = fread((char *)work, 1, 8, stdin)) != 8){
                     cnt = ((char *)prev)[7];/*extract no valid bytes   */
                      if(cnt<0 || cnt>7){ /* argh - bad file             */
                           fprintf(stderr,"%s:Corrupted ciphertext",Name);
                           exit(-1);
                      }else if (cnt>0)/* last block  */
                                   /* has valid chars*/
                             fwrite((char *)prev, 1,  cnt, stdout);
                            }else{ /* normal case in middle of file*/
                              fwrite((char *)prev, 1, 8, stdout);
                             }

          } while (cnt == 8);
             for (i=0; i<16; i++){
                       fprintf(MATRICES, "\n");
                       for (j=0; j<8; j++)
                             fwrite(&subkey[i][j], 1, 1, MATRICES);
                  }
      }

/*
 * gethex(kin, k) and getkey(kin, k) convert the string supplied for the key
 * into a 64 bit  block, assuming the string is hex chars, or standard 7 bit
 * ascii chars respectively. getkey will provide odd parity for each byte
 * ( to overcome differing systems default values). This is compatible with
 * the SUN des(1) command.
 */
 gethex(kin, k)
 char     *kin;       /* input string supplied for key                    */
 Long     k[2];       /* output 64 bit key value                          */
{
           if((strlen(kin) != 16) || /* check for valid length key          */
              (sscanf(kin, "%8x%8x", &k[0], &k[1]) != 2)){
                  fprintf(stderr, "%s: Bad key specification", Name);
                  exit(-1);
                }
         }
getkey(kin, k)
  char *kin;           /* input string supplied for key                    */
  Long k[2];           /* output 64 bit key value                          */
  {
```

```
        char *lk = (char *)k; /* char pointer to 64 bit value          */
        int  i, j, p;
        if(strlen(kin) < 8){   /* check for valid length key           */
             fprintf(stderr, "%s: Bad key specification", Name);
             exit(-1);
                           }

                           /* Calculate parity bit for each key byte  */
        for (i=0;i<8; i++){ /* for each key char                       */
                  lk[i] = kin[i]; /* copy it into 64 bit block         */
                  for (p=0, j=0; j<7; j++) /*calculate parity & set it */
                       if (lk[i] & (1 << j))/* count number of 1's     */
                            p++;
                  if ((p & 1) == 0)              /* if even           */
                       lk[i] |= 0x80;            /* set paraty bit    */
                  else                           /* otherwise         */
                       lk[i] &= 0x7f;            /* clear parity bit  */
                           }
#ifdef LITTLE_ENDIAN
        swap(k);
#endif LITTLE_ENDIAN
  }
#ifdef LITTLE_ENDIAN
        swap(b)
        Longn b[2];
        {
              register char *cb = (char *)b;
              register char c;
              c = cb[0]; cb[0] = cb[3]; cb[3] = c;
              c = cb[1]; cb[1] = cb[2]; cb[2] = c;
              c = cb[4]; cb[4] = cb[7]; cb[7] = c;
              c = cb[5]; cb[5] = cb[6]; cb[6] = c;
         }
#endif LITTLE_ENDIAN
```

Appendix B

Selected VHDL source code files

The following source code files are provided within this appendix, and comprise the behavioral sections of the code only. Structural descriptions were considered too long to include, due to the large number of internal signals which had to be declared. Also excluded is a large section of the *ciphertext* package body, which declares eight long arrays for use in the cipher module.

- ciphertext
- data_buffer_in
- des_tiempo
- mux_tiempo
- perm1_comb
- perm2_comb
- left_shift_comb
- perm64_new1
- perm64o_new1
- perm32_new1
- perm32o_new1
- pos_new1
- noc_new1
- xor32_new1
- xor48_new1
- des_mux_tiempo
- mux_tiempo1
- data_buffer_out

```
-- FILENAME : ciphertext.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Contains a number of common data types, and constant declarations
--            for use in the Data Encryption Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

PACKAGE ciphertext IS
    SUBTYPE vec160 IS std_ulogic_vector    (0 to 159);
    SUBTYPE vec64  IS std_ulogic_vector    (0 to 63);
    SUBTYPE vec56  IS std_ulogic_vector    (0 to 55);
    SUBTYPE vec48  IS std_ulogic_vector    (0 to 47);
    SUBTYPE vec32  IS std_ulogic_vector    (0 to 31);
    SUBTYPE vec6   IS std_ulogic_vector    (0 to 5);
    SUBTYPE vec4   IS std_ulogic_vector    (0 to 3);
    SUBTYPE vec2   IS std_ulogic_vector    (0 to 1);
    TYPE memory  IS array    (0 to 63) OF vec4;
    TYPE memory2 IS array    (0 to 15) OF vec48;
    TYPE subk56  IS array    (0 to 15) OF vec56;
    TYPE subk48  IS array    (0 to 15) OF vec48;
    TYPE powerarray IS array    (5 downto 0) OF integer;

    CONSTANT ROM1 : memory := ( ('1','1','1','0'),
                                ('0','1','0','0'),...
    .
    .
    .

    CONSTANT zerokey : vec48 := (others => '0');
    CONSTANT zeros56 : vec56 := (others => '0');

END ciphertext;
```

```
-- FILENAME : data_buffer_in.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Input Data Buffer. Accepts single bit input and places
--            into a 160-bit array for output.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY data_buffer_in IS
    PORT ( data_clock    :  IN  bit;
           master_clock  :  IN  bit;
           ri            :  IN  std_ulogic := 'U';
           rs            :  OUT vec160 );
END data_buffer_in;

ARCHITECTURE behavior OF data_buffer_in IS

    SIGNAL i : std_ulogic := 'U';
    SIGNAL mem_sig : vec160 := (others => 'U');

    BEGIN
      syncro : PROCESS
        BEGIN
          WAIT UNTIL master_clock'event AND master_clock = '1';
          rs <= mem_sig;
          i <= '0';
        END PROCESS;

      read : PROCESS
        VARIABLE j : integer := 0;
        VARIABLE  sync : std_ulogic := '0';
        BEGIN
          WAIT UNTIL data_clock'event AND data_clock = '0';
          IF i = '0' AND sync = '0' THEN
            j := 1;
            sync := '1';
            mem_sig(0) <= ri;
          ELSE
            mem_sig(j) <= ri;
            j := j + 1;
            IF j = 160 THEN
              sync := '0';
            END IF;
```

```
        END IF;
      END PROCESS;
END behavior;
```

```
-- FILENAME : des_tiempo.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Sorts 160-bit input into two 64-bit outputs, the
--            remaining 32-bits being discarded.


LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY des_tiempo IS
    PORT ( ri   : IN  vec160;
           rs1  : OUT vec64;
           rs2  : OUT vec64 );
END des_tiempo;

ARCHITECTURE arch_block OF des_tiempo IS

    BEGIN
      rs1 <= ri(0 to 7) & ri(20 to 27) & ri(40 to 47) & ri(60 to 67) &
             ri(80 to 87) & ri(100 to 107) & ri(120 to 127) & ri(140
             to 147);
      rs2 <= ri(10 to 17) & ri(30 to 37) & ri(50 to 57) & ri(70 to 77)
             & ri(90 to 97) & ri(110 to 117) & ri(130 to 137) & ri(150
             to 157);

END arch_block;
```

```
-- FILENAME : mux_tiempo.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : In response to control bit 1, determines
--            whether input should be read from the data buffer
--            inputs (2 off) or the microprocessor inputs (2 off),
--            and then, in response to control bit 2, allocates
--            each of the required two inputs to the output
--            port, one after the other.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY mux_tiempo IS
    PORT (ri1      : IN  vec64;
          ri2      : IN  vec64;
          mi1      : IN  vec64;
          mi2      : IN  vec64;
          control1 : IN  std_ulogic;
          control2 : IN  std_ulogic;
          rs       : OUT vec64);
END mux_tiempo;

ARCHITECTURE arch_block OF mux_tiempo IS

    SIGNAL control : vec2;

    BEGIN
      control <= control1 & control2;
      WITH control SELECT
        rs <= ri1  WHEN "00",
              ri2  WHEN "01",
              mi1  WHEN "10",
              mi2  WHEN "11",
              (others => '0') WHEN others;
END arch_block;
```

```
-- FILENAME : perm1_comb.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Combinational module for permutation PC-1 of the Key
--            Generation Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY perm1_comb IS PORT ( keyin : IN vec64 := (others => '0');
       permed : OUT vec56 );
END perm1_comb;

ARCHITECTURE behavior OF perm1_comb IS

    BEGIN
      permed(0) <= keyin(56);
      permed(1) <= keyin(48);
      permed(2) <= keyin(40);
      permed(3) <= keyin(32);
      permed(4) <= keyin(24);
      permed(5) <= keyin(16);
      permed(6) <= keyin(8);
      permed(7) <= keyin(0);
      permed(8) <= keyin(57);
      permed(9) <= keyin(49);
      permed(10) <= keyin(41);
      permed(11) <= keyin(33);
      permed(12) <= keyin(25);
      permed(13) <= keyin(17);
      permed(14) <= keyin(9);
      permed(15) <= keyin(1);
      permed(16) <= keyin(58);
      permed(17) <= keyin(50);
      permed(18) <= keyin(42);
      permed(19) <= keyin(34);
      permed(20) <= keyin(26);
      permed(21) <= keyin(18);
      permed(22) <= keyin(10);
      permed(23) <= keyin(2);
      permed(24) <= keyin(59);
      permed(25) <= keyin(51);
      permed(26) <= keyin(43);
      permed(27) <= keyin(35);
```

```
        permed(28) <= keyin(62);
        permed(29) <= keyin(54);
        permed(30) <= keyin(46);
        permed(31) <= keyin(38);
        permed(32) <= keyin(30);
        permed(33) <= keyin(22);
        permed(34) <= keyin(14);
        permed(35) <= keyin(6);
        permed(36) <= keyin(61);
        permed(37) <= keyin(53);
        permed(38) <= keyin(45);
        permed(39) <= keyin(37);
        permed(40) <= keyin(29);
        permed(41) <= keyin(21);
        permed(42) <= keyin(13);
        permed(43) <= keyin(5);
        permed(44) <= keyin(60);
        permed(45) <= keyin(52);
        permed(46) <= keyin(44);
        permed(47) <= keyin(36);
        permed(48) <= keyin(28);
        permed(49) <= keyin(20);
        permed(50) <= keyin(12);
        permed(51) <= keyin(4);
        permed(52) <= keyin(27);
        permed(53) <= keyin(19);
        permed(54) <= keyin(11);
        permed(55) <= keyin(3);
END behavior ;
```

```
-- FILENAME : perm2_comb.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Combinational module for permutation PC-2 of the Key
--            Generation Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY perm2_comb IS
        PORT ( inkey  : IN  vec56 := zeros56;
               outkey : OUT vec48  );
END perm2_comb;

ARCHITECTURE behavior OF perm2_comb IS

    BEGIN
      outkey(0) <= inkey(13);
      outkey(1) <= inkey(16);
      outkey(2) <= inkey(10);
      outkey(3) <= inkey(23);
      outkey(4) <= inkey(0);
      outkey(5) <= inkey(4);
      outkey(6) <= inkey(2);
      outkey(7) <= inkey(27);
      outkey(8) <= inkey(14);
      outkey(9) <= inkey(5);
      outkey(10) <= inkey(20);
      outkey(11) <= inkey(9);
      outkey(12) <= inkey(22);
      outkey(13) <= inkey(18);
      outkey(14) <= inkey(11);
      outkey(15) <= inkey(3);
      outkey(16) <= inkey(25);
      outkey(17) <= inkey(7);
      outkey(18) <= inkey(15);
      outkey(19) <= inkey(6);
      outkey(20) <= inkey(26);
      outkey(21) <= inkey(19);
      outkey(22) <= inkey(12);
      outkey(23) <= inkey(1);
      outkey(24) <= inkey(40);
      outkey(25) <= inkey(51);
      outkey(26) <= inkey(30);
```

```
      outkey(27) <= inkey(36);
      outkey(28) <= inkey(46);
      outkey(29) <= inkey(54);
      outkey(30) <= inkey(29);
      outkey(31) <= inkey(39);
      outkey(32) <= inkey(50);
      outkey(33) <= inkey(44);
      outkey(34) <= inkey(32);
      outkey(35) <= inkey(47);
      outkey(36) <= inkey(43);
      outkey(37) <= inkey(48);
      outkey(38) <= inkey(38);
      outkey(39) <= inkey(55);
      outkey(40) <= inkey(33);
      outkey(41) <= inkey(52);
      outkey(42) <= inkey(45);
      outkey(43) <= inkey(41);
      outkey(44) <= inkey(49);
      outkey(45) <= inkey(35);
      outkey(46) <= inkey(28);
      outkey(47) <= inkey(31);
END behavior;
```

```
-- FILENAME : left_shift_comb.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Combinational module for shifting
--            section of the Key Generation Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY left_shift_comb is
        GENERIC ( places : IN   integer := 1 );
        PORT    ( cd_in  : IN   vec56 := zeros56 ;
                  cd_out : OUT  vec56  );
END left_shift_comb ;

ARCHITECTURE behavior OF left_shift_comb IS

    BEGIN
      WITH places SELECT
        cd_out <= cd_in(1 to 27) & cd_in(0) &
                    cd_in(29 to 55) & cd_in(28) WHEN 1,
                  cd_in(2 to 27) & cd_in(0 to 1) &
                    cd_in(30 to 55) & cd_in(28 to 29) WHEN 2,
                  cd_in WHEN others;
END behavior;
```

```
-- FILENAME : perm64_new1.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : First permutation if cipher block.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY perm64_new1 IS
    PORT (x_in  : IN vec64;
          r0,  r1,  r2,  r3,  r4,  r5,  r6,  r7,
          r8,  r9,  r10, r11, r12, r13, r14, r15,
          r16, r17, r18, r19, r20, r21, r22, r23,
          r24, r25, r26, r27, r28, r29, r30, r31,
          r32, r33, r34, r35, r36, r37, r38, r39,
          r40, r41, r42, r43, r44, r45, r46, r47,
          r48, r49, r50, r51, r52, r53, r54, r55,
          r56, r57, r58, r59, r60, r61, r62, r63  : OUT std_ulogic  );
END perm64_new1;


ARCHITECTURE arch_block OF perm64_new1 IS

    BEGIN
      r0  <= x_in(57);
      r1  <= x_in(49);
      r2  <= x_in(41);
      r3  <= x_in(33);
      r4  <= x_in(25);
      r5  <= x_in(17);
      r6  <= x_in(9);
      r7  <= x_in(1);
      r8  <= x_in(59);
      r9  <= x_in(51);
      r10 <= x_in(43);
      r11 <= x_in(35);
      r12 <= x_in(27);
      r13 <= x_in(19);
      r14 <= x_in(11);
      r15 <= x_in(3);
      r16 <= x_in(61);
      r17 <= x_in(53);
      r18 <= x_in(45);
      r19 <= x_in(37);
```

```
      r20 <= x_in(29);
      r21 <= x_in(21);
      r22 <= x_in(13);
      r23 <= x_in(5);
      r24 <= x_in(63);
      r25 <= x_in(55);
      r26 <= x_in(47);
      r27 <= x_in(39);
      r28 <= x_in(31);
      r29 <= x_in(23);
      r30 <= x_in(15);
      r31 <= x_in(7);
      r32 <= x_in(56);
      r33 <= x_in(48);
      r34 <= x_in(40);
      r35 <= x_in(32);
      r36 <= x_in(24);
      r37 <= x_in(16);
      r38 <= x_in(8);
      r39 <= x_in(0);
      r40 <= x_in(58);
      r41 <= x_in(50);
      r42 <= x_in(42);
      r43 <= x_in(34);
      r44 <= x_in(26);
      r45 <= x_in(18);
      r46 <= x_in(10);
      r47 <= x_in(2);
      r48 <= x_in(60);
      r49 <= x_in(52);
      r50 <= x_in(44);
      r51 <= x_in(36);
      r52 <= x_in(28);
      r53 <= x_in(20);
      r54 <= x_in(12);
      r55 <= x_in(4);
      r56 <= x_in(62);
      r57 <= x_in(54);
      r58 <= x_in(46);
      r59 <= x_in(38);
      r60 <= x_in(30);
      r61 <= x_in(22);
      r62 <= x_in(14);
      r63 <= x_in(6);
END arch_block;
```

```
-- FILENAME : perm64o_new1.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Final permutation if cipher block.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY perm64o_new1 IS
    PORT (
          r32, r33, r34, r35, r36, r37, r38, r39,
          r40, r41, r42, r43, r44, r45, r46, r47,
          r48, r49, r50, r51, r52, r53, r54, r55,
          r56, r57, r58, r59, r60, r61, r62, r63,
          r0,  r1,  r2,  r3,  r4,  r5,  r6,  r7,
          r8,  r9,  r10, r11, r12, r13, r14, r15,
          r16, r17, r18, r19, r20, r21, r22, r23,
          r24, r25, r26, r27, r28, r29, r30, r31  : IN std_ulogic;
          y_out : out vec64);
END perm64o_new1;

ARCHITECTURE arch_block OF perm64o_new1 IS

    BEGIN
      y_out(0) <= r39;
      y_out(1) <= r7;
      y_out(2) <= r47;
      y_out(3) <= r15;
      y_out(4) <= r55;
      y_out(5) <= r23;
      y_out(6) <= r63;
      y_out(7) <= r31;
      y_out(8) <= r38;
      y_out(9) <= r6;
      y_out(10) <= r46;
      y_out(11) <= r14;
      y_out(12) <= r54;
      y_out(13) <= r22;
      y_out(14) <= r62;
      y_out(15) <= r30;
      y_out(16) <= r37;
      y_out(17) <= r5;
      y_out(18) <= r45;
      y_out(19) <= r13;
```

```
        y_out(20) <= r53;
        y_out(21) <= r21;
        y_out(22) <= r61;
        y_out(23) <= r29;
        y_out(24) <= r36;
        y_out(25) <= r4;
        y_out(26) <= r44;
        y_out(27) <= r12;
        y_out(28) <= r52;
        y_out(29) <= r20;
        y_out(30) <= r60;
        y_out(31) <= r28;
        y_out(32) <= r35;
        y_out(33) <= r3;
        y_out(34) <= r43;
        y_out(35) <= r11;
        y_out(36) <= r51;
        y_out(37) <= r19;
        y_out(38) <= r59;
        y_out(39) <= r27;
        y_out(40) <= r34;
        y_out(41) <= r2;
        y_out(42) <= r42;
        y_out(43) <= r10;
        y_out(44) <= r50;
        y_out(45) <= r18;
        y_out(46) <= r58;
        y_out(47) <= r26;
        y_out(48) <= r33;
        y_out(49) <= r1;
        y_out(50) <= r41;
        y_out(51) <= r9;
        y_out(52) <= r49;
        y_out(53) <= r17;
        y_out(54) <= r57;
        y_out(55) <= r25;
        y_out(56) <= r32;
        y_out(57) <= r0;
        y_out(58) <= r40;
        y_out(59) <= r8;
        y_out(60) <= r48;
        y_out(61) <= r16;
        y_out(62) <= r56;
        y_out(63) <= r24;
END arch_block;
```

```
-- FILENAME : perm32_new1.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : First 32-bit permutation in encryption
--            section of the Data Encryption Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY perm32_new1 is
    PORT (
          ri0,  ri1,  ri2,  ri3,  ri4,  ri5,  ri6,  ri7,
          ri8,  ri9,  ri10, ri11, ri12, ri13, ri14, ri15,
          ri16, ri17, ri18, ri19, ri20, ri21, ri22, ri23,
          ri24, ri25, ri26, ri27, ri28, ri29, ri30, ri31  : IN  std_ulogic;
          rs0,  rs1 , rs2,  rs3,  rs4,  rs5,  rs6,  rs7,
          rs8,  rs9,  rs10, rs11, rs12, rs13, rs14, rs15,
          rs16, rs17, rs18, rs19, rs20, rs21, rs22, rs23,
          rs24, rs25, rs26, rs27, rs28, rs29, rs30, rs31,
          rs32, rs33, rs34, rs35, rs36, rs37, rs38, rs39,
          rs40, rs41, rs42, rs43, rs44, rs45, rs46, rs47  : OUT  std_ulogic );
END perm32_new1;


ARCHITECTURE arch_block OF perm32_new1 IS

    BEGIN
      rs0  <= ri31;
      rs1  <= ri0;
      rs2  <= ri1;
      rs3  <= ri2;
      rs4  <= ri3;
      rs5  <= ri4;
      rs6  <= ri3;
      rs7  <= ri4;
      rs8  <= ri5;
      rs9  <= ri6;
      rs10 <= ri7;
      rs11 <= ri8;
      rs12 <= ri7;
      rs13 <= ri8;
      rs14 <= ri9;
      rs15 <= ri10;
      rs16 <= ri11;
```

```
      rs17 <= ri12;
      rs18 <= ri11;
      rs19 <= ri12;
      rs20 <= ri13;
      rs21 <= ri14;
      rs22 <= ri15;
      rs23 <= ri16;
      rs24 <= ri15;
      rs25 <= ri16;
      rs26 <= ri17;
      rs27 <= ri18;
      rs28 <= ri19;
      rs29 <= ri20;
      rs30 <= ri19;
      rs31 <= ri20;
      rs32 <= ri21;
      rs33 <= ri22;
      rs34 <= ri23;
      rs35 <= ri24;
      rs36 <= ri23;
      rs37 <= ri24;
      rs38 <= ri25;
      rs39 <= ri26;
      rs40 <= ri27;
      rs41 <= ri28;
      rs42 <= ri27;
      rs43 <= ri28;
      rs44 <= ri29;
      rs45 <= ri30;
      rs46 <= ri31;
      rs47 <= ri0;
END arch_block;
```

```
-- FILENAME : perm32o_new1.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Final 32-bit permutation in encryption
--            section of the Data Encryption Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY perm32o_new1 IS
    PORT (
         ri0,  ri1,  ri2,  ri3,  ri4,  ri5,  ri6,  ri7,
         ri8,  ri9,  ri10, ri11, ri12, ri13, ri14, ri15,
         ri16, ri17, ri18, ri19, ri20, ri21, ri22, ri23,
         ri24, ri25, ri26, ri27, ri28, ri29, ri30, ri31  : IN std_ulogic;
         rs0,  rs1,  rs2,  rs3,  rs4,  rs5,  rs6,  rs7,
         rs8,  rs9,  rs10, rs11, rs12, rs13, rs14, rs15,
         rs16, rs17, rs18, rs19, rs20, rs21, rs22, rs23,
         rs24, rs25, rs26, rs27, rs28, rs29, rs30, rs31  : OUT std_ulogic);
END perm32o_new1;


ARCHITECTURE arch_block OF perm32o_new1 IS

    BEGIN
      rs0   <= ri15;
      rs1   <= ri6;
      rs2   <= ri19;
      rs3   <= ri20;
      rs4   <= ri28;
      rs5   <= ri11;
      rs6   <= ri27;
      rs7   <= ri16;
      rs8   <= ri0;
      rs9   <= ri14;
      rs10  <= ri22;
      rs11  <= ri25;
      rs12  <= ri4;
      rs13  <= ri17;
      rs14  <= ri30;
      rs15  <= ri9;
      rs16  <= ri1;
      rs17  <= ri7;
      rs18  <= ri23;
```

```
-- FILENAME : pos_new1.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Computation block in encryption
--            section of the Data Encryption Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_unsigned.all
USE WORK.ciphertext.all;

ENTITY pos_new1 is
    PORT (
          ri0,  ri1,  ri2,  ri3,  ri4,  ri5,  ri6,  ri7,
          ri8,  ri9,  ri10, ri11, ri12, ri13, ri14, ri15,
          ri16, ri17, ri18, ri19, ri20, ri21, ri22, ri23,
          ri24, ri25, ri26, ri27, ri28, ri29, ri30, ri31,
          ri32, ri33, ri34, ri35, ri36, ri37, ri38, ri39,
          ri40, ri41, ri42, ri43, ri44, ri45, ri46, ri47 :IN std_ulogic;
          rs0,  rs1,  rs2,  rs3,  rs4,  rs5,  rs6,  rs7,
          rs8,  rs9,  rs10, rs11, rs12, rs13, rs14, rs15,
          rs16, rs17, rs18, rs19, rs20, rs21, rs22, rs23,
          rs24, rs25, rs26, rs27, rs28, rs29, rs30, rs31 : OUT  std_ulogic);
END pos_new1;


ARCHITECTURE arch_block OF pos_new1 IS

    SIGNAL s1,s2,s3,s4,s5,s6,s7,s8: vec6;
    SIGNAL a1,a2,a3,a4,a5,a6,a7,a8: vec4;
    SIGNAL add1,add2,add3,add4,add5,add6,add7,add8: integer:= 0;

    BEGIN
      s1(0) <= ri0;
      s1(2) <= ri1;
      s1(3) <= ri2;
      s1(4) <= ri3;
      s1(5) <= ri4;
      s1(1) <= ri5;
      s2(0) <= ri6;
      s2(2) <= ri7;
      s2(3) <= ri8;
      s2(4) <= ri9;
      s2(5) <= ri10;
      s2(1) <= ri11;
```

```
s3(0) <= ri12;
s3(2) <= ri13;
s3(3) <= ri14;
s3(4) <= ri15;
s3(5) <= ri16;
s3(1) <= ri17;
s4(0) <= ri18;
s4(2) <= ri19;
s4(3) <= ri20;
s4(4) <= ri21;
s4(5) <= ri22;
s4(1) <= ri23;
s5(0) <= ri24;
s5(2) <= ri25;
s5(3) <= ri26;
s5(4) <= ri27;
s5(5) <= ri28;
s5(1) <= ri29;
s6(0) <= ri30;
s6(2) <= ri31;
s6(3) <= ri32;
s6(4) <= ri33;
s6(5) <= ri34;
s6(1) <= ri35;
s7(0) <= ri36;
s7(2) <= ri37;
s7(3) <= ri38;
s7(4) <= ri39;
s7(5) <= ri40;
s7(1) <= ri41;
s8(0) <= ri42;
s8(2) <= ri43;
s8(3) <= ri44;
s8(4) <= ri45;
s8(5) <= ri46;
s8(1) <= ri47;
add1 <= CONV_INTEGER( To_StdLogicVector (s1));
a1 <= ROM1(add1);
add2 <= CONV_INTEGER( To_StdLogicVector (s2));
a2 <= ROM2(add2);
add3 <= CONV_INTEGER( To_StdLogicVector (s3));
a3 <= ROM3(add3);
add4 <= CONV_INTEGER( To_StdLogicVector (s4));
a4 <= ROM4(add4);
add5 <= CONV_INTEGER( To_StdLogicVector (s5));
```

```
      a5 <= ROM5(add5);
      add6 <= CONV_INTEGER( To_StdLogicVector (s6));
      a6 <= ROM6(add6);
      add7 <= CONV_INTEGER( To_StdLogicVector (s7));
      a7 <= ROM7(add7);
      add8 <= CONV_INTEGER( To_StdLogicVector (s8));
      a8 <= ROM8(add8);
      rs0  <= a1(0);
      rs1  <= a1(1);
      rs2  <= a1(2);
      rs3  <= a1(3);
      rs4  <= a2(0);
      rs5  <= a2(1);
      rs6  <= a2(2);
      rs7  <= a2(3);
      rs8  <= a3(0);
      rs9  <= a3(1);
      rs10 <= a3(2);
      rs11 <= a3(3);
      rs12 <= a4(0);
      rs13 <= a4(1);
      rs14 <= a4(2);
      rs15 <= a4(3);
      rs16 <= a5(0);
      rs17 <= a5(1);
      rs18 <= a5(2);
      rs19 <= a5(3);
      rs20 <= a6(0);
      rs21 <= a6(1);
      rs22 <= a6(2);
      rs23 <= a6(3);
      rs24 <= a7(0);
      rs25 <= a7(1);
      rs26 <= a7(2);
      rs27 <= a7(3);
      rs28 <= a8(0);
      rs29 <= a8(1);
      rs30 <= a8(2);
      rs31 <= a8(3);
END arch_block;
```

```
-- FILENAME : noc_new1.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Module to swap left and right halves in encryption
--            section of the Data Encryption Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY noc_new1 IS
    PORT (
          ro0,  ro1,  ro2,  ro3,  ro4,  ro5,  ro6,  ro7,
          ro8,  ro9,  ro10, ro11, ro12, ro13, ro14, ro15,
          ro16, ro17, ro18, ro19, ro20, ro21, ro22, ro23,
          ro24, ro25, ro26, ro27, ro28, ro29, ro30, ro31 : IN std_ulogic;
          ls0,  ls1,  ls2,  ls3,  ls4,  ls5,  ls6,  ls7,
          ls8,  ls9,  ls10, ls11, ls12, ls13, ls14, ls15,
          ls16, ls17, ls18, ls19, ls20, ls21, ls22, ls23,
          ls24, ls25, ls26, ls27, ls28, ls29, ls30, ls31  : OUT std_ulogic);
END noc_new1;


ARCHITECTURE arch_block OF noc_new1 IS

    BEGIN
      ls0  <= ro0;
      ls1  <= ro1;
      ls2  <= ro2;
      ls3  <= ro3;
      ls4  <= ro4;
      ls5  <= ro5;
      ls6  <= ro6;
      ls7  <= ro7;
      ls8  <= ro8;
      ls9  <= ro9;
      ls10 <= ro10;
      ls11 <= ro11;
      ls12 <= ro12;
      ls13 <= ro13;
      ls14 <= ro14;
      ls15 <= ro15;
      ls16 <= ro16;
      ls17 <= ro17;
      ls18 <= ro18;
```

```
      ls19 <= ro19;
      ls20 <= ro20;
      ls21 <= ro21;
      ls22 <= ro22;
      ls23 <= ro23;
      ls24 <= ro24;
      ls25 <= ro25;
      ls26 <= ro26;
      ls27 <= ro27;
      ls28 <= ro28;
      ls29 <= ro29;
      ls30 <= ro30;
      ls31 <= ro31;
END arch_block;
```

```
-- FILENAME : xor32_new1.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : 32-bit exclusive-OR block in encryption
--            section of the Data Encryption Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY xor32_new1 is
    PORT (
          ri0,  ri1,  ri2,  ri3,  ri4,  ri5,  ri6,  ri7,
          ri8,  ri9,  ri10, ri11, ri12, ri13, ri14, ri15,
          ri16, ri17, ri18, ri19, ri20, ri21, ri22, ri23,
          ri24, ri25, ri26, ri27, ri28, ri29, ri30, ri31 : IN std_ulogic;
          li0,  li1,  li2,  li3,  li4,  li5,  li6,  li7,
          li8,  li9,  li10, li11, li12, li13, li14, li15,
          li16, li17, li18, li19, li20, li21, li22, li23,
          li24, li25, li26, li27, li28, li29, li30, li31 : IN std_ulogic;
          rs0,  rs1,  rs2,  rs3,  rs4,  rs5,  rs6,  rs7,
          rs8,  rs9,  rs10, rs11, rs12, rs13, rs14, rs15,
          rs16, rs17, rs18, rs19, rs20, rs21, rs22, rs23,
          rs24, rs25, rs26, rs27, rs28, rs29, rs30, rs31 : OUT std_ulogic);
END xor32_new1;


ARCHITECTURE arch_block OF xor32_new1 IS

    BEGIN
      rs0  <= ri0  xor li0;
      rs1  <= ri1  xor li1;
      rs2  <= ri2  xor li2;
      rs3  <= ri3  xor li3;
      rs4  <= ri4  xor li4;
      rs5  <= ri5  xor li5;
      rs6  <= ri6  xor li6;
      rs7  <= ri7  xor li7;
      rs8  <= ri8  xor li8;
      rs9  <= ri9  xor li9;
      rs10 <= ri10 xor li10;
      rs11 <= ri11 xor li11;
      rs12 <= ri12 xor li12;
      rs13 <= ri13 xor li13;
      rs14 <= ri14 xor li14;
```

```
      rs15 <= ri15 xor li15;
      rs16 <= ri16 xor li16;
      rs17 <= ri17 xor li17;
      rs18 <= ri18 xor li18;
      rs19 <= ri19 xor li19;
      rs20 <= ri20 xor li20;
      rs21 <= ri21 xor li21;
      rs22 <= ri22 xor li22;
      rs23 <= ri23 xor li23;
      rs24 <= ri24 xor li24;
      rs25 <= ri25 xor li25;
      rs26 <= ri26 xor li26;
      rs27 <= ri27 xor li27;
      rs28 <= ri28 xor li28;
      rs29 <= ri29 xor li29;
      rs30 <= ri30 xor li30;
      rs31 <= ri31 xor li31;
END arch_block;
```

```
-- FILENAME : xor48_new1.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : 48-bit exclusive-OR block in encryption
--            section of the Data Encryption Algorithm.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY xor48_new1 IS
    PORT (
          ri0,  ri1,  ri2,  ri3,  ri4,  ri5,  ri6,  ri7,
          ri8,  ri9,  ri10, ri11, ri12, ri13, ri14, ri15,
          ri16, ri17, ri18, ri19, ri20, ri21, ri22, ri23,
          ri24, ri25, ri26, ri27, ri28, ri29, ri30, ri31,
          ri32, ri33, ri34, ri35, ri36, ri37, ri38, ri39,
          ri40, ri41, ri42, ri43, ri44, ri45, ri46, ri47 : IN std_ulogic;
          si1   : IN vec48;
          si2   : IN vec48;
          enci  : IN bit;
          rs0,  rs1,  rs2,  rs3,  rs4,  rs5,  rs6,  rs7,
          rs8,  rs9,  rs10, rs11, rs12, rs13, rs14, rs15,
          rs16, rs17, rs18, rs19, rs20, rs21, rs22, rs23,
          rs24, rs25, rs26, rs27, rs28, rs29, rs30, rs31,
          rs32, rs33, rs34, rs35, rs36, rs37, rs38, rs39,
          rs40, rs41, rs42, rs43, rs44, rs45, rs46, rs47 : OUT std_ulogic);
END xor48_new1;


ARCHITECTURE arch_block OF xor48_new1 IS

    SIGNAL b48: vec48;

    BEGIN
      WITH enci SELECT
        b48 <= si1 WHEN '1',
               si2 WHEN '0';
      rs0  <= ri0  xor b48(0);
      rs1  <= ri1  xor b48(1);
      rs2  <= ri2  xor b48(2);
      rs3  <= ri3  xor b48(3);
      rs4  <= ri4  xor b48(4);
      rs5  <= ri5  xor b48(5);
      rs6  <= ri6  xor b48(6);
```

```
      rs7  <= ri7  xor b48(7);
      rs8  <= ri8  xor b48(8);
      rs9  <= ri9  xor b48(9);
      rs10 <= ri10 xor b48(10);
      rs11 <= ri11 xor b48(11);
      rs12 <= ri12 xor b48(12);
      rs13 <= ri13 xor b48(13);
      rs14 <= ri14 xor b48(14);
      rs15 <= ri15 xor b48(15);
      rs16 <= ri16 xor b48(16);
      rs17 <= ri17 xor b48(17);
      rs18 <= ri18 xor b48(18);
      rs19 <= ri19 xor b48(19);
      rs20 <= ri20 xor b48(20);
      rs21 <= ri21 xor b48(21);
      rs22 <= ri22 xor b48(22);
      rs23 <= ri23 xor b48(23);
      rs24 <= ri24 xor b48(24);
      rs25 <= ri25 xor b48(25);
      rs26 <= ri26 xor b48(26);
      rs27 <= ri27 xor b48(27);
      rs28 <= ri28 xor b48(28);
      rs29 <= ri29 xor b48(29);
      rs30 <= ri30 xor b48(30);
      rs31 <= ri31 xor b48(31);
      rs32 <= ri32 xor b48(32);
      rs33 <= ri33 xor b48(33);
      rs34 <= ri34 xor b48(34);
      rs35 <= ri35 xor b48(35);
      rs36 <= ri36 xor b48(36);
      rs37 <= ri37 xor b48(37);
      rs38 <= ri38 xor b48(38);
      rs39 <= ri39 xor b48(39);
      rs40 <= ri40 xor b48(40);
      rs41 <= ri41 xor b48(41);
      rs42 <= ri42 xor b48(42);
      rs43 <= ri43 xor b48(43);
      rs44 <= ri44 xor b48(44);
      rs45 <= ri45 xor b48(45);
      rs46 <= ri46 xor b48(46);
      rs47 <= ri47 xor b48(47);
END arch_block;
```

```
-- FILENAME : des_mux_tiempo.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : In response to control bit 1, determines whether
--            output should be sent to the data buffer outputs
--            (2 off) or the microprocessor outputs (2 off).
--            In response to control bit 2, allocates the input to
--            each of the required two output ports in turn.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY des_mux_tiempo IS
    PORT (ri       : IN  vec64;
          rs1      : OUT vec64;
          rs2      : OUT vec64;
          ms1      : OUT vec64;
          ms2      : OUT vec64;
          control1 : IN  std_ulogic;
          control2 : IN  std_ulogic );
END des_mux_tiempo;

ARCHITECTURE arch_block OF des_mux_tiempo IS

    BEGIN

      PROCESS (control1,control2,ri)
        VARIABLE control: vec2;
        BEGIN
          control := control1 & control2;
          CASE control IS
            WHEN "00" =>
              rs1 <= ri;
            WHEN "01" =>
              rs2 <= ri;
            WHEN "10" =>
              ms1 <= ri;
            WHEN "11" =>
              ms2 <= ri;
            WHEN others =>
              NULL;
          END CASE;
      END PROCESS;
END arch_block;
```

```
-- FILENAME : mux_tiempo1.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Creates 160-bit output from two 64-bit inputs, the
--            extra 32-bits being replaced by '1'.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY mux_tiempo1 IS
    PORT (ri1   : IN  vec64;
          ri2   : IN  vec64;
          rs    : OUT vec160);
END mux_tiempo1;

ARCHITECTURE arch_block OF mux_tiempo1 IS

    SIGNAL a1 : vec2 := ('1','1');

    BEGIN
      rs <= ri1(0 to 7) & a1 & ri2(0 to 7) & a1 & ri1(8 to 15) & a1 &
            ri2(8 to 15) & a1 & ri1(16 to 23) & a1 & ri2(16 to 23) &
            a1 & ri1(24 to 31) & a1 & ri2(24 to 31) & a1 & ri1(32 to
            39) & a1 & ri2(32 to 39) & a1 & ri1(40 to 47) & a1 &
            ri2(40 to 47) & a1 & ri1(48 to 55) & a1 & ri2(48 to 55) &
            a1 & ri1(56 to 63) & a1 & ri2(56 to 63) & a1;
END arch_block;
```

```
-- FILENAME : data_buffer_out.vhd
-- AUTHOR   : Philip Mills & Alfonso Naranjo
-- DATE     : 31st August 1995
-- FUNCTION : Takes 160-bit input and outputs one bit at a
--            time, in reponse to data_clock.

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.ciphertext.all;

ENTITY data_buffer_out IS
    PORT ( data_clock :  IN  bit;
           control    :  IN  std_ulogic;
           ri         :  IN  vec160;
           rs         :  OUT std_ulogic );
END data_buffer_out;

ARCHITECTURE behavior OF data_buffer_out IS

    SIGNAL i : std_ulogic := 'U';
    SIGNAL mem_sig : vec160 := (others => 'U');

    BEGIN
      syncro : PROCESS
        BEGIN
          WAIT UNTIL control = '1';
          mem_sig <= ri;
          i <= '0';
        END process;

      write : PROCESS
        VARIABLE j : integer := 0;
        VARIABLE sync : std_ulogic := '0';
        BEGIN
          WAIT UNTIL data_clock'event AND data_clock = '0';
          IF i = '0' AND sync = '0' THEN
            j := 1;
            sync := '1';
            rs <= mem_sig(0);
          ELSE
            rs <= mem_sig(j);
            j := j + 1;
            IF j = 160 THEN
              sync := '0';
            END IF;
```

```
        END IF;
      END PROCESS;
END behavior;
```

An Introduction to the integrated use of Cadence design tools for VHDL design

Contents

Chapter 1

Introduction

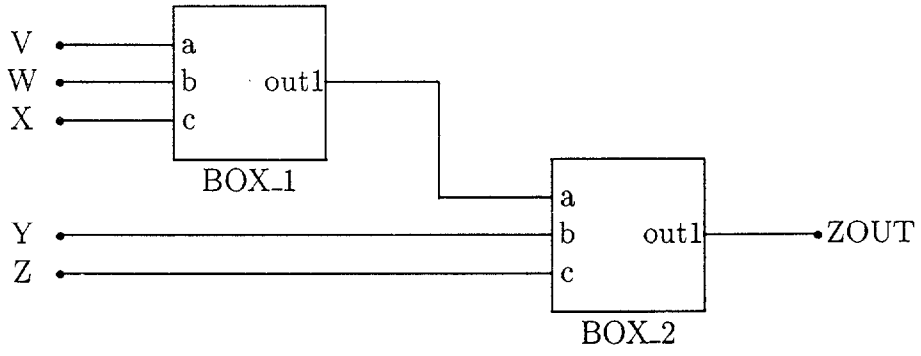
1.1 Introduction

The Cadence Design Framework provides an integrated environment for using the automated tools required at all stages of the electronic design process. The Schematic Editor and Language Sensitive Editor (LSE) provide a method of data input in the form of schematic and VHDL code. The Leapfrog Simulation tool provides a very powerful environment for simulating the design, and observing the performance of the system over time. The Synergy synthesis tool provides a method of optimizing the design, and carrying out the snthesis step to provide a logic gate layout of the design.

The Cadence system comes complete with comprehensive on-line help information, on all the various products, and also with a number of tutorials aimed at introducing the novice user to the individual tools. The purpose of this tutorial is to show, using a very simple example, how the different tools may be used within the integrated environment of the VHDL Tool Box, to carry out all the steps in the design process, as described below :

1. Create a top level schematic block diagram.
2. Create VHDL code for the modules.
3. Create a test bench.
4. Define the hierarchy.
5. Simulate and debug the design.
6. Synthesize the design.

The example which will be used in the tutorial is a very simple circuit containing two blocks, as shown below:



The logic functions for the behavior of the two units are as follows :

- For BOX_1, **out1 = (a AND b) OR (a AND c)**.
- For BOX_2, **out1 = (a OR b) AND (b OR c) AND (a OR c)**.
- For BOX_1, delay time is 12 ns.
- For BOX_2, delay time is 15 ns.

Chapter 2

Create and Check top level Schematic

2.1 Create and Check Top Level Schematic

2.1.1 Starting up the Cadence Design Framework

Start up the Cadence Design Framework environment from a command tool window by typing :

```
icfb &
```

The Cadence Command Interpreter Window (CIW) appears.

2.1.2 Setting up the System Environment

When the Cadence system starts up, you have to tell the system where to find various pieces of information required to run the design, simulation and synthesis software. This includes telling the software where the various cell libraries reside. These libraries are located in various subdirectories of the *software installation path*, so the first task is to determine the location of this directory, and then to add the various sub-directories to your search path.

Identifying the Software Installation Path

Enter the following command in the CIW command line.

```
println(cdsGetInstPath())
```

The software installation path appears in the CIW message window.

Setting your Library Search Path

You now need to set up the search path to your design libraries and also to existing Cadence libraries, so that the system knows where to find them.

Select **Design Manager - Set Search Path** in the CIW.

The Set Library Search Path window appears.

In the *SearchPath* field, type the following directory names, separated by spaces. <install-dir> is the software installation path obtained in the previous section.

<install-dir>/tools/dfII/etc/cdslib

<install-dir>/tools/dfII/etc/cdslib/sheets

<install-dir>/tools/dfII/etc/amDFlow

<install-dir>/tools/dfII/etc/pcblibs

<install-dir>/tools/dfII/etc/cdslib/artist

<install-dir>/tools/dfII/samples/cdslib

Click **OK**.

Setting VHDL Data Types for Pins

In order that the pins on your top level schematic diagram can be assigned VHDL properties, the VHDL configuration file must be loaded.

Enter the following command in the CIW :

```
load(prependInstallPath(''samples/local/vhdlConfig.il''))
```

The letter *t* should appear in the CIW window, indicating that the command was successful.

2.1.3 Creating a VHDL Design Library

You must now create a library in which to store all the files comprising your VHDL design.

Select **Open - Library** in the CIW.

In the *Library Name* field, type the name of the new design library, in this case **bbexample**.

In the *Library Path* field, type the path to your home directory.

Click **OK**.

A dialog box appears, informing you that the library does not exist, and asking if you wish to create it.

Click **OK** in the dialog box.

The Create Library form appears, with the name and path of the new library.

Click **OK**.

2.1.4 Opening the VHDL Tool Box

The complete design process, from drawing up the top level schematic, through simulation and synthesis, is controlled from the VHDL Tool Box.

Select **Open - VHDL Toolbox** in the CIW.

The VHDL Tool Box window appears.

2.1.5 Opening a new Cellview

The first stage in the VHDL design process is to create a top level schematic view of the design. This is done using the Cadence Schematic Editor.

In the *Working Cellview Library Name* field of the VHDL Tool Box, type the name of the design library where you want the design to reside, in this case **bbexample**.

In the *Working Cellview Cell Name* field, type the name of the cell view, in this case **black_box**.

In the *Working Cellview View Name* field, type the name of the view to be created, in this case **schematic**.

Click **Edit**.

A dialog box appears informing you that the new cellview does not exist, and asking if you wish to create it.

Click **OK** in the dialog box.

The Schematic Editor window opens with an empty window for you to create the schematic.

2.1.6 Creating a Block Diagram

Adding Blocks

The first items to be added to the schematic are the two rectangles representing the objects *BOX_1* and BOX_2.

Select **Add - Block**.

The Add Block form appears.

In the *Cell Name* field, type **BOX_1** and in the *Instance Name* field type **B1**.

In the Schematic Editor window, click with the mouse on the top left corner of the position where you want BOX_1 to appear, and then click on the bottom right corner.

In the Add Block form, in the *Cell Name* field, type **BOX_2** and in the *Instance Name* field type **B2**.

In the Schematic Editor window, click with the mouse on the top left corner of the position where you want BOX_2 to appear, and then click on the bottom right corner.

In the Add Block form, click **Hide** to remove the form from the screen.

Adding Pins

In all, the Black_Box design contains six external pins, five input (V, W, X, Y and Z) and one output (ZOUT).

Select **Add - Pin**. The Add Pin from appears.

In the *Pin Names* field, type the pin names **V, W, X, Y, Z** and **ZOUT**, separating each with a space.

Make sure that the *Direction* cyclic box reads **input**.

In the *VHDL Data Type* field, type **BIT**.

In the *VHDL Initial Value* field, type **'0'**.

In the Schematic Editor window, place the five input pins in appropriate positions by clicking with the left mouse button.

On the Add Pin form, change the *Direction* cyclic box to read **output**.

In the *VHDL Data Type* field, type **BIT**.

In the *VHDL Initial Value* field, type **'0'**.

In the Schematic Editor window, place the output pin in an appropriate position by clicking with the left mouse button.

On the Add Pin form, click **Cancel** to cancel the Add Pinn command.

Connecting the Blocks and Pins

The next stage in the schematic diagram is to make the connections between the external pins and the objects, and also between the objects themselves.

Select **Add - Wire Narrow**.

Add wires from the input and ouput pins to the boxes, and the single wire connecting the boxes by clicking first on the wire start point, and then on the end point.

When all the wires have been added, press the <ESCAPE> key to cancel the Add - Wire command.

Naming the internal signals

There is only one internal signal to name in the Black Box example.

Select **Add - Wire name**. The Add Wire Name form appears.

In the *Names* field, type **S1**.

In the Schematic Editor, click the mouse on the position where you want the label to apper, and then click on the wire to which the name applies.

In the Add Wire Name form, click **Hide** to remove the form from the screen.

2.1.7 Renaming the Pins

When the wires were added to the schematic, the pin names were generated automatically. These names will now be changed.

Select **Edit - Properties - Objects**, and then click on the rectangle representing BOX_1. The Edit Object Properties form appears.

Click on **system** to enlarge the form, and change the existing pin names to read **a, b, c, out1** in that order.

Click on **Apply** to confirm the changes. The Edit Object Properties window will remain on the screen.

In the Schematic Editor, click on the rectangle representing BOX_2.

In the Edit Object Properties form, again change the existing pin names to read **a, b, c, out1** in that order.

Click **Apply** to confirm the changes, and then click **Hide** to remove the form from view.

2.1.8 SPecifying Pin Ordering and Data Types

Specifying Pin Ordering for Top Level Entity

The logical ordering of the external pins will now be changed, so that they appear in alphabetical order, with input pins preceding output pins.

Select **Edit - Properties - Pin Order**. The Edit Pin Order form appears.

Highlight **ZOUT** and then click **Move to Bottom**.

Click **OK**.

Pin Ordering and Data Types - BOX_1

The pin ordering has now been set for the top level schematic. It must now also be done for the lower level entities.

Select **Design - Hierarchy - Descend Edit**, and then click on the rectangle representing BOX_1. The Symbol Editor appears, containg the symbol diagram for BOX_1.

Select **Edit - Properties - Pin Order**. The Edit Pin Order form appears.

Highlight **out1** and then click **Move to Bottom**.

Click **OK**.

Select **Edit - Properties - Objects**, and then click on any one of the three input pins. The Edit Object Properties form appears.

Set the **Apply To** cyclic buttons to read **all, symbol pins, same direction**. In the *VHDL Data Type* field, type **BIT**.

Click **Apply** to confirm the change. The Edit Object Properties form stays on the screen.

Click on the single output pin.

Set the **Apply To** cyclic buttons to read **all, symbol pins, same direction**. In the *VHDL Data Type* field type **BIT**.

Click **Apply** to confirm the change, and then click **Hide** to remove the Edit Object Properties form from the screen.

Select **Design - Hierarchy - Return**. A dialogue box appears warning that the symbol has been changed, and asking if you wish to save the changes.

Click **Yes**.

Pin Ordering and Data Types - BOX_2

The process for changing the pin ordering and data types for BOX_2 is identical to that for BOX_1.

In the Schematic Editor, select **Design - Hierarchy - Descend Edit**, and then click on the rectangle representing BOX_2. The Symbol Editor appears, containg the symbol diagram for BOX_2.

Select **Edit - Properties - Pin Order**. The Edit Pin Order form appears.

Highlight **out1** and then click **Move to Bottom**.

Click **OK**.

Select **Edit - Properties - Objects**, and then click on any one of the three input pins. The Edit Object Properties form appears.

Set the **Apply To** cyclic buttons to read **all, symbol pins, same direction**. In the *VHDL Data Type* field type **BIT**.

Click **Apply** to confirm the change. The Edit Object Properties form stays on the screen.

Click on the single output pin.

Set the **Apply To** cyclic buttons to read **all, symbol pins, same direction**. In the *VHDL Data Type* field type **BIT**.

Click **Apply** to confirm the change, and then click **Hide** to remove the Edit Object Properties form from the screen.

Select **Design - Hierarchy - Return**. A dialogue box appears warning that the symbol has been changed, and asking if you wish to save the changes.

Click **Yes**.

2.1.9 Checking the Schematic Design

Before checking the integrity of the top level schematic, it is necessary to set up the checking rules so that they check the VHDL properties of the design.

Select **Check - Rules Setup**. The Setup Schematic Rules Check form appears.

Set the *Packaged Objects* cyclic button to **VHDL**, and click **OK**.

Select **Check - Current Cellview**. The current Cellview will be checked and an appropriate message displayed in the CIW.

When the schematic has been checked without any errors, select **Design - Save** to save the schematic diagram.

Chapter 3

Create VHDL code

3.1 Create the VHDL Code

3.1.1 ENTITY declarations

The VHDL code for the entities present in the *Black_Box* example can be created directly from the Schematic Editor window, and checked using the Cadence Language Sensitive Editor (LSE).

Top level ENTITY declaration

Select **Design - Create Cellview - From Cellview.**

The Create Cellview from Cellview form appears.

Ensure that the *Display Cellview* and *Edit Options* buttons are checked.

Change the *To View Name* cyclic button to **entity** and click **OK.**

The LSE window opens, and the following text appears.

```
--Create Entity
--Library=bbexample, Cell=black_box, View=entity
--Time
--By

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY black_box IS
    PORT (
         \V\ : IN BIT := '0';
         \W\ : IN BIT := '0';
         \X\ : IN BIT := '0';
         \Y\ : IN BIT := '0';
         \Z\ : IN BIT := '0';
         \ZOUT\ : OUT BIT := '0');
END black_box;
```

This code simply declares the entity *black_box* as having five input ports of type BIT, and one output port, also of type BIT. All ports have the default value of 0.

Select **File - Exit** to exit the LSE.

The code will be saved automatically.

BOX_1 ENTITY declaration

Select **Design - Create Cellview - From Instance**, and then click on the rectangle representing BOX_1.

The Create Cellview from Component form appears.

Ensure that the *Display Cellview* and *Edit Options* buttons are checked.

Change the *To View Name* cyclic button to **entity** and click **OK**.

The LSE window opens, and the following text appears.

```
--Create Entity
--Library=bbexample, Cell=BOX_1, View=entity
--Time
--By

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY BOX_1 IS
    PORT (
         a : IN BIT;
         b : IN BIT;
         c : IN BIT;
         out1 : OUT );
END BOX_1;
```

This declares BOX_1 as having three inputs and one output, all of type BIT.

BOX_2 ENTITY declaration

Select **Design - Create Cellview - From Instance**, and then click on the rectangle representing BOX_2.

The Create Cellview from Component form appears.

Ensure that the *Display Cellview* and *Edit Options* buttons are checked.

Change the *To View Name* cyclic button to **entity** and click **OK**.

The LSE window opens, and the following text appears.

```
--Create Entity
--Library=bbexample, Cell=BOX_2, View=entity
--Time
```

```
--By

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY BOX_2 IS
    PORT (
         a : IN BIT;
         b : IN BIT;
         c : IN BIT;
         out1 : OUT );
END BOX_2;
```

This declares BOX_2 as having three inputs and one output, all of type BIT.

The remainder of the VHDL code will be created from the VHDL Tool Box.

Select **Window - Close** in the Schematic Editor window.

3.1.2 Structural Description of Black_Box

The structural description of the *black_box* module will be created from the VHDL Tool Box using the LSE.

In the *Working Cellview Library Name* field type **bbexample**.

In the *Working Cellview Cell Name* field type **Black_box**.

In the *WOrking Cellview View Name* field type **structure**.

Click **Edit**.

A dialogue box appears informing you that the structural view of the unit *black_box* does not exist, and asking if you wish to create it.

Click **OK**.

The LSE window opens, with a new file, for the creation of the VHDL code. In the LSE window, create the following file:

```
architecture STRUCTURE of BLACK_BOX is
  signal S1: BIT := '0' ;
  component BOX_1
    port    (
              a:in BIT ;
              b:in BIT ;
              c:in BIT ;
```

```
            out1:out BIT
          ) ;
  end component ;
  component BOX_2
    port    (
            a:in BIT ;
            b:in BIT ;
            c:in BIT ;
            out1:out BIT
          ) ;
  end component ;
  for all : BOX_1 use entity WORK.BOX_1 ( BEHAVIOR )
                      ;
  for all : BOX_2 use entity WORK.BOX_2 ( BEHAVIOR )
                      ;

begin
  B1 : BOX_1
      port map ( \V\,
                 \W\,
                 \X\,
                 S1 ) ;
  B2 : BOX_2
      port map ( S1,
                 \Y\,
                 \Z\,
                 \ZOUT\ ) ;

end STRUCTURE ;
```

This structural declaration indicates that *black_box* is made up of two types of component, namely BOX_1 and BOX_2. There is also one internal signal called S1.

The code also tells the software that for every instance of a component of type BOX_1, the behavioral description of the entity *WORK.BOX_1* should be used. Similarly, for every instance of a component of type BOX_2, the behavioral description of the entity *WORK.BOX_2* should be used.

The code then instantiates two components named B1 and B2, of type BOX_1 and BOX_2 respectively. The ports of these two instances are then mapped on to the external ports of *black_box* and the one internal signal.

When the entire text has been entered, select **File - Save Cell** and save the cell as *bbexample black_box structure.*

3.1.3 Behavioral Descriptions

The Behavioral descriptions of the two lower level entities, BOX_1 and BOX_2 will be created using the LSE.

BOX_1 Behavioral Description

The behavioral description of BOX_1 contains only one signal assignment statement, which is similar to the logic function described at the start of this tutorial.

In the *Working Cellview Library Name* field type **bbexample**.

In the *Working Cellview Cell Name* field type **BOX_1**.

In the *WOrking Cellview View Name* field type **behavior**.

Click **Edit**.

A dialogue box appears informing you that the behavioral view of the unit *BOX_1* does not exist, and asking if you wish to create it.

Click **OK**.

The LSE window opens, with a new file, for the creation of the VHDL code. In the LSE window, create the following file:

```
ARCHITECTURE BEHAVIOR OF BOX_1 IS
begin
  out1 <= (a AND b) OR (a AND c) after 12 ns;
end BEHAVIOR;
```

When the text has been entered, select **File - Save Cell** and save the cell as *bbexample BOX_1 behavior.*

BOX_2 Behavioral Description

The behavioral description of BOX_2 is similar to that of BOX_1, except that the signal assignment expression is different.

In the *Working Cellview Library Name* field type **bbexample**.

In the *Working Cellview Cell Name* field type **BOX_2**.

In the *WOrking Cellview View Name* field type **behavior**.

Click **Edit**.

A dialogue box appears informing you that the behavioral view of the unit *BOX_2* does not exist, and asking if you wish to create it.

Click **OK**.

The LSE window opens, with a new file, for the creation of the VHDL code. In the LSE window, create the following file:

```
ARCHITECTURE BEHAVIOR OF BOX_2 IS
begin
  out1 <= (a OR b) AND (b OR c) AND (a OR c) after 12 ns;
end BEHAVIOR;
```

When the text has been entered, select **File - Save Cell** and save the cell as *bbexample BOX_2 behavior*.

3.1.4 Checking the VHDL Hierarchy

Checking the VHDL Hieracrchy is carried out by setting the structural description of *black_box* to be the Top of Hierarchy.

In the *Top of Hierarchy Library Name* field type **bbexample**.

In the *Top of Hierarchy Cell Name* field type **black_box**.

In the *Top of Hierarchy View Name* field type **structure**.

Select **Commands - Check Hierarchy**.

A dialogue box will appear indicating whether or not the VHDL Hierarchy Check was successful.

Chapter 4

Create a Test Bench

4.1 Create a Test Bench

4.1.1 Truth Table for Black Box System

Before creating the VHDL code for the Test Bench, it is necessary to draw up a truth table for the design, showing all possible input combinations, together with the anticipated output.

The truth table for the Black Box design is shown below.

V	W	X	Y	Z	ZOUT
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	0	1	1
1	1	1	1	0	1
1	1	1	1	1	1

4.1.2 Creating the VHDL Test Bench code

The VHDL Test Bench code can be created from within the VHDL Tool Box, using the Language Sensitive Editor.

Select **Commands - Create Test Bench.**

The VHDL Create Test Bench form appears. The software will now create a test bench consisting of two components, an ENTITY called *test* which has no input or output ports, and an ARCHITECTURE called *stimulus* which contains the test bench routines.

Click **OK.**

The LSE opens, with the following code :

```
--Create Test Bench:
--Library=bbexample,Cell=test,View=stimulus
--Time:Wed Jul  5 15:04:04 1995
--By:pmills
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ARCHITECTURE stimulus OF test IS

    COMPONENT black_box
        PORT(
            \V\ : IN  BIT;
            \W\ : IN  BIT;
            \X\ : IN  BIT;
            \Y\ : IN  BIT;
            \Z\ : IN  BIT;
            \ZOUT\ : OUT  BIT);
    END COMPONENT;

    -- Fill in values for each generic

    -- Fill in values for each signal
    SIGNAL \V\ : BIT;
    SIGNAL \W\ : BIT;
    SIGNAL \X\ : BIT;
    SIGNAL \Y\ : BIT;
    SIGNAL \Z\ : BIT;
    SIGNAL \ZOUT\ : BIT;

BEGIN
```

```
    dut : black_box

        PORT MAP (\V\, \W\, \X\, \Y\, \Z\, \ZOUT\);

END stimulus;
```

This is skeleton code for the test bench, and must be edited by the user to suit the test routines required.

The data for the test input values as shown in the truth table will be held in a set of 32 vectors, each of which is made up of 6 bits. The first five bits contain the input values, and the sixth contains the anticipated output value. This is achieved using the following code, which should be inserted after the six SIGNAL statements :

```
subtype TEST_VECTORS is BIT_VECTOR ( 0 to 5 );
type VECTOR_SET is array ( NATURAL range <> ) of TEST_VECTORS ;

constant MY_VECTOR_SET: VECTOR_SET := (
          "000000","000010","000100","000111",
          "001000","001010","001100","001111",
          "010000","010010","010100","010111",
          "011000","011010","011100","011111"
          "100000","100010","100100","100111",
          "101000","101011","101101","101111",
          "110000","110011","110101","110111",
          "111000","111011","111101","111111");
```

The main body of the test code consists of a loop which is executed 32 times, one for each set of input values. Upon each iteration, the five five bit values are sent to the input ports of BLACK_BOX, and after a pause to allow the unit to respond, the output is tested against the sixth bit value using the ASSERT statement. If this check is not successful, a warning is displayed in the CIW window. The main body of the test bench code which is as follows should be inserted within the main body of the code, after the PORT MAP statement.

```
process
    begin
        for i in 0 to 31
          loop
            \V\ <=  MY_VECTOR_SET(i)(0)   ;
            \W\ <=  MY_VECTOR_SET(i)(1)   ;
```

```
            \X\ <=  MY_VECTOR_SET(i)(2)  ;
            \Y\ <=  MY_VECTOR_SET(i)(3)  ;
            \Z\ <=  MY_VECTOR_SET(i)(4)  ;

            wait for 20 ns ;

            assert \ZOUT\ = MY_VECTOR_SET(i)(5)
                report " ********** Test Vector Failed ******"
                severity warning ;

          end loop ;
        wait ;

end process  ;
```

The entire architecture file should now read as follows :

```
--Create Test Bench:
--Library=boxtest,Cell=test,View=stimulus
--Time:Tue Jul  4 14:43:34 1995
--By:pmills
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ARCHITECTURE stimulus OF test IS

  COMPONENT black_box
      PORT(
          \V\ : IN  BIT;
          \W\ : IN  BIT;
          \X\ : IN  BIT;
          \Y\ : IN  BIT;
          \Z\ : IN  BIT;
          \ZOUT\ : OUT  BIT);
  END COMPONENT;

  -- Fill in values for each generic

  -- Fill in values for each signal
  SIGNAL \V\ : BIT;
  SIGNAL \W\ : BIT;
  SIGNAL \X\ : BIT;
  SIGNAL \Y\ : BIT;
```

```
  SIGNAL \Z\ : BIT;
  SIGNAL \ZOUT\ : BIT;

  subtype TEST_VECTORS is BIT_VECTOR ( 0 to 5 );
  type VECTOR_SET is array ( NATURAL range <> ) of TEST_VECTORS ;

  constant MY_VECTOR_SET: VECTOR_SET := (
            "000000","000010","000100","000111",
            "001000","001010","001100","001111",
            "010000","010010","010100","010111",
            "011000","011010","011100","011111",
            "100000","100010","100100","100111",
            "101000","101011","101101","101111",
            "110000","110011","110101","110111",
            "111000","111011","111101","111111");

BEGIN

  dut : black_box

      PORT MAP (\V\, \W\, \X\, \Y\, \Z\, \ZOUT\);

  process
      begin
          for i in 0 to 31
            loop
              \V\ <=  MY_VECTOR_SET(i)(0)  ;
              \W\ <=  MY_VECTOR_SET(i)(1)  ;
              \X\ <=  MY_VECTOR_SET(i)(2)  ;
              \Y\ <=  MY_VECTOR_SET(i)(3)  ;
              \Z\ <=  MY_VECTOR_SET(i)(4)  ;

              wait for 20 ns ;

              assert \ZOUT\ = MY_VECTOR_SET(i)(5)
                report " ********** Test Vector Failed ******"
                severity warning ;

            end loop ;
          wait ;

  end process  ;
```

```
END stimulus;
```

Chapter 5

Define the Hierarchy

5.1 Define the Hierarchy

There are three types of hierarchy which may be specified :

- Simple. For designs that have component configurations contained in the architecture body, or for designs which have only one view per cell.
- VHDL Configuration. For designs that have a VHDL Configuration design unit. This option supports instance-specific bindings and subconfigurations.
- Switch List. For designs that are already defined structurally with an iterative design or synthesis and that do not need all the complexity of a VHDL Configuration.

In the BLACK_BOX example, the configuration is contained within the architecture body, so the hierarchy is a simple one.

Select **Setup - Hierarchy** in the VHDL Tool Box. The VHDL Seup Hierarchy form appears.

Select **Simple** on the VHDL Setup Hierarchy form.

Click **OK**.

Chapter 6

Simulate and Debug

6.1 Simulate and Debug the Design

6.1.1 Checking the Test Bench Hierarchy

In order to run the simulation, it is necessary to set the top of hierarchy top be the test bench. In the VHDL Tool Box window, ensure that the Top of Hierarchy reads *bbexample test stimulus.*

Click on the **Check Hierarchy** button in the VHDL Tool Box.

The system will check, netlist and elaborate your design. A dialogue box will appear, indicating the success (or otherwise) of this action, and asking if you wish to view the log file.

Click **Yes**. The resulting log file should look as follows :

```
Netlisting library: pmblackbox,
           cellView: "test stimulus"
SearchPath: ".  /net/mazo/cadence/9401/tools.sun4/df........
Traversing pmblackbox.test(stimulus)
Traversing pmblackbox.black_box(structure)
Traversing pmblackbox.box_1(behavior)
Traversing pmblackbox.box_2(behavior)
Netlisting entity box_2
Netlisting architecture behavior of entity box_2
Netlisting entity box_1
Netlisting architecture behavior of entity box_1
Netlisting entity black_box
Netlisting architecture structure of entity black_box
Netlisting entity test_box
Netlisting architecture stimulus of entity test
Analyzing pmblackbox.box_1(entity)
Analyzing pmblackbox.box_1(behavior)
Analyzing pmblackbox.box_2(entity)
Analyzing pmblackbox.box_2(behavior)
Analyzing pmblackbox.black_box(entity)
Analyzing pmblackbox.black_box(structure)
Analyzing pmblackbox.test(entity)
Analyzing pmblackbox.test(stimulus)
Elaborating design.
```

6.1.2 Running the Simulation

The system design, along with the Test Bench, will be simulated using the Leapfrog VHDL Simulation package, run from the VHDL Tool Box.

Select **Commands - Simulate** in the VHDL Tool Box. The Leapfrog VHDL Simulation window appears.

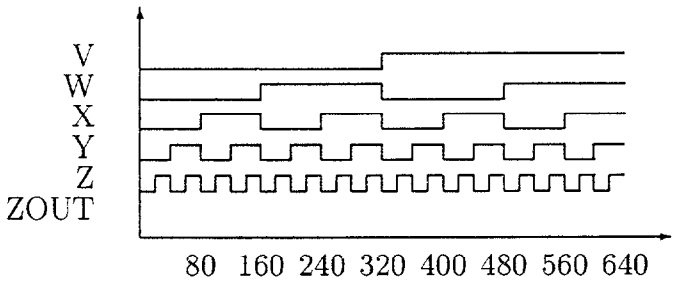
In the *Scope* field, hold down the right mouse button, and select **Set Trace - Simple**. The Wave View window appears. This enables the user to observe the waveforms of both the input and output signals during the simulation.

Resize both the Leapfrog VHDL Simulator window, and the Wave View window so that both can be viewed on the screen at the same time.

In the Leapfrog VHDL Simulation window, click **Run**.

The simulation runs to completion, and a message box appears informing you that the simulation is complete. Click **Close**.

In the Wave View window, select **View - Fit**. The graph should now show the complete waveforms for the six signals over the entire simulation time.



This graph can be compared with the truth table which was drawn up prior to creating the Test Banch code. Any discrepancies can be seen on the plot. Note that any discrepancies will also have been reported in the CIW, due to the ASSERT statement in the Test Bench code.

6.1.3 Cycling through the simulation

If you want to cycle throught the simulation, showing the result from one set of inputs at a time, this can be done by setting a breakpoint in the loop of the test banch code.

In the Leapfrog VHDL Simulator window, locate the ASSERT command in the source code. This should be around line 62. With the mouse pointer on the line number, hold

down the right mouse button, and select **Set Breakpoint - Simple**. A little stop sign will appear beside the line number, indicating a breakpoint.

To reset the simulation time to zero, select **Misc - Restore**. The Restore Simulation window appears. In the Snapshots list box, select *SIM*.

Click **OK**. The simulation time is now reset to zero.

To step through the simulation, click **Run**. The simulation will run until it reaches the first breakpoint, after one traversal of the loop, and it will then pause. Click **Run** to execute the next cycle. Note that the plot in the Wave View window will be updated after each cycle.

6.1.4 Stepping through the simulation

If you want to step through the simulation in smaller steps, showing the result of every action, this can be done using the **Step** command.

In the Leapfrog VHDL Simulator window, locate the breakpoint set in the previous section. With the mouse pointer on the line number, hold down the right mouse button, and select **Delete Breakpoint**. The little stop sign will disappear, indicating that the breakpoint has been cleared.

To reset the simulation time to zero, select **Misc - Restore**. The Restore Simulation window appears. In the Snapshots list box, select *SIM*.

Click **OK**. The simulation time is now reset to zero.

To step through the simulation, click **Step**. The simulation will step through each action in the source code. Note that some actions may have no effect on the state of any of the signals, and also that the Wave View window will only be updated when the simulation time is incremented.

6.1.5 Exiting from the Leapfrog Simulation

To exit from the simulator, select **Window - Quit**. The Leapfrog VHDL Simulator window closes.

Chapter 7

Synthesis

7.1 Synthesize the Design

7.1.1 Analyzing the Design Files

In order to run the simulation, it is necessary to ensure that the Top of Hierarchy is set correctly. In the VHDL Tool Box window, click on the **Browse** button below the Top of Hierarchy Library Name. When the Library Browser Window appears, select the cellview *bbexample - black_box - structure.*

Click on the **Sythesis** button in the VHDL Tool Box.

The VSH window appears, and the system will automatically check, netlist and elaborate your design. All the design units will also be analyzed. This transforms the VHDL source code into an intermediate form which is suitable for synthesis. It also checks the source file for syntax errors, and some modelling style errors.

7.1.2 Checking the Design

Having analyzed the design, you must now check that all the design units have been analyzed in the correct order, and that none of them are out of date. This is an essential part of the synthesis process, as synthesis cannot be carried out until the integrity of the intermediate files has been verified.

Select **Libraries - List Units** from the VSH window. The Libraries: List Units form appears.

In the *Library List* cyclic field, select **bbexample.**

The Units List form appears, displaying a list of all the design units in the *bbexample* library.

Highlight the design unit *Black_Box.Structure* and click **OK.**

In the VSH window select **Units - Ready to Synthesize.** This checks the top level of the design to see if the design is ready for synthesis.

If the message *Unit "Bbexample.Black_Box.Structure" is ready for synthesis* appears in the VSH window, then the design is ready to be synthesized.

If this is not the case, then any syntax errors must be corrected, and the design reanalyzed and updated.

7.1.3 Running The Synthesis

Before the design can actually be synthesized, the synthesis environment must be set up. This includes creating a new directory in which all the synthesis files will be stored.

Select **Units - Synthesize** in the VSH window. The Initialize Run Directory form appears.

Check that the Design Unit name is the same as the top level of your design, i.e. *Black_Box.Structure*.

Check that the name of the run directory in the Synthesis Run Directory field is acceptable, and click **OK**.

The VSH window changes into *syn* mode, with the title changing to "VHDL Synthesizer".

7.1.4 Setting up a Target Library

The system needs to know where to find information regarding the cells which will make up the synthesized design. This is done by specifying a Target Library.

Select **Session - Options - Synthesis Library** from the VHDL Synthesizer window. The Synthesizer Options form appears.

Check the *Target Library Symbol Libraries* field to make sure that the *basic* and *US.8ths* libraries are in your search path. These libraries contain special symbols used in schematic generation, and also borders for the schematic views.

Change both the *Source Library* and *Target Library Type* to *Generic* and click **OK**.

The Synthesizer Options window disappears.

7.1.5 Specifying Constraints

By setting constraints, it is possible to dictate, to a certain extent, the nature of the circuit which will be produced by the synthesizer. There are 2 types of constraint which may be specified :

Goals are those constraints which the optimizer will attempt to satisfy, but which may not be met. Any goals which are not met will be reported on the Violations Report.

Controls are those constraints which must be met by the optimizer.

Global Cost Constraints

Global constraints are those which apply to the entire design, not just to a single object. A cost constraint designates the size of circuit to be created. The optimizer selects cells in the target library which allow it to meet the cost constraints as closely as possible. Note that this type of constraint is a *goal*, and as such may not be satisfied.

Select **Constraints - Globals**. The Global Constraints form appears.

In the *Maximum Cost* field, type **5**.

Click **OK**.

The Global Constraints form disappears.

Object Cost Constraints

Object Cost constraints are those constraints which are applied to a single object in the design, rather than to the whole design. When a design contains both global and object constraints, the optimizer attempts to meet the object constraints first, and then uses these results in the higher level optimization. It should also be noted that when constraints are applied to obects, the hierarchy of that object is retained within the design, and is not merged with other units. Therefore, if a constraint is applied to BOX_2, then BOX_2 will still appear as a unit in the top level schematic, whereas BOX_1, with no constraints, will be split into its various components. BOX_2 will have a schematic in its own right at the next lower level.

Select **Constraints - Cost...** The Cost Constraints form appears.

In the *Design Unit* field, type **Bbexample.Box_2.Behavior**.

In the *Max Cost* field, type **3**.

Click **OK**.

The Cost Constraints form disappears.

Managing Constraints

In order to view, edit, and enable or disable the various constraints, use the Constraints Manager.

Select **Constraints - Constraint Manager**.

The Constraints Manager window appears detailing all the constraints which have been defined for the current design. Within this window, it is possible to enable and disable one or more of the various constraints in order to observe the effects of each one.

To exit from the Constraint Manager, select **Window - Close.**

The constraints may also be viewed, but not edited, by selecting **Constraints - Show** from the VSH window. A text window appears, listing all the constraints. To close this window, select **File - Close Window.**

7.1.6 Invoking a Synthesis Run

Having specified the libraries to be used during synthesis, and also specified the constraints to be applied, the design may now be synthesized.

Select **Run - Synthesizer** from the VHDL Synthesizer window. The Run VHDL Synthesizer and Optimizer form appears.

When invoking a synthesis run, there are a number of different optimization techiques which may be used :

Normal - This technique should be used under the following circumstances :

1. When performing the first syhthesis run on a design, and you want to be sure that the design contains no style errors, and that the constraints have been set correctly.
2. When there are very specific area or timing constraints in mind for the design.

Trade off Curve - Once the design has been verified to have no style errors, and the constraints set correctly, you may wish to generate a trade-off curve. This is a graphical display of a range of circuits that can be generated from the same design description. The points generated on the trade off curve represent potential circuits, with specific area and delay values. As such, small circuits may produce only a small number of points. Note that under the trade off curve option, no netlists or schematics are generated.

Select Point Run - After generating a trade off curve, you can carry out a Select Point run to generate the schematic and other reports for a selected point on the curve.

Mapping Run - In a mapping run, no changes are made to the logic structure of the design. VHDL Synthesizer and Optimizer reads in the design, decomposes the design into combinational logic and sequential logic, and then maps the logic to cells in the target library. This option may be useful in under the following circumstances :

1. When the logic structure of the design is satisfactory, and you wish to map it to a different technology.
2. When you wish to transfer a RTL design into a structural design without any optimization.

The example design is a very simple one, so *normal* optimzation will be used.

Check to see that the *Generate Schematic* button is set to *yes* and that the Optimization Alternative cyclic field is set to *normal.*

Set the *Processing Effort* field to *1.* This controls the amount of CPU time which is spent in both optimizing and mapping, before selecting the best design.

For the first run through of a design, select 1. If the required results are not achieved, the design can be re-synthesized using a higher processing effort.

Click **OK** to start the VHDL Synthesizer and Optimizer.

7.1.7 Viewing Run Status

While the synthesis and optimization is being carried out, you can monitor the execution status of the job using the Job Monitor.

Select **Session - Job Monitor** from the VHDL Synthesizer window. The Analysis Job Monitor form appears on the screen, listing all the jobs executed during the current session. The status indicates whether the job is running, suspended, killed, failed or has succeeded.

Click on the check box beside the currently running job to select it, and display the current log by clicking *Show Run Log.*

The run log will inform the user as to the current step being executed in the synthesis and schematic generation, and also which reports are being generated. The log will inform the user as to whether or not the synthesis and schematic generation has been successful, whereupon the window may be closed.

To close the view window, select **File - Close Window.**

To close the job monitor, click **OK** in the Job Monitor window.

7.1.8 Viewing Synthesis and Optimization Reports

Constraints Violation Report

The Constraints Violation report lists all the constraints which were established prior to running the synthesis, along with required values, actual values, and whether or not the constraint was violated.

Select **Show - Report - Violation** from the VHDL Synthesis window.

A View File window opens with the first page of the Constraints Violation report displayed.

To exit the report, select **File - Close Window.**

Timing Report

The Timing Report lists a number of the longest and shortest critical paths through the synthesized design, along with some other timing information.

Select **Show - Report - Timing** from the VHDL Synthesis window.

A View File window opens showing the Timing Report.

To exit the report, select **File - Close Window**.

Cost Report

This Cost Report lists the components in the synthesized design, along with their associated areas and costs. It also lists the blocks and nodes in the synthesized design.

Select **Show - Report - Cost** from the VHDL Synthesis window.

A View File window opens showing the Cost Report.

To exit the report, select **File - Close Window**.

Netlist Statistics

The Netlist Statistics Report shows the types and number of gates in the synthesized design, along with the number and direction of pins etc.

Select **Show - Report - Netlist Statistics** from the VHDL Synthesis window.

A View File window opens showing the netlist statistics for the synthesized design.

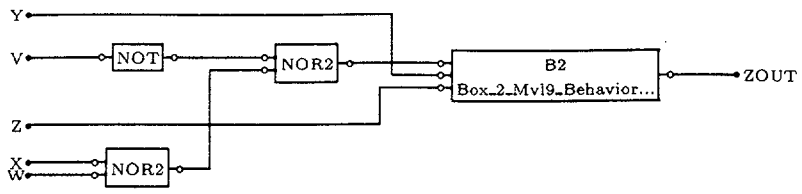
To exit the report, select **File - Close Window**.

7.1.9 Viewing the Schematic of the Synthesized Design

The schematic layout which was generated during the synthesis process may be viewed using the Schematic Editor.

Select **Show - Output - Schematic** from the VHDL Synthesizer window.

The Schematic Editor window opens, with read permission only, showing the schematic diagram for the synthesized design, similar to that shown below :

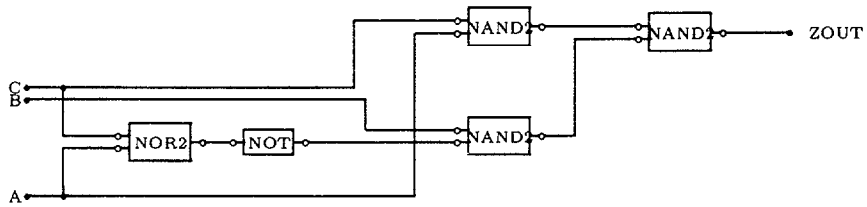


Note that the unitBOX_2 still appears in its own right in the top level schematic, its hierarchy being preserved due to the constraints which were placed on it.

Select **Design - Hierarchy - Descend Read**, and then click on the recatangle representing BOX_2.

The Descend Hierarchy form appears. In this form, set the view name to *schematic* and click **OK**.

The schematic layout for the entity BOX_2 appears, similar to that shown below.



To return to the top level schematic, select **Design - Hierarchy - Return**.

The top level schematic reappears.

To exit the Schematic Editor, select **Window - Close**.