CIRCUIT DESCRIPTION AND CIRCUIT SIMULATION USING C++

ROZITA BT OTHMAN

WEK010397

Faculty Of Computer Science And Information Technology ,

University Of Malaya

Mac 2005

ABSTRACT

Techniques used for circuit description have often been treated differently from the methods used to simulate hardware. This report will attempt to demonstrate how a common platform, and hence a uniform environment, can be established to both describe and simulate hardware, using a programming paradigm known as *object-oriented programming* as supported by the C++ language. This paradigm is powerful enough to allow a hierarchical description of the hardware and at the same time provide an extensible method for its simulation.

This report will provide a brief overview of C++ and its support for the object-oriented programming paradigm.

ACKNOWLEDGEMENT

Utter most gratitude goes to the almighty Allah for all the confidence and patience in the completion of my thesis

I would like to express my deep gratitude to my supervisor Mr. Yamani Idna Idris for the tremendous help he has given me during this project, technical advise and thoughtful comment. And also to my moderator Ms. Rafidah Mohd Noor for her valuable advise and motivation.

Also taking this opportunity expressing my thanks to all fellow members and especially the family of Computer Science and Networking for their support to face the difficulties and challenging time.

Finally but not least, I am much obliged to my lovely husband, Mawardi Saad and my parents who have been given invaluable support and inspiration to me throughout my university life.

TABLE OF CONTENTS

CHAPTER IV: SYSTEM ANALYSIS

CHAPTER V: SYSTEM DESIGN

CHAPTER VI: SYSTEM IMPLEMENTATION AND CODING

CHAPTER VII: SYSTEM TESTING

CHAPTER VIII: DISCUSSION

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Introduction

As high-level computer programming languages become increasingly more powerful and abstract, there is a tendency amongst the computing society to use such general purpose languages for the benefit of their own fields of research and study. Instead of using highly specific languages, which are usually restricted in their scope and availability, members of both the software and hardware community are tempted to use higher level languages that are more accessible and flexible. This report will show how one such high-level language and its support for the object-oriented programming paradigm can be used for both the description and simulation of hardware modules.

The advantages of using a popular general purpose language for hardware description and simulation are multifold. Firstly, if the hardware designer is already familiar with the language from software development, then all he or she has to do is to apply the language to the problem of hardware description and simulation. This means that the designer does not have to learn an entirely new language with its own peculiar syntax rules and idiosyncrasies. Secondly, general purpose languages have a much greater population of users than do languages which specifically support hardware description. Therefore, translators, compilers and/or interpreters for such high-level languages are usually more readily available, efficient and reliable. Thirdly, using the same language to both describe and simulate hardware creates a more uniform and consistent environment for the user. This uniform environment results in more control, since the designer does not have to keep track of one language for circuit description and an entirely different language for parsing the circuit description and simulating it.

Obviously, some high-level languages are better suited for hardware description and simulation than others. A language which can be used for hardware description and simulation must permit the designer to specify the hardware in an intuitive and hierarchical manner, consistent with how the designer thinks. In addition, such a language must be easily extensible so that new modules can be easily added without any modifications being made to the simulation code. Ideally, such a language must also provide support for circuit designs at several levels of abstraction and should also give the designer the choice of either using a ``bottom-up" or a ``top-down" approach to circuit description.

One language which appears to satisfy all the requirements for a hardware description and simulation language is C++. In addition to providing support for object-oriented programming, this language is also undergoing standardization by ANSI, which means that the language mechanisms and features used by this report should be portable across most C++ compilers. Since use of the language is undergoing exponential growth, C++ compilers and translators have be written for a variety of platforms. Therefore, C++ is available to a wide community of programmers.

1.2 Problem Definition

When designing a hardware, commonly we used HDL or VHDL hardware description language. There are some major obstacles to do successful system-on-chip. Some of the obstacles are:-

(i) There is a lack of a single system-level environment that can be used throughout the design flow. While algorithms might be designed at high level in C, gates still have to synthesised out of HDL. Each manual format translation, no matter how small, is a possible source of errors.

(ii) The designer has insufficient control over the design process. He or she has to accept the result that (synthesis) tools produce. This is result of those tools being sold as closed boxes. Assembling a system level design flow out of such tools however requires an open environment.

(iii) There is lack of a systematic verification strategy. There are as many testbenches as there are tools used the design flow.

Since hardware description languages are closely related to programming language, it is natural to try to adapt object-oriented technique to hardware description.

1.3 Scope

In this project scope, there are two major sections dealing with C++ and the object-oriented paradigm, circuit description and circuit simulation using C++.

1. The first section serves as an introduction to the basic mechanics and techniques associated with the object-oriented paradigm and how C++ provides support for this paradigm.
2. Describe and simulate circuits using this paradigm. It will be shown how concepts of object-oriented programming map cleanly into the problem of hardware description.

1.4 Objectives

There are two types of objective that must be achieve, so that the target of system development can sucessfully accomplish.

1. General/overall project objective
2. Explicit project objective.

Some of the overall project objectives are:-

(i) To understand system process or system development

(ii) To practice all the knowledge that we have learned such as skill in system analysis, system design and programming and also system development.

(iii) To adapt with all the software that related in designing system.

(iv) To understand all the problem and constraint in developing system and software.

Explicit project objectives are:-

(i) To discuss the advantages of C++ as object-oriented programming to describe and simulate hardware.

(ii) To introduce classification of circuit components which is component, connector, wire and port programming in C++.

(iii) To describe simple hardware using C++ programming which is **AND** gate, **RS-Latch**, **Adder** and the result of simulation.

1.5 Schedulling

The bar chart below shows the activities of each process phase that will be carried out through the development of the system. It will take an approximate time of 9 months to finish the whole thesis project. Starting on the first phase, which is system analysis from June to July. At this phase, information is collected on systems available and study is made on methodology that will be used in this project.

The second phase starts from August until September, which is working on the system design. At the beginning of October and second part of the thesis will be started by the implementation of the system, which is the system coding. System testing will be carried out at the middle of December until the end of January. The system will be tested to check if it's free from errors.

The last phase of system development is the system evaluation. It starts at the end of January until the end of February. The required system output will be checked in this phase.

System Development

No.	Year	2004				2005					
	Phase & Month	Jun	Jul	Og	Sep	Okt	Nov	Dis	Jan	Feb	Mac
1	Identify Constraint and Objectives	■	■	■	■	■					
2	Identify Information		■	■	■						
3	System Analysis		■	■	■						
4	System Design				■	■	■				
5	Documentation and Develop Software		■	■	■	■	■	■	■	■	■
6	System Testing and Maintenance								■	■	■
7	Implement & System Evaluation										■

CHAPTER II

LITERATURE REVIEW

2.1 C++ Background and History

C++ is a programming language developed at AT&T Bell Laboratories by Bjarne Stroustrup in the early 1980's. The language was designed with the intent of merging the efficiency and conciseness of C with the object-oriented programming features of SIMULA-67. Since its creation, the language has evolved rapidly and several new features have been added since its initial release in 1985. The language also promises to provide support for several other useful mechanisms such as parameterized types and exception handling in the near future. A formal ANSI-C++ committee (X3J16) has since been established to help develop an accurate and reliable standard for the language which should eliminate most, if not all, ambiguities in the C++ compilers and translators of today. It is expected that this committee will adopt most of the rules present in the ANSI base document *The Annotated C++ Reference Manual* as written by Ellis and Stroustrup.

With a few modest exceptions, C++ can be considered a superset of the C programming language. While C++ is similar to C in syntax and structure, it is important to realize that the two languages are radically different. C++ and its support for object-oriented programming provide a new methodology for designing, implementing and ease of maintaining software projects which C, a structured programming language, is unable to support. Extensive libraries are available for the C programming language; consequently, a deliberate effort was made on behalf of the developers of C++ to maintain backward compatibility with C. Any major deviation from the C programming language would have meant that all the libraries available for C would have to be tediously rewritten for C++. This would have severely limited the usefulness of C++ in an environment where C libraries were used extensively.

2.1.1 Features Borrowed from Other Languages

C++ is largely an amalgamation of several other programming languages. Obviously, C++ inherits most of its characteristics, such as its syntax, looping mechanisms and the like, from C. A part from C, C++ borrows most heavily from the aforementioned SIMULA-67 programming language. Nearly all the support that C++ provides for object-oriented programming comes from this language. The concept of a class and the so-called virtual function mechanism are a few of the features present in SIMULA-67 which have been integrated in C++.

To a limited extent, C++ also borrows some programming mechanisms from Algol-68. These include support for operator overloading and the declaration of variables almost anywhere in the code. The newer C++ compilers will provide support for parameterized types and exception handling, concepts borrowed from Ada and Clu.

2.1.2 C++ Versus C

When a user defines a type in C++, support is provided in the language to permit that type to behave in a manner similar to types already built into the language. The user may define how the standard operators act upon these user defined types (operator overloading) and how these type can be converted to another type (user defined conversions). The user may also specify how memory is allocated or deallocated when an instance of that type is created or destroyed. This is done through the use of constructors and destructors which are called implicitly by the compiler when an instance of that type is brought into and taken out of scope respectively.

C++ provides support for function prototypes, hence enabling strong type checking of function parameters to take place during compilation. In addition, C++ provides support for the *pass by reference* mechanism and also supports default arguments to functions. This means that should a function require an argument that often has the same value, the user can default the argument to that value and not pass that parameter when the function is called. In the few cases where the function has to be called with a different value for the default argument, the user simply passes that argument into the function and the new value overrides the default value.

The most important features which C++ provides support for are data encapsulation, inheritance and runtime binding which form the foundation for the language's support for object-oriented programming.

2.2 Object-Oriented Programming

The object-oriented paradigm was first conceived in the 1960's and implemented in languages such as SIMULA-67. One of the initial concerns with early object-oriented languages was their efficiency. Programs written using structured languages, such as Pascal and C, executed faster than programs written using early object-oriented languages. Although programs which used the object-oriented paradigm were more extensible and easier to maintain from a programmer's point of view, an unacceptable price had to be paid in the program's runtime behaviour. Recently, however, the runtime execution of object-oriented programs has improved considerably. This has been due in part to both the development of faster hardware and the creation of efficient languages and compilers which support object-oriented programming, such as C++. These facts, in addition to the ever-increasing accessibility of object-oriented languages to the common programmer has created a major evolution in the area of software development.

There is, as yet, no universally agreed upon definition of exactly what constitutes object-oriented programming. Booch suggests:

``Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships."

From this definition, one can infer that object-oriented programming consists of instantiating a number of objects which communicate with one another so as to achieve some desired behaviour. This paradigm is natural with how humans see the world; as a series of cause-effect relationships, where an action performed on one object has effects on the objects with which it communicates.

2.2.1 The Problem of Complexity

Object-oriented programming is particularly useful for reducing problems of complexity associated with the writing of large scale software packages. It is useful to think of a typical complex system as consisting of two orthogonal planes of hierarchy (**Figure** 2.1).

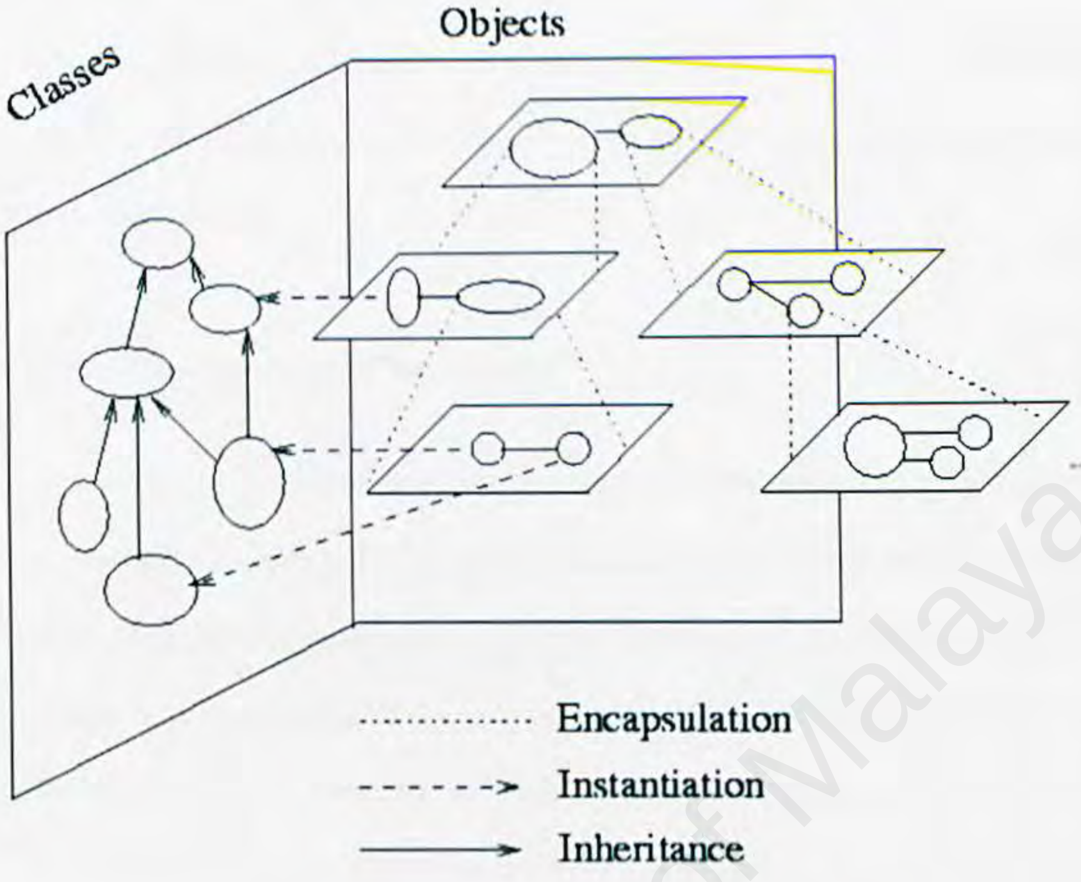


Figure 2.1: The Problem of Complexity

The first plane consists of a set of *classes*, each of which acts as a ``blueprint" for the instantiation of *objects*. This plane contains a hierarchy known as the ``is a kind of" relationship. Any class, *D*, in the plane which points to another class, *B*, is said to be *a kind of* class *B*. All the properties of class *B* (commonly called a *base class*) are passed down to class *D* (referred to as a *derived class*). As can be seen from the diagram, a derived class can be used as the base class of another; and a class can be specified as being ``a kind of" two or more classes. This ``is a kind of" hierarchy is supported in object-oriented languages through the mechanism of *inheritance*. The second plane consists of a series of *objects* which, as mentioned above, are instantiated from the classes of the first plane. This hierarchy is called the "is a part of" relationship and is achieved through the encapsulation, or aggregation, of classes. The hierarchy suggests that one object may be comprised of several objects; each of which can be made up of even more objects.

2.2.2 The Problem of Classification

One of the major problems encountered by designers of object-oriented software is *classification*; that is, finding which classes should be grouped together under a shared base class. When attempting to perform classification on the problem space, several issues must be addressed. For example, the designer must decide which properties should be used to determine commonality. The classification should also be flexible enough to permit the introduction of new objects into the system which appear to belong to neither class nor appear to have properties of several classes.

2.3 Features of Object-Oriented Programming Languages

Object-oriented programming languages support three features: data *encapsulation*, *inheritance* and *dynamic binding of function calls*. Each helps the programmer build more abstract, powerful and malleable data types.

2.3.1 Data Encapsulation

Data encapsulation, sometimes referred to as data hiding, is the mechanism whereby the implementation details of a class are kept hidden from the user. The user can only perform a restricted set of operations on the hidden members of the class by executing special functions commonly called *methods*. The actions performed by the methods are determined by the designer of the class, who must be careful not to make the methods either overly flexible or too restrictive. This idea of hiding the details away from the user and providing a restricted, clearly defined interface is the underlying theme behind the concept of an *abstract data type*.

The advantage of using data encapsulation comes when the *implementation* of the class changes but the *interface* remains the same. The concept of data encapsulation is supported in C++ through the use of the `public`, `protected` and `private` keywords which are placed in the declaration of the class. Anything in the class placed after the `public` keyword is accessible to all the users of the class; elements placed after the `protected` keyword are accessible only to the methods of the class or classes derived from that class; elements placed after the `private` keyword are accessible only to the methods of the class.

As a convention, calling a method of an object instantiated from a class is commonly referred to as sending a *message* to that object.

2.3.2 Inheritance

Inheritance is the mechanism whereby specific classes are made from more general ones. The child or derived class inherits all the features of its parent or base class, and is free to add features of its own. In addition, this derived class may be used as the base class of an even more specialized class.

Inheritance, or derivation, provides a clean mechanism whereby common classes can share their common features, rather than having to rewrite them.

Inheritance is supported in C++ by placing the name of the base class after the name of the derived class when the derived class is declared. It should be noted that a standard conversion occurs in C++ when a pointer or reference to a base class is assigned a pointer or reference to a derived class.

2.3.3 Dynamic Binding of Function Calls

Quite often when using inheritance, one will discover that a series of classes share a common behaviour, but how that behaviour is implemented is different from class to class. Such a situation is a prime candidate for the use of dynamic or runtime binding which is also referred to as *polymorphism*.

C++ implements dynamic binding through the use of *virtual* functions. While function calls resolved at runtime are somewhat less efficient than function calls resolved statically, Stroustrup notes that a typical virtual function invocation requires just five more memory accesses than a static function invocation. This is a very small penalty to pay for a mechanism which provides significant flexibility for the programmer, as will be shown later.

It is from inheritance and runtime binding of function calls that object-oriented programming languages derive most of their power. Some problems lend themselves very well to these two concepts, while others do not. As Stroustrup notes:

> ``How much types have in common so that the commonality can be exploited using inheritance and virtual functions is the litmus test of the applicability of object-oriented programming."

2.4 VHDL (VHSIC Hardware Description Language)

VHDL is one of the computer language for describing digital systems. VHDL is a hierarchical acronym denoting VHSIC **H**ardware **D**escription **L**anguage; VHSIC in turn, denotes **V**ery **H**igh **S**peed **I**ntegrated **C**ircuits. The United States Department of Defense Very High Speed Integrated Circuits (VHSIC) Program initiated the design of VHDL to support the development of a new generation of digital system technology. The design of VHDL formally began in 1983 and after a series of several iterations and versions, culminated in 1987 with the acceptance of VHDL as an IEEE (Institute of Electrical and Electronic Engineers) standard.

2.4.1 Basic Language Organization

Figure 2.2(a) shows the graphical symbol and VHDL model of a 2-input and operation. The VHDL model shown in Figure2.2(b) comprises a design entity, which is the basic construct in VHDL for modeling a digital system. The digital system can be physical piece of hardware that has been designed or a conceptual piece of hardware that is being designed.

VHDL design entity is composed of two parts: an *interface* and a *body*. The interface is denoted by the keyword **entity** and the body is denoted by the keyword **architecture**. A convenient way to view the roles of the *interface* and a *body* is to imagine a digital system enclosed inside a casing or black box. The *interface* describes aspects of the digital system visible outside the black box that define the boundary between the system and its environment, such as signals that flow into and out of the box. The *body* describes aspects of the digital system inside the black box that define how the outputs respond to the inputs.

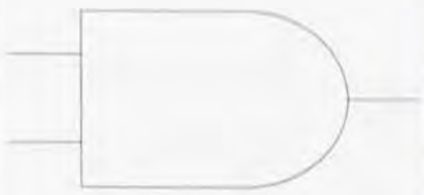
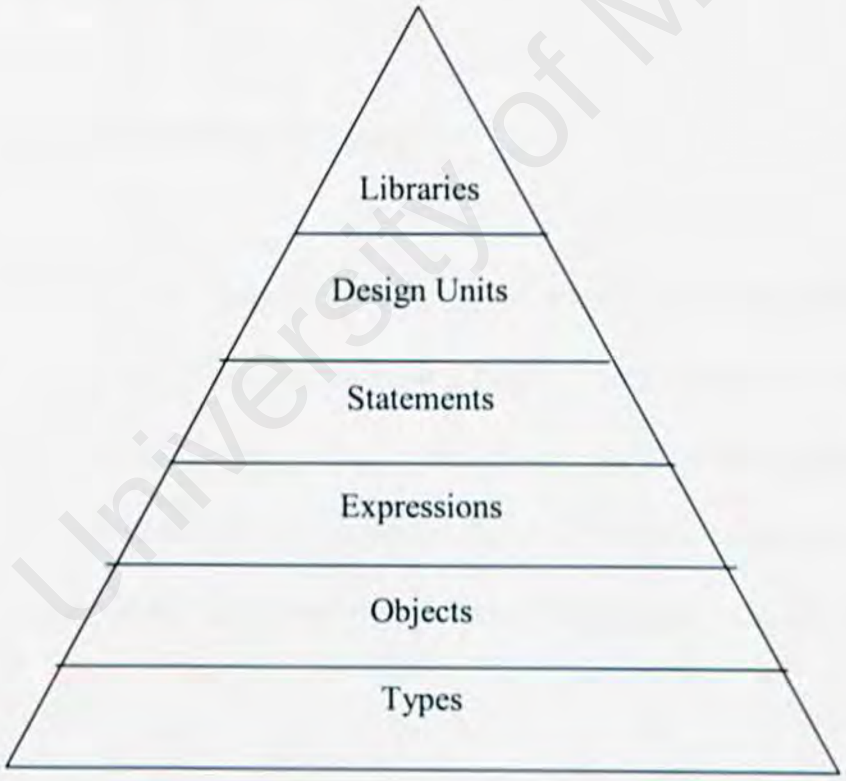
	`--Interface`
	`entity AND_OP is`
	`  --Input/Output ports`
	`  port`
	`    (A,B: in BIT;`
	`     Z  : out BIT);`
	`end AND_OP;`
	
	`--Body`
	`architecture EX_CONJUNCTION of AND_OP is`
	`begin`
	`  z <= A and B; --signal`
	`end EX_CONJUNCTION;`

Figure 2.2(a): Logic Symbol **Figure** 2.2(b): VHDL Description of a 2-input **and** operator

2.4.2 Language Organization of VHDL



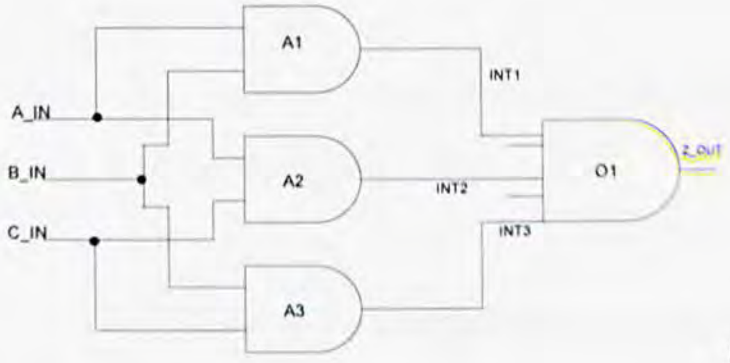
From the base of pyramid, predefined and user-defined types define data "templates" or sets. Objects, such as signals, hold values if the defined data types. Expressions combine operations with objects to yield new values, which are used by statements to describe aspects of digital hardware. Statements are contained within design units and design units are, in turn, contained within libraries.

The design units in VHDL are the following:

- Primary Design Units
    1. Entity Declaration
    2. Package Declaration
    3. Configuration Declaration
- Secondary Design Units
    1. Architectural Body
    2. Package Body

2.4.3 Structural Modeling in VHDL

Modeling logic schematic or netlists introduces a descriptive style called structural modeling. Structural modeling defines the behavior of a design entity by defining the components that comprise the design entity and their interconnection. A structural model implicitly or indirectly defines function because the design entity input/output transform can be derived knowing the constituent and their behaviors.



(a)

```
-- Interface
entity MAJORITY is
  --Input/output ports
  port
    (A_IN, B_IN, C_IN : in BIT;
     Z_OUT            : out BIT);
end MAJORITY;

--Body
architecture STRUCTURE of MAJORITY is
  --Declare logic operators
  component AND2_OP
    port (A, B : in BIT; Z : out BIT);
  end component;
  component OR3_OP
    port (A, B, C : in BIT; Z : out BIT);
  end component;

-- Declare signals to interconnect logic operators
signal INT1, INT2, INT3 : BIT;
begin
  --connect logic operators to describe schematic
  A1: AND2_OP port map (A_IN, B_IN, INT1);
  A2: AND2_OP port map (A_IN, C_IN, INT2);
  A3: AND2_OP port map (B_IN, C_IN, INT3);
  O1: OR3_OP port map (INT1, INT2, INT3, Z_OUT);
End STRUCTURE
```

Figure 2.3(a) and 2.3(b) show the logic schematic and VHDL description of majority function.

From the **Figure** 2.3, the declarative part of the architecture STRUCTURE contains three declarations: two component declarations and one signal declaration. The signal declaration declares three signals.

Component declarations start with the reserved keyword **component**, followed by the name of the component. The AND2_OP component has two input ports, A and B, and one output port, Z, all of type BIT. The OR3_OP component has three input ports, A, B, and C and one output port Z, all type BIT. Finally the keywords **end component** completes a component declaration.

We also need signals to interconnect the component instances. The input/output signals for the majority function, A_IN, B_IN, C_IN and Z_OUT are declared in the definition of the design interface. Port **map** clause defines the mapping between which signal are connected to which component ports, in other words, how a component is 'wired-up'

2.5 Comparison in Object-oriented C++ Technology and VHDL

2.5.1 Object-oriented design strategies

OO programming presents a number of powerful design strategies based on practical and proven software engineering techniques expressed by means of the corresponding object-oriented software programming language structures. They are fundamental and are encountered in different problem-solving contexts and without doubt in hardware design problem-solving context too.

Widely used standard hardware description languages like VHDL or Verilog do not support all of the variations of each of these strategies. Therefore the OO programming is not completely available with standard HDLs.

2.5.2 Abstraction and Separation.

Abstraction: A named, tangible representation of the attributes and behavior relevant to modeling a given entity for some particular purpose." According to the purpose of the design, a single entity may have many valid abstractions. The widely used abstraction in hardware design is a hardware component or a "primitive" expressed in the two domains:

(i) Structural domain: a component is described in terms of an interconnection of more primitive components.

(ii) Behavioral domain: a component is defining by its input/output response.

In VHDL, the primary entity is called a design entity; it corresponds to a module in SC. In VHDL only structural domain abstraction can be expressed. SC provides additional types of abstraction: communication abstractions can be expressed through the channels, and the ordinary C++ class concept can be used to express the behavioral abstractions. These additional abstractions are very important in H/S co design, allowing the designer to commit to a particular H/S partitioning late in the design process. In the pure hardware modeling these abstractions are also very important since they enable transaction based modeling which are very efficient regarding simulation performance. They are also important in inheriting specific behaviors and attributes when designing a library of components. CAD houses were forced to describe these libraries outside of VHDL, due to the lack of behavioral abstractions.

"**Separation**: the independent specification of an interface and one or more implementations of that interface."

In VHDL, a design entity consists of two different types of descriptions: the interface description and one or more architectural bodies. In hardware design the interface is presented by the description of entity's inputs and outputs. The architectural bodies can specify the behavior of the entity or a structural decomposition of the entity using more primitive components. In software engineering, an interface defines an external aspect that must be understood to use the software. This is a more general notion than just object inputs and outputs, it can abstract the actions that the object can use to interact with its environment, such as read, write actions for a bus object. The abstraction and separation concepts in the OO programming are expressed by means classes and objects. Each class has private and/or public members. Access from outside to private members is very limited allowing the notion of encapsulation.

"**Encapsulation**: can be defined as the restriction of access to data within an object to only those methods defined by the object's class".

The last software structure to which the abstraction and separation strategies are mapped is an object.

"**Object**: a distinct instance of a given class that encapsulates its implementation details and is structurally identical to all other instances of that class." Multiple instantiations of a given class can be made and each of them represents a distinct object. In VHDL, the notion of object corresponds to the one of a component instance that is used to create unique references to lower-level components. In SC, we may have a richer instantiation mechanism where an instance of an object can be an instance of a subclass of the declared type.

2.5.3 Composition

"**Composition**: an organized collection of components interacting to achieve a coherent, common behavior." There are two forms of composition:

(i) association

(ii) aggregation

In an aggregation, the whole is visible and therefore accessible. In association the interacting parts may be shared by different compositions, as they are externally visible. The association purpose is to allow objects to be connected to each other or to know about each other. The association method of building systems is known as the plug-and-play technique and is widely used in the validation and evaluation of the created system.

In hardware design the structural design decomposition is a basic modeling technique and that is why the composition is naturally presented by syntactical HDL structures. In VHDL the composition is presented, in our opinion, in the aggregation form, the design entity is visible as a whole by the others ones. The association form of the composition, which is missing in VHDL can be very useful in the high-level descriptions of the designed system, in test-bench generations, in design exploration and in application of the incremental refinement methodology to a design. The different associations of objects encapsulating some functionality can be validated or verified to find the optimal solution.

2.5.4 Generalization.

"**Generalization**: the identification, and possible organization, of common properties of abstractions." The generalization identifies commonalities among a set of entities. The commonality may be established in terms of attributes, behavior, or both. The generalization design strategy is directly connected to reusability. There are four forms of generalization expressed by object-oriented software structures:

(i) • Hierarchy;

(ii) • Genericity;

(iii) • Polymorphism;

(iv) • Patterns;

2.6 Using the Synopsis Logic Synthesis Tools with Nimbus

Nimbus [14] is a tool set that allows the capture of digital systems models as a collection of concurrent Algorithmic State Machine (ASM) models. Nimbus provides a design editor, compiler, cycle-based simulator, and model translator to both synthesizable VHDL and Verilog HDL. The technology is industry-proven, having been embodied in tools from another vendor for over a decade, and having been used by such companies as 3-Com, Alcatel, Sony, Hitachi and Ricoh. The use of this technology is as a design capture and architecture exploration medium, as well as a means to clearly communicate design intent.

The Design Analyzer is an older tool set, which primarily is used in logic synthesis and analysis of circuit results for ASIC applications. The FPGA Compiler (FC2) is used for the specific needs of synthesizing circuits on FPGA devices.

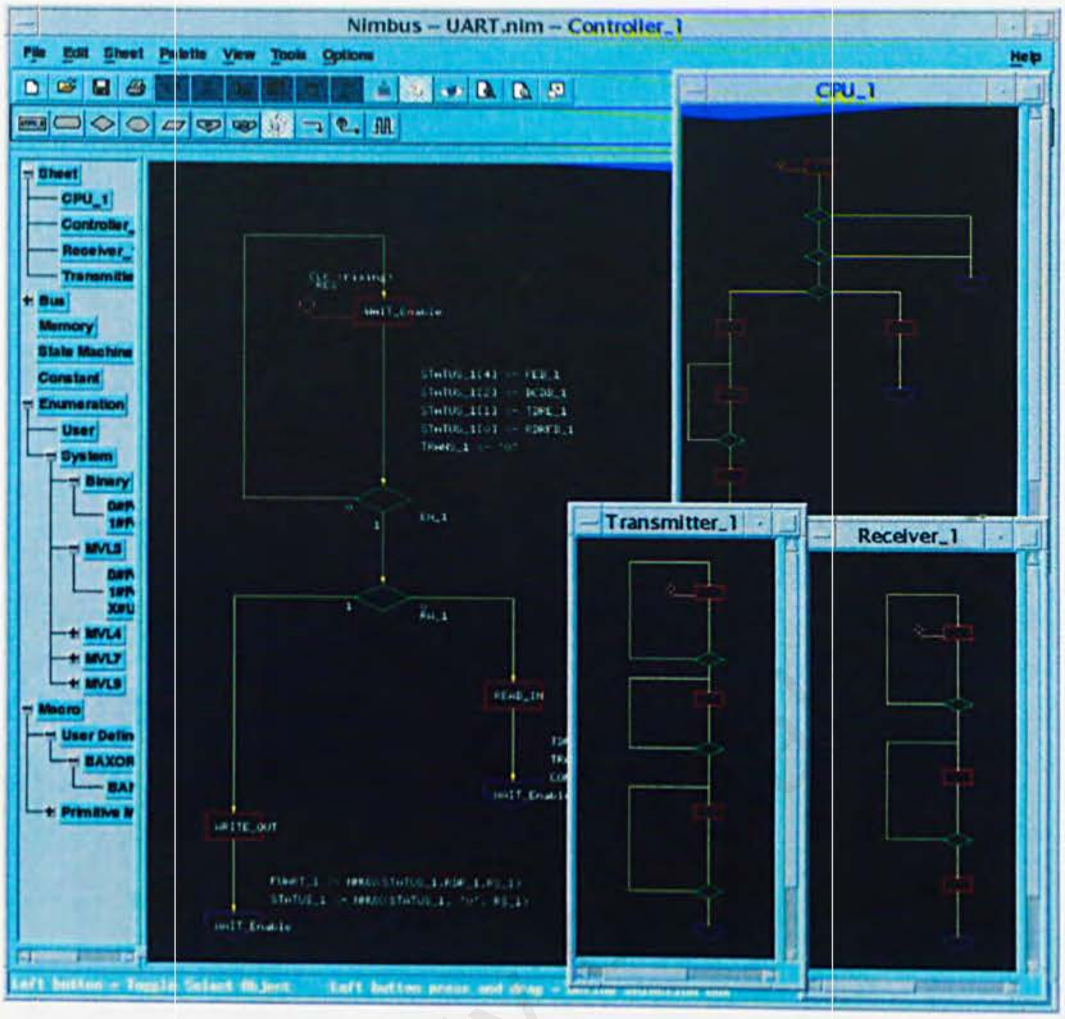
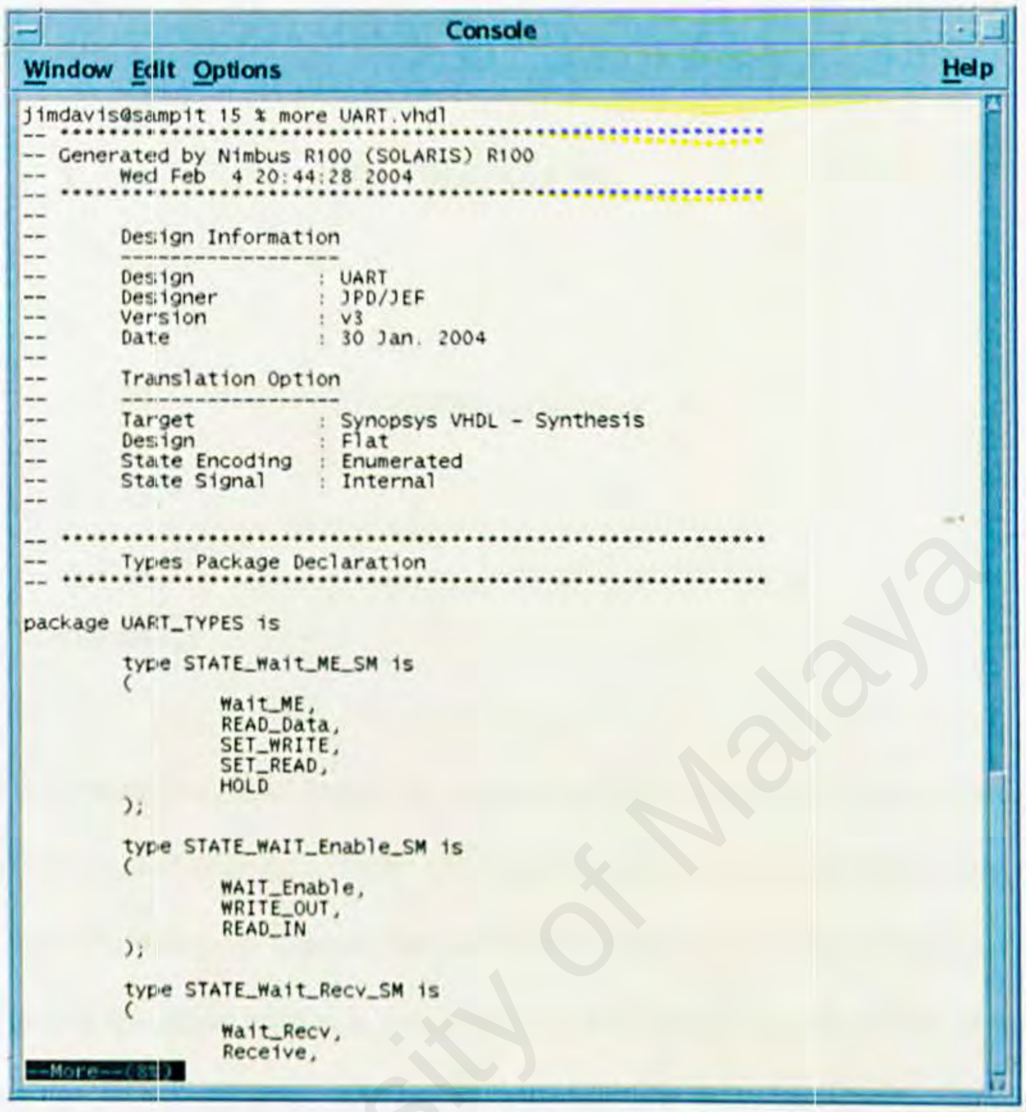


Figure 2.4: The Nimbus display of 3 threads of the UART model



```
jimdavis@sampit 15 % more UART.vhdl
-- ***********************************************************
-- Generated by Nimbus R100 (SOLARIS) R100
--      Wed Feb  4 20:44:28 2004
-- ***********************************************************
--
--      Design Information
--      ------------------
--      Design          : UART
--      Designer        : JPD/JEF
--      Version         : v3
--      Date            : 30 Jan. 2004
--
--      Translation Option
--      ------------------
--      Target          : Synopsys VHDL - Synthesis
--      Design          : Flat
--      State Encoding  : Enumerated
--      State Signal    : Internal
--

-- ***********************************************************
--      Types Package Declaration
-- ***********************************************************

package UART_TYPES is

        type STATE_Wait_ME_SM is
        (
                Wait_ME,
                READ_Data,
                SET_WRITE,
                SET_READ,
                HOLD
        );

        type STATE_WAIT_Enable_SM is
        (
                WAIT_Enable,
                WRITE_OUT,
                READ_IN
        );

        type STATE_Wait_Recv_SM is
        (
                Wait_Recv,
                Receive,
--More--(8%)
```

Figure 2.5:. Viewing the Generated VHDL Code in Text Window.

CHAPTER III

METHODOLOGY

3.1 Methodology

A methodology is a system of methods and principles used in a particular sub-discipline of software design. There are a large number of these, reflecting the way in which software design in practice has specialized. Those which are mature usually are supported by specialist tools and techniques. A traditional view of design in software engineering is analogous to building a cathedral: we make careful, comprehensive blueprints; use these as the reference point for coordinating architects and craftspeople; and ensure that the individual components connect precisely in the way described in our blueprints. We shall look at an example of this style of design: the Unified Process this, however, is not the only approach available. We also can adopt a less prescriptive, opportunistic style of design where a greater emphasis is placed on rapid revision of blueprints and the artifacts built from them.

In order to get the overview requirement of the Circuit Description and Elementary Hierarchical Circuit Simulation Using C++, an analysis to this topic is needed. The purpose of the analysis is to determine the entire functional requirement and also non-functional requirement for the tool.

3.2 Unified Process

The Unified Process is a software engineering process. It provides a disciplined approach to aasigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end-users, within a predictable schedule and budget. (Ali Bahrami, 1999)

The Unified Process is a traditional "cathedral" style of incremental design driven by constructing views of system architecture. It has the following key features:

(i) It is component based, commonly being used to coordinate object oriented programming projects.

(ii) It uses UML - a diagrammatic notation for object oriented design - for all for all blueprints.

(iii) The design process is anchored, and driven by, use-cases which help keep sight of the anticipated behaviors of the system.

(iv) It is architecture centric.

(v) Design is iterative and incremental - via a prescribed sequence of design phases within a cyclic process.

The Unified Process is a process product, developed and maintain by Rational Software. The Unified Process is a guide for how to effectively use the Unified Modelling Language (UML). UML is the tool that we use to represent (Model) the target software product. A major reason for using a graphical representation like UML is best expressed by the old proverb, a picture is worth a thouseand words. UML diagrams enable software professionals to communicate with one another more quickly and more accurately than if only verbal descriptions were used..

3.3 Iteration and Incrementation

In this project, I will use object-oriented modelling. The object-oriented paradigm is an iterative and incremental methodology. Within the Unified Process, each cycle contains four phases and each phase contains one or more iterations. Iterations is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development which grows incrementally from iteration to iteration to become the final system. This is a system that released with additional or improved functionality.

The advantages of the Iterative and Incremental Approach compare with the traditional waterfall process are:

(i) Risks re mitigated earlier.

(ii) Change is more manageable.

(iii) Higher level of reuse.

(iv) The project team can learn along the way.

(v) Better overall quality.

3.4 The Phases of the Unified Process

The software life cycle is broken into few cycles, each cycles working on a new generation of the product. Unified Process divides one development cycle in four consecutive phases. Those phases are:

(i) Inception phase

(ii) Elaboration phase

(iii) Construction phase

(iv) Transition phase

Each phase will end with the milestone as conclusion. A milestone means a point to make the critical decision about whether to continue development.

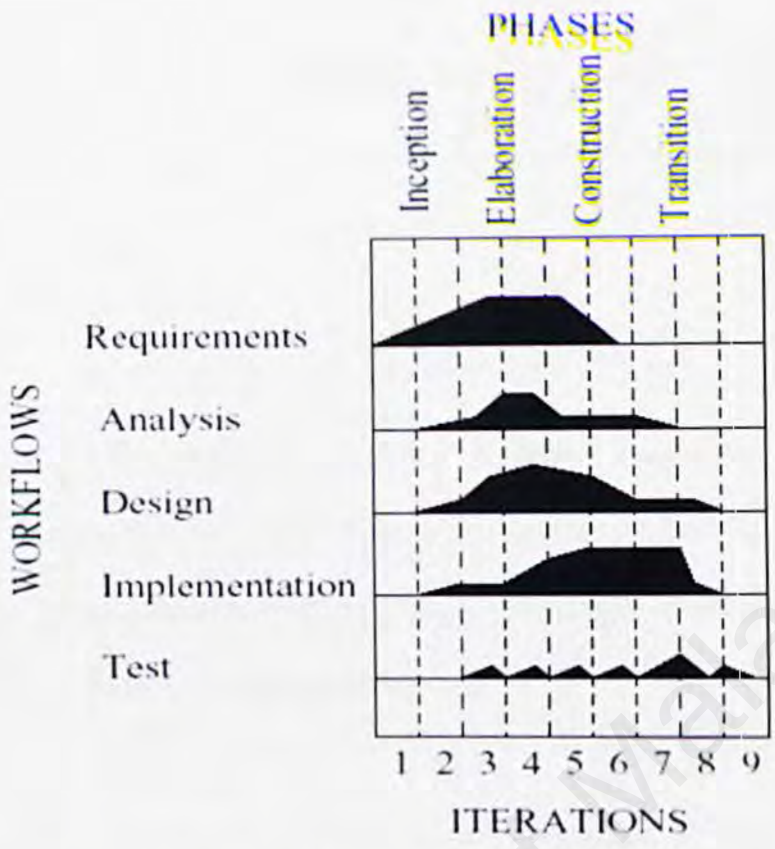


Figure 3.1: The core workflows and the phases of the Unified Process

3.4.1 The Inception Phase

Inception is the first phase for the life cycle, developers establish the business case for the system and delimit the poject scope during this phase. The aim of the inception phase is to determine whether it is worthwhile to develop the target software product. In other words, the primary aim of this phase is to determine whether the proposed software product is economically viable. The major milestone associated with the inception phase is called life-cycle objectives. The evaluation criteria for the inception phase are:

- Stakeholder agree on the scope of the system.
- Bussiness case for the system is strong enough to continue development.
- Sets of criticalhigh-level requirements are celarly addresses.

3.4.2 The Elaboration Phase

The aim of the elaboration phase is to refine the initial requirements, refine the architecture, monitor the risks and refine their priorities, refine the business case and poduce the software project management plan. The major activities of this phase are refinements or elaborations of the previous phase.

Basically the primary goal of the elaboration phase is to establish the ability to build the new system given the financial constraint, and other kinds of constraints that the development project faces. The elaboration phase also ensures the plan, requirement and architecture are stable enough and the risks are sufficiently mitigated.

The major milestone associated with this phase is called Life Cycle Architecture. At this point, developers examine the detailed system objectives and scope, the choice of architecture and the resolution of the major risks. The indications that the project has reached this milestone are:

- Most of the functional requirements for the systems have been captured with use case diagram.
- The project team has an initial project plan that describes how the construction phase will produce.

- The architecture baseline is a small, shinny system that will serve as a solid foundation for ongoing development.

3.4.3 Construction Phase

The purpose of this phase is to build system that is capable of operating successfully in beta customer environment. This means all remaining components and application features are developed and integrated into the product and all features are throughly tested during construction phase. The milestone for the construction phase is initial peration capability. The project has reached this milestone if a set of beta customers has more or less fully operational system in their hands.

3.4.4 Transition Phase

The aim of the transition phase is to ensure that the client's requirements have indeed been meet. This phase is driven by feedback from the sites at which the beta version has been installes. Faults in the software product are corrected. Also, all the manuals are completed. During this phase, it is important to try to discover any previously unidentified risks. The main objectives of the transition phase are:

- Achieving user self-supportability
- Achieving stakeholder concurrence thet deployment baselines are complete and consistent with the evaluation criteria of the vision
- Achieving final product baseline as rapidly and cost effectively as practical

The milestone associated with this phase is called product release. At this point, developers should decide if the objectives were met, or should start another development cylce.

3.5 Core Workflows

The Unified Process identifies core workflows that occur during the software development process. These workflows include Business Modeling, Requirements, Analysis, Design, Implementation and Test. The workflows are not sequential and likely will be worked on during all of the four phases. The workflows are described separately in the process for clarity but they do in fact run concurrently, interacting and using each other's artifacts.

3.5.1 Requirement Workflow

- Capture the functional requirement with case model.
- Describe what the system should do and allows the developers and the customer to agree on that description.

3.5.2 Analysis Workflow

- Aimed at building the analysis model to help the developer refine and structure the functional requirement captured.
- This workflow contains realizations of use cases that lend themselves better than the use cases to design and implementation work.

3.5.3 Design Workflow

- Aimed at building the design model to describe the physical realizations of the use cases
- Also focus on the deployment model, which defines physical organization of the system in terms of computational nodes.

3.5.4 Implementation Workflow

- Aimed at building the implementation model, which describes how the elements of the design model are packaged into software component.

3.5.5 Test Workflow

- Aimed at building the test model, which describes how integration and system tests will exercise executable components from the implementation model.

- Contains test case that are often derived directly from use cases.

3.6 Requirement Elicitation

During the analyzing process, many data and information are requiring to do analysis. The data congregation is an essential part in order to have a throughout understanding of the system to develop. There are few techniques are use to get useful data and information at the beginning of the project.

To define the requirement, the following techniques are used:

- Internet surfing
- Interview
- Library research

3.6.1 Internet Surfing

The internet is the largest information pool in the world. Whatever information requires also can get from the internet such as the information about circuit description and circuit simulation using object-oriented language and the advantages compared to other language.

3.6.2 Interview

At the beginning, I try to contact by e-mail some of the researcher that doing their research and journal writing on this topic. Luckily I get some respond from them and here I make some discussion and Q & A session regarding on circuit description and circuit simulation using C++ and why they did research on that.

3.6.3 Library Research

Collecting some journal paper, reference books and articles that related to the project topic. Studies are done on various issues, problems and solutions that are detailed in the journal paper.

CHAPTER IV

SYSTEM ANALYSIS

4.1 System Requirement Specification

System analysis is the one of the important phase, which focuses on understanding a system domain and the requirement. System analysis is conducted with the following objectives:

(i) Determine the functional and non-functional requirement for Circuit Description and Elementary Hierarchical Circuit Simulation Using C++.

(ii) To determine the tools that will be used.

(iii) To determine the programming language, software and hardware needs for Circuit Description and Elementary Hierarchical Circuit Simulation Using C++.

The system requirement need to drawn out to provide a guideline when developing a system. Therefore, the requirement analysis needs to cover the area of the functional requirement and non-functional requirement of Circuit Description and Elementary Hierarchical Circuit Simulation Using C++.

4.2 Functional Requirements

Functional requirement is a function or feature that must be included in an information system to satisfy business need and acceptable to the user. Functional requirement describe an interaction between system and the environment. In this project, the functional requirement is circuit component and the class that will be used in the circuit description.

4.2.1 Classification of Circuit Components

For the purpose of this report, objects which are at the digital level and above will only be considered. Therefore, switch level devices such as transistors will not be considered.

One possible way to classify these numerous objects is to consider all the circuit entities at their highest level of abstraction and attempt to group objects which have similar properties under the same base class. From this, three very general classes are formed:

- `Component` -- All elements derived from this class process input signals and generate output signals to the objects to which they are connected. It is possible for one component to be part of another component. Circuit elements which can be considered as kind of components would include AND gates, RS-latches and random functional blocks.
- `Connector` -- All elements derived from this class would be responsible for connecting components with other components or with the external world. Each connector is part of a component at some level of abstraction. Since a connector can ``feed'' one or more components via fan-out, a list of components can be considered part of a connector. Some circuit elements which are kind of connectors include wires, and I/O ports.
- `Signals` -- Objects instantiated from this class are passed from component to component via the connectors. A list of signals also can be considered as part of a wire. This report will consider signals as two entities: a signal value (such as HIGH, LOW or X) and an associated unit of time.

As alluded to above, linked list classes are required for components and signals. The need will also arise for a linked list class for I/O ports. Due to the current lack of parameterized types in the C++ language, some duplication of code is necessary to create the three linked list classes. Fortunately, the replication of code is relatively small.

The `Component` Class

Components are responsible for getting inputs, processing them and producing outputs. In order for components to get their inputs from and send their outputs to the external world, ports must be made part of the components in some way. To increase the ease at which the simulation algorithm can access the component's ports, pointers to both the input and output ports will be stored in separate linked lists within the components. Therefore, there are at least two elements of the `Component` class: two linked lists of input and output port pointers. Since ports form the interface of a component, these linked lists are placed in the public section of the class. However, the operations that can be performed on this linked list are limited by the public interface of the `Port_List` class.

Next, the user should be given the option of assigning some name to a component, which will be useful from a debugging viewpoint. Therefore, a pointer to a character (string) will be placed in the class. In addition, in this particular implementation, every component maintains its own *local time* during the simulation. Since nothing else outside of the class should be allowed to access the name or local time, these two data members should be placed in the private section of the class so they can be accessed only by methods of this class, such as the constructor.

Every component also has a delay which represents how long it takes to produce its outputs upon receiving its inputs. During the simulation, objects derived from `Component` should be permitted to change the delay time of the component (for example, the delay time could increase during the simulation, modelling the effect of a component getting warmer, hence increasing its resistance). Since classes derived from `Component` are the only ones able to change the delay, the `delay` data member is protected in the class.

Two public methods are declared, which are used extensively during the simulation of the circuit. These two methods are `process()` and `simulate()`.

Finally, a constructor is required which is used to actually build the component. Since a component is too abstract a concept to be useful from an instantiation perspective, only specialized classes derived from Component may call this constructor. Therefore the Component constructor is made a protected member of the class.

The `Component_List` Class

The `Component_List` class, maintains a linked list of components which may be present in the fan-out of a connector.

The `Connector` Class

The `Connector` class is responsible for connecting components together and for connecting components with the outside world. In circuit description, there are two type of connectors, wires and ports. The `Connector` class will be used as an abstract base class from which a `Wire` class and a `Port` class will be derived.

The `Wire` Class

The `Wire` class is derived from the `Connector` class. Wires connect components together and also maintain a history of signals which have travelled through them during the course of the simulation. Connections to components are achieved through the `fan_out` data member as inherited from the `Connector` class. However, in order for the class to keep track of all the signals that have passed through it, it must maintain a linked list of signals. Hence a new class, `Signal_List` is created for this purpose. The `Wire` class defines two constructors. The first one accepts an optional name and simply passes that name up to its base class, `Connector`, for initialization. The linked list of signals is then initialized and a single signal is added to the wire to represent its initial value.

The `Port` Class

Ports provide a means whereby components are connected with the external world. It is through ports that components send and receive signals. In addition to maintaining a fan-out of the components that it feeds (which is inherited from `Connector`), each port must also maintain a pointer to the connector that ``feeds" it. Note that a port can be fed by one, and only one, `Connector`. This means that a port may be fed by a single wire or by a single port; and not by, for example, two wires. Note, however, that a wire may feed one or more distinct ports. To keep track of the connector that feeds it, the `Port` class maintains a pointer to the specific `Connector`.

Since ports are somewhat too generalized, an `Input` and `Output` class will be derived from `Port`. To prevent the programmer from accidentally creating a `Port` object, the constructor will be kept protected and is therefore usable only by the `Input` and `Output` classes.

The `Input` and `Output` Class

The `Input` and `Output` classes are almost the same with just one minor difference.

The `Port_List` Class

The `Component` class contains a `Port_List` (a linked list of pointers to ports) in the public section of its class. The `Port_List` class is almost identical to the `Component_List` class. This similarity could be exploited using generic types or templates. Since C++ does not yet support templates, some code replication will be necessary.

4.3 Non-Functional Requirement

A non-functional requirement or constraint describes a restriction on the system that limits on choice for constructing a solution to the problem.

4.3.1 Correctness

Correctness is the extent to which program satisfies its specification and fullfill user's requirement and objective.

4.3.2 Reliability

The system will be developed in a way that is reliable and will not cause any unnecessary failure at the overall operation. System will not cause any technical or costly failure when it is used in reasonable manner. Any information display will be risk-free.

4.3.3 Response Time

The data retriever time should be considered with on a reasonable interval time. All the desired information should be availabe to user at any point in time. The user should not be asked to tolerate with slow response time.

4.3.4 Expandability

Expandability measures the capability of a system to be upgraded or enhanced in future. It is important when an existing system needs enhancement to overcome changes in environment and requirement. Expandability of a system also determines whether the system can be integrated with sub-system to increase its functionality.

4.4 Consideration of Programming Language

To implement circuit description and simulation using object-oriented language, I will use Microsoft Visual C++ 6.0. This is because this type of version are widely used in university and other educational institution.

Microsoft's *Visual C++ 6.0* (VC++ 6.0) lets programmers unlock the power of Office and Internet Explorer and create custom Office and Windows apps. Every version of Microsoft Office and Internet Explorer has powerful custom features-dockable toolbars, tool tips, OLE automation, and ActiveX, for example. But to really put these features to work you need a full-fledged programming language. Features provided by VC++ 6.0 are:

- Fully integrated editor, compiler, and debugger
- Ability to create complex software systems

The list of updates in version 6.0 is lengthy, but two stand above the rest. This rendition of Microsoft's Visual C++ gets smart with IntelliSense technology, Microsoft lingo for auto-completion. What this means for you is after you type a period after a variable name, a handy drop-down menu appears offering the *soup du jour* in the way of the available members for the aforementioned object. Enter a method name and an open parenthesis and you are presented with prototypes, arguments and their types. Better yet, IntelliSense works with all the expected iterations and your code, saving quite a lot of time. Also, the intuitive edit and continue feature allows you to incorporate common, simple edits during your debugging without having to quit, rebuild and restart the debugger.

Other new features include the HTML Help Workshop, a tool for creating HTML-based context-sensitive help that can be integrated with the Web; a gallery of prepackaged C++ components and ActiveX controls; and a slew of inline optimization switches, programs and codes.

4.5 Hardware Requirement

Specification	Minimum Spec.
Required Operating System	Microsoft Windows 95/98, Microsoft Windows NT 4.0 or later
Required Memory	24MB
Required Disk Space	290MB
Required Processor Class	Intel Pentium
Required Processor Speed	90 MHz
Other Requirement	Mouse or compatible device, CD-ROM

CHAPTER V

SYSTEM DESIGN

5.1 What is Systems Design

Information system design is define as those tasks that focus on a specification of a detailed computer based solution. It is also called physical design. System designs focuses on the technical or implementation concerns of the system.

Object-oriented design (OOD) is the newest design strategy. The aim of OOD is to design the product in terms of objects, that is instantiations of the classes and subclasses extracted during object-oriented analysis.

5.2 The Method of Designing

The technique used to describe hardware can be outlined in six major steps:

1. **Identify the internal circuit elements of the component**. After the end of this step, one should have *I* input ports, *O* output ports, *S* subcomponents and *W* wires.

2. **Create a class for the component**. Make sure that this class is derived from `Component` so that it will inherit all the features of this base class. The parameters passed to the constructor of the class should include *I* + *O* references to connector objects, a parameter for the delay of the component and a parameter for the name of the component. The constructor should be in the public part of the class. The elements identified above should be encapsulated within the class. Keeping them in the private portion of the class prevents the relationships amongst the wires, subcomponents and ports from becoming corrupt by something from outside the class.

3. **Connect primary inputs and outputs**. When defining the constructor, calls are made to the *I* + *O* port constructors. This expresses the connectivity between the ports of the component and the connectors of the external environment. This will have the effect of connecting the I/O ports with the primary inputs and outputs of the circuit. Calling the I/O port constructors also has the effect of placing the ports in their respective linked lists. This is hidden from the user. It is often useful to disguise the call to the port constructors by using a CONNECT macro (CONNECT (Port, Wire, "Name").

4. **Call the *W* constructors for the wires.** This will bring the wires into existence and place an initial value on each wire.

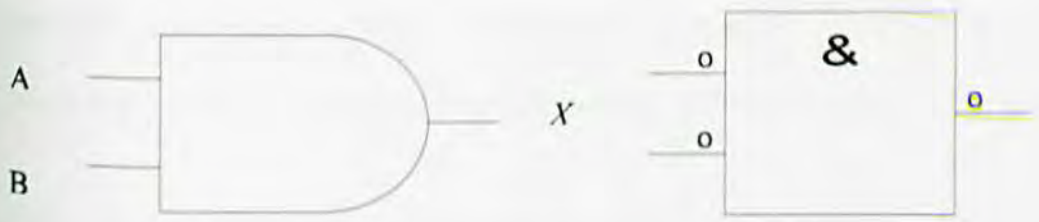
5. **Construct each of the *S* subcomponents.** One important point to remember when constructing the subcomponents is to pass only the wires and ports that are declared within the component class to the subcomponent constructors. If the external connectors passed to the encompassing component constructor are passed to the encapsulated subcomponents, then the designer may risk corrupting the description.

6. **Create *I* + *O* external wires, and instantiate the component.** This is usually done in the `main()` program of the C++ code. The external wires act as signal sources when hooked up to an input port. When connected to an output port, they act as signal destinations.

5.3 Hardware Description Component

In this project, I will concentrate on three type of hardware component. The hardware component are two input AND gate, Three Input AND Gate, RS-Latch and Full-Adder..

5.3.1 Two Input AND Gate



(a) Distinctive shape (b) Rectangular outline with the AND(&) qualifying symbol

Figure 5.1: Standard logic symbols for the AND gate with two input

The **Figure** 5.1 shown standard logic symbols for the AND gate with two input. The lines connected to each symbol are the inputs and outputs. The inputs are on left of each symbol and the output is on the right. A circuit that performs a specific logic operation AND is called a logic gate. AND gates can have any number of inputs.

An AND gate produces a HIGH output only when all of the inputs are HIGH. When any of the input is LOW, the output is LOW. Therefore, the basic purpose of an AND gate is to determine when certain conditions are simultaneously true, as indicated by HIGH levels on all of its inputs and to produce a HIGH on its output to indicate that all these conditions are true. The gate operation can be stated as follows:

For a 2-input AND gate, output *X* is HIGH if inputs A and B are HGH; *X* is LOW if either A or B is LOW, or if both A and B are LOW.

Since a two-input AND gate is at the lowest level of abstraction, the only encapsulated circuit elements will be two input ports and a single output port. When a two-input AND gate is actually instantiated and connected to primary input and output wires in the main C++ program, the data structure shown in **Figure** 5.2 is produced.

A two-input AND class may be declared as follows:

```
class And2 : public Component
{
public:
        And2(Connector &, Connector &, Connector &,
             ckt_time = 1L, char* = "And2");
        void     process(ckt_time);
private:
        Input   I1, I2;
        Output  O1;
};


And2::And2(Connector &ci1, Connector &ci2, Connector &co1,
           ckt_time dly, char *name) :
        Component(dly, name),
        CONNECT(I1, ci1, "And2 I1"),
        CONNECT(I2, ci2, "And2 I2"),
        CONNECT(O1, co1, "And2 O1")
{ }
```

The constructor is declared as taking three references to connector objects as parameters and the three ports are hidden in the private section of the class. The delay and the name of the component are first passed to the `Component` base class constructor for initialization. The ports of the class are then connected with its primary inputs and outputs.

Since the AND gate has no nested wires and no nested components, the description of the AND gate is finished. Its functionality is specified by redefining the virtual `process()` method.

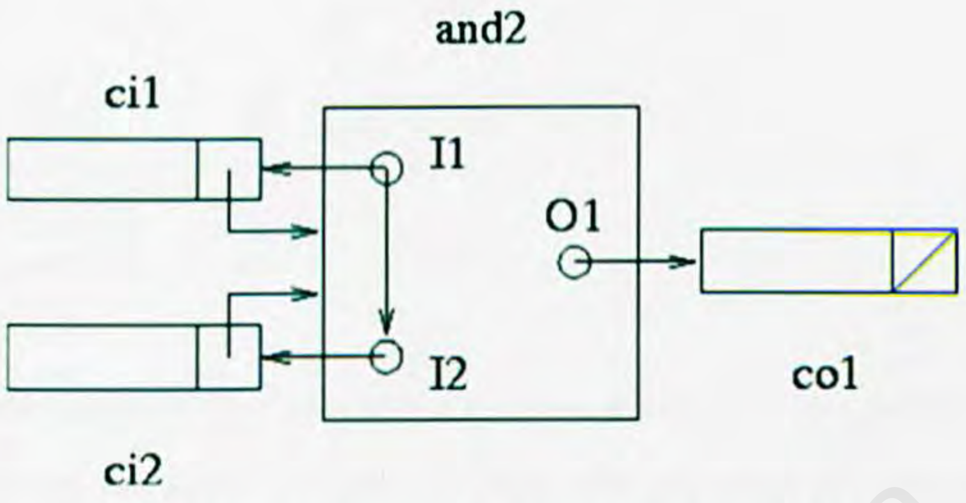


Figure 5.2: Two-Input AND Gate with External Wires

5.3.2 Three-Input AND Gate

Next, a three-input AND gate will be built using two two-input AND gates, which were created previously, and a single wire that will connect the output of the first AND gate with the input of the second gate. Summarizing the fundamental components: there are three input ports, one output port, one wire and two-input AND gates.

The class declaration follows:

```
class And3 : public Component
{
public:
        And3(Connector &, Connector &,
             Connector &, Connector &,
             ckt_time = UNDEF_TIME, char* = "And3");
private:
        Input   I1, I2, I3;
        Output  O1;
        Wire    w;
        And2    and2a, and2b;
};
```

```
And3::And3(Connector &ci1, Connector &ci2,
         Connector &ci3, Connector &co1,
         ckt_time dly, char *name) :
      Component(dly, name),
      CONNECT(I1, ci1, "And3 I1"),
      CONNECT(I2, ci2, "And3 I2"),
      CONNECT(I3, ci3, "And3 I3"),
      CONNECT(O1, co1, "And3 O1"),
      w("And3 wire"),
      and2a(I1, I2, w, 1L, "and2a"),
      and2b(w, I3, O1, 1L, "and2b")
{ }
```

The constructor now takes four connector objects since a three input AND gate has a total of four ports. As usual, the ports, wire and two-input AND gates are encapsulated in the private section of the class. The delay time of the three-input AND gate is defaulted to an undefined time because the delay of the gate actually depends upon the delay associated with the two two-input AND gates.

To define the constructor of the three-input AND gate, all the connector objects are connected with the ports accordingly; the wire is instantiated and the two two-input AND gates are constructed. The two inputs of the first two-input AND gate come from the first two input ports of the three-input AND gate, while its output is connected to the nested wire. The second two-input AND gate receives its first input from this wire and gets its second input from the third input port of the three-input AND gate. The output of this two-input AND gate is directed to the output port of the three-input AND gate.

Figure 5.3 shows how all the elements of the circuit are connected when a three-input AND gate is instantiated using external wires. For clarity, the linked list pointers which link the three input ports together have been omitted.

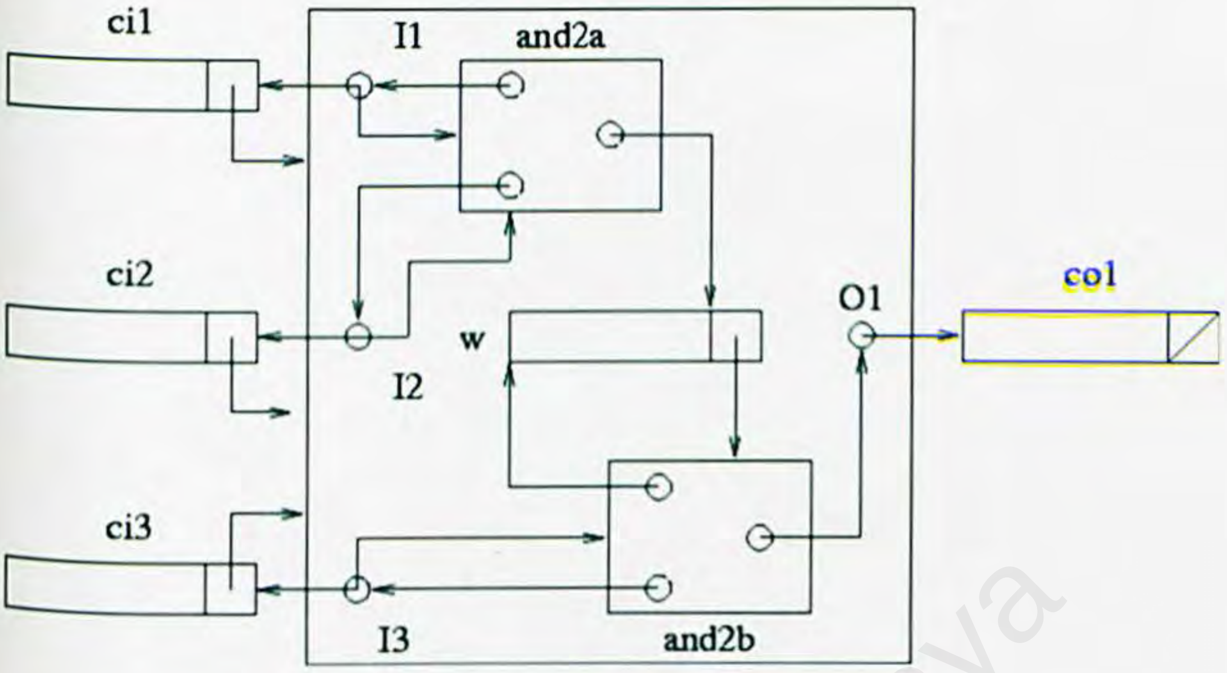


Figure 5.3: Three-Input AND Gate with External Wires

5.3.3 S-R (SET-RESET) Latch

A latch is a type of bistable logic device or multivibrator. An active-LOW input S'-R' latch is formed with two cross-coupled NAND gates as shown in **Figure** 5.4. The output of each gates is connected to an input of the opposite gate. This produces the regenerative feedback that is characteristic of all latches and flip-flops.

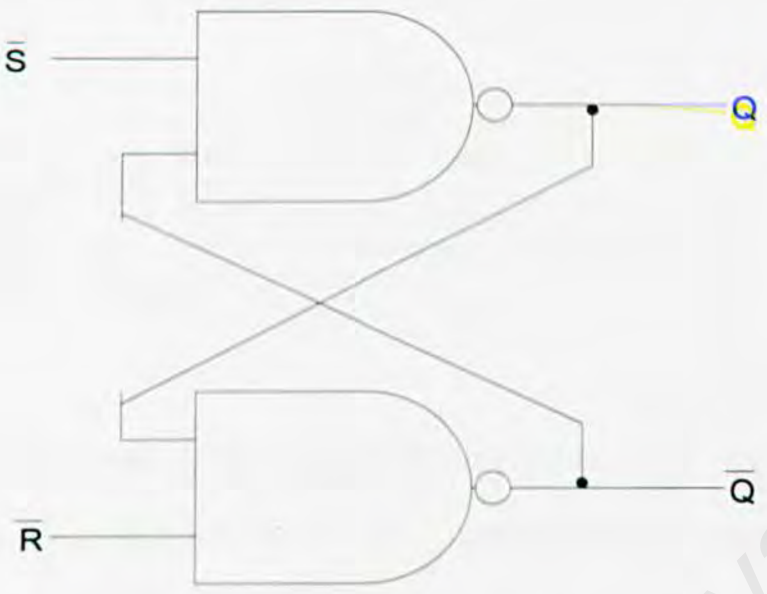


Figure 5.4: Active-LOW input S'-R' latch

An RS-latch has two input ports, two output ports and two two-input NAND gates which can be described in a manner identical to the description of the two-input AND gate. Note that an RS-latch does *not* have two wires embedded within it. The feedback mechanism means that the two output ports, `Q` and `Qb`, also act as input ports to the two NAND gates. Therefore there are no wires created by the RS-latch.

The four ports and the two NAND gates are encapsulated in the RS-latch class as shown in the following class declaration:

```
class RS_Latch : public Component
{
public:
        RS_Latch(Connector &, Connector &,
                 Connector &, Connector &,
                 ckt_time = UNDEF_TIME, char* = "RS_Latch");
private:
        Input   S, R;
        Output  Q, Qb;
        Nand2   nand2a, nand2b;
};

RS_Latch::RS_Latch(Connector &ci1, Connector &ci2,
                   Connector &co1, Connector &co2,
                   ckt_time dly, char *name) :
        Component(dly, name),
        CONNECT(S, ci1, "S"), CONNECT(R, ci2, "R"),
        CONNECT(Q, co1, "Q"), CONNECT(Qb, co2, "Qb"),
        nand2a(S, Qb, Q, 1L, "nand2a"),
        nand2b(Q, R, Qb, 1L, "nand2b")
{ }
```

Next, the constructor of the class is defined. The four ports are connected with the four connectors passed into the constructor. The first NAND gate gets its first input from the `S` input port and its second input from the `Qb` output port. It sends its output to the `Q` output port. The second NAND gate gets its two inputs from the `Q` output port and the `R` input port of the RS-latch. Its output is sent to the `Qb` output port.

A diagram showing the interconnectivity of the components within the RS-latch is presented in **Figure** 5.5. The pointers connecting the input and output ports are omitted for clarity.

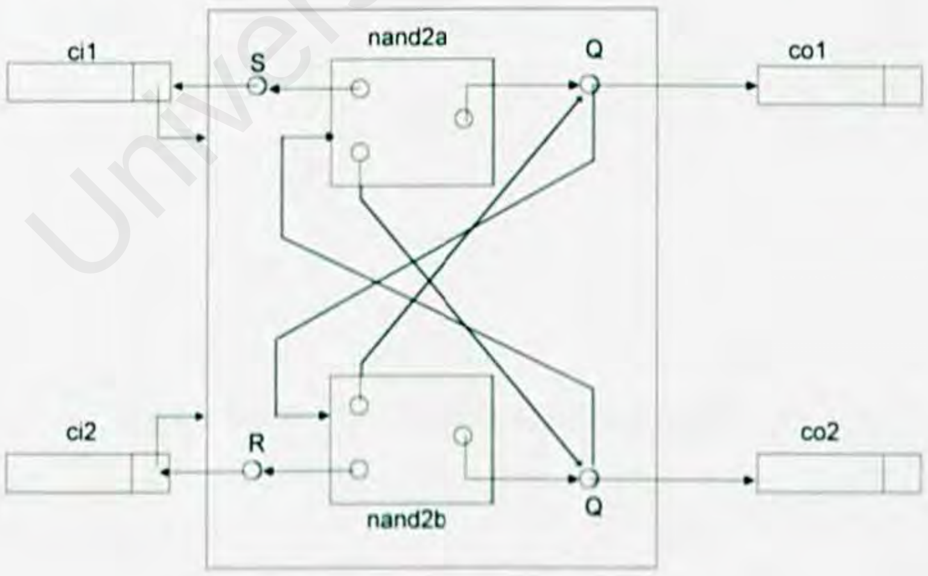


Figure 5.5: RS-Latch with External Wires

5.3.4 Full-Adder

Adders are important not only in computers but also in many types of digital systems in which numerical data are processed. An understanding of the basic adder operation is fundamental to the study of digital systems.

The full-adder accepts two input bits and an input carry and generates a sum output and an output carry. A logic diagram for *sum* of full-adder is shown in figure 5.6 (a) and *carry* in figure 5.6 (b).

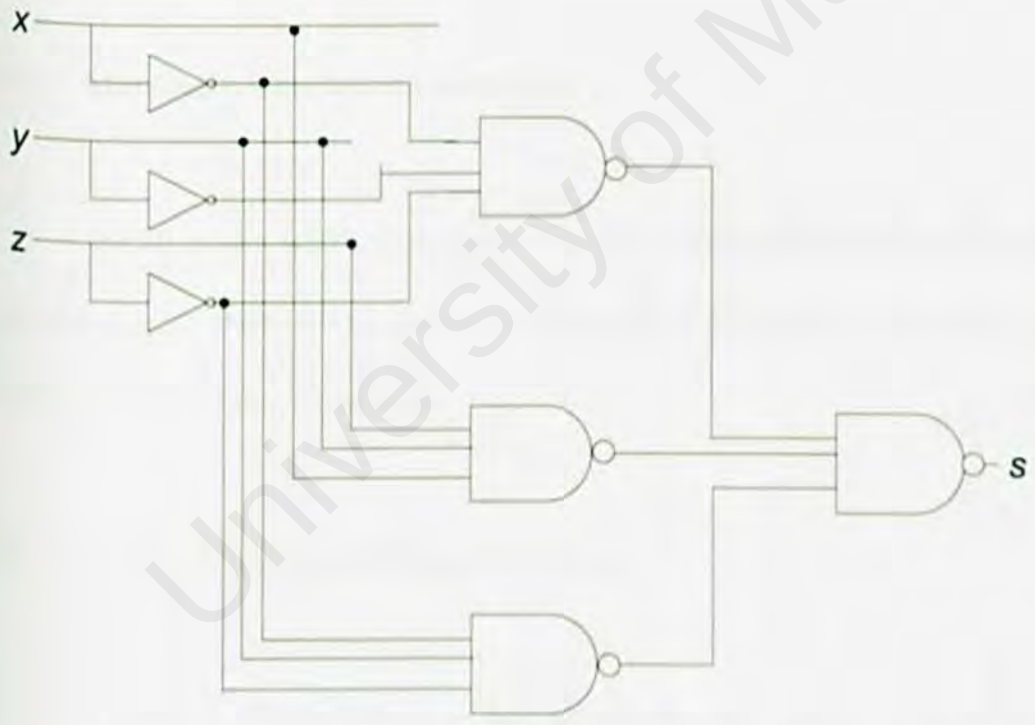


Figure 5.6(a): Full-Adder Logic Diagram for *Sum*

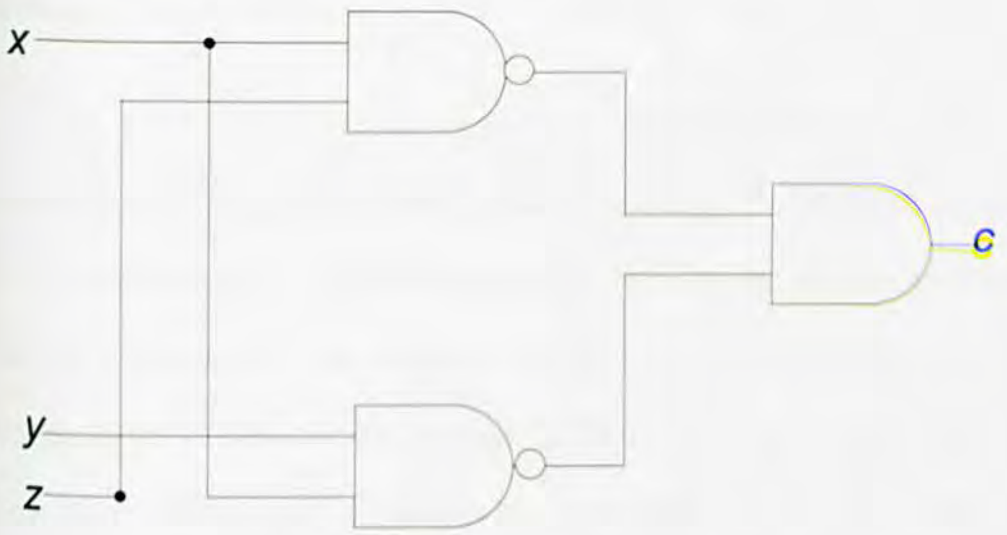


Figure 5.6(b): Full-Adder Logic Diagram for *Carry*

5.4 Hardware Simulation Using C++

An algorithm which implements the new simulation technique will be examined and the `signal` class will be analyzed. This method of simulation has also been used in various forms in other simulators.

5.4.1 Circuit Simulation: Time and Queues

Central to the foundation of any simulator is the concept of time, and how the elements which make up a component move through it. Here we will look at methods of treating time and how that concept of time is manifested in the implementation.

Distributed Event Queues

Here I will describe a new approach to hardware simulation which challenges the commonly used technique described above. Basically, the approach is to encapsulate one or more queues within the components themselves, thereby eliminating all the inherent problems of maintaining a global structure. The queues are distributed throughout the component and can exist at virtually any level of the description. Each queue keeps a history of the signals which have been sent to it during the course of the simulation and maintains a list of all the components which expect the signal. Therefore, the distributed queue serves as a connector between two components and also serves as the means by which signals may be propagated in parallel using a sequential programming language.

With respect to hardware, the distributed event queues represent wires; signals travel along wires and wires connect components together. Since the queues are encapsulated within the circuit components, an asynchronous signal would only affect those components which receive it and would not force the entire circuit back in time. Only those elements who use the asynchronous signal directly or indirectly will actually be moved backwards. The other components would continue to move forward in time where they left off.

5.4.2 Signals and Signal Transmission

Every signal is composed of two elements: a signal value (for example, HIGH, LOW, X) and a time when that value was produced. They are stored in linked lists in much the same way that the `Component_List` class stores components.

The `Signal` class is declared as follows:

```
class Signal
{
friend ostream &operator <<(ostream &, const Signal &);
public:
        Signal(Sig_Val = X, ckt_time = INIT_TIME);
        operator        Sig_Val();
        ckt_time        get_time();
private:
        Sig_Val         value;
        ckt_time        t;
};
```

```
Signal::Signal(Sig_Val sv, ckt_time ct) :
        value(sv), t(ct)
{ }
```

The `operator Sig_Val()` function is a special function called a *user-defined conversion*. It returns the `value` field whenever a signal object is used in the context where a `Sig_Val` is expected. This function essentially converts a signal into a signal value.

```
Signal::operator Sig_Val()
{
        return value;
}
```

The `get_time()` method is simply an access method which returns the time that the signal occurred.

```
ckt_time
Signal::get_time()
```

```
{
        return t;
}
```

The `ostream &operator <<(ostream &, const Signal &)` function enables the signal's value and time to be output using the `<<` operator. This function is made a friend of the class so it has access to the hidden members of the class. Overloading this operator enables signals to be treated just like other built-in types which are output using the same technique.

The `Signal_List` class is declared as follows:

```
class Signal_List
{
public:
        Signal_List();
        void            add(Signal);
        Signal          find(ckt_time);
        void            dump();
private:
        Signal_Node     *sig_list;
};
```

The constructor and the `add()` method are identical to the corresponding methods in the `Component_List` class. The `add()` function also checks to make sure that signals enter the list in the correct time order. Should a signal be found whose time is less than or equal to the last signal on the wire, a warning message is displayed.

The `find()` method simply scans the nodes of the signal list searching for the signal which occurred at the specified time. If a signal could not be found, an undefined signal is returned.

```
Signal
```

```
Signal_List::find(ckt_time t)
{
        // Make sure we are not looking too far in the future.
        if (sig_list->sig.get_time() < t)
                return Signal(UNDEF_SIG, UNDEF_TIME);
        for (Signal_Node *list = sig_list; list != 0; list =
             list->next)
        {
                if (list->sig.get_time() == t)
                        return list->sig;
                if (list->next != 0 && list->next-
                    >sig.get_time() <= t)
                        return list->next->sig;
        }

        // If signal not found, return an error signal.
        return Signal(UNDEF_SIG, UNDEF_TIME);
}
```

The `display()` method simply traverses the signal list, displaying each signal it encounters using the overloaded `<<` operator.

```
void
Signal_List::dump()
{
        for (Signal_Node *list = sig_list; list != 0; list =
             list->next)
        {
                cout << list->sig;
        }
}
```

Signal transmission occurs using the `get_Signal()` and `send_Signal()` methods which are defined as virtual methods in the `Wire` and `Port` class. These two functions were defined virtually because the method that a `Port` uses to get and send a signal is quite different from the method used by a `Wire`. A wire gets a signal by sending a `find()` message to its encapsulated signal list. A Port gets a signal by sending a `get_Signal()` message to the connector which feeds it. Hence, the `get_Signal()` message for a wire and a port is as follows:

```
Signal
```

```
Wire::get_Signal(ckt_time t)
{
        return signals.find(t);
}

Signal
Port::get_Signal(ckt_time t)
{
        return external->get_Signal(t);
}
```

The `send_Signal()` command operates in a similar fashion except that in both the wire and port class, the signal is propagated to all the components in the fan-out of the wire/port.

```
void
Wire::send_Signal(Signal s)
{
        signals.add(s);
        fan_out.propagate();
}

void
Port::send_Signal(Signal s)
{
        external->send_Signal(s);
        fan_out.propagate();
}
```

CHAPTER VI

SYSTEM IMPLEMENTATION AND CODING

6.1 Introduction

System implementation is a process that converts the system requirements and design into program codes. This phase at a time involves some modifications to the previous design. Techniques and approaches used to build the system will be described in more details manner. Basically, the implementation will try to match the design as much as possible. If time allowed, then there will be some enhancement in certain area that necessary.

6.2 Program Coding

In this stage, the programs are written using programming language. In this project, I'm using Microsoft Visual C++ 6.0. The components built during development are put into operational use. The system is built according to the original design that was done.

6.2.1 Coding Style

Coding style is an important attribute of source code and it determines the intelligibility of a program. An easy to read source code makes the system easier to maintain and enhance. The element of coding style includes internal (source code level) documentation, method of data declaration and approach to statement construction.

6.2.2 Code Documentation

Code documentation begin with the selection of identifier (variable and variable names), continues with the composition of connectivity and end with the organization program.

6.2.3 Internal documentation

Internal comment provides a clear guide during the maintenance phase of the system. Comments provide the development with means of communicating with readers of the source code. Statement of purpose indicating the function of the module and a descriptive comment that embedded within the body of the source code is needed to describe processing function.

6.2.4 Naming Convention

A good and meaningful naming technique for the variables, controls and modules provides easy identification for the programmer. The naming convention is created with coding consistency and standardization in mind.

6.2.5 Modularity

Before entering the coding phase, the project has been divided into several modules. The main purpose of modularity is to reduce the complexity of the system. In order to reduce complexity and facilitate changes that result in easier implementation by encouraging parallel development of different parts of the system.

6.2.6 Readability

Codes should be easy to understand. Adherence to coding conventions such as naming conventions and indentation contribute to program readability.

6.2.7 Robustness

The codes should be able to handle cases for user error by responding appropriately. It should be able to avoid any abrupt termination or system failure.

6.2.8 Maintainability

Codes should be easily revised or corrected. To facilitate maintenance, code should be readable, modular and as general as possible.

6.3 Implementation of the Simulation Algorithm

All components are simulated in the same manner. In terms of pseudo-code, this algorithm could be summarized as follows:

```
function simulate (component)
while (inputs to component are ready) do
    increment local time of component
    if (component has no subcomponents) then
        call virtual process function
    else for each (input port of component) do
        for each (element in fan_out of port) do
            execute simulate function for fan_out
            component
```

In the `Component` class, the `simulate()` method is defined as follows:

```
void Component::simulate()
{
        while (I_List.inputs_are_ready(local_time) == TRUE)
        {
            local_time++;
            if (I_List.is_lowest_level() == TRUE)
            {
                    if (delay <= 0)
                    {
                            cerr << "Error: invalid
                                     delay\n";
                            exit(1);
                    }
                    process(local_time - 1);
            }
            else
            {
                    I_List.descend();
            }
    }
}
```

The `simulate()` message is sent to a component that the user wishes to simulate. In high-level terms, the method continues to execute until at least one of the distributed queues (wires) which feeds the component runs out of signals to supply to it. While there are inputs available, the component advances forward in time one unit and then determines its level of abstraction. If it is at the lowest level, that is, this component is not made up of any subcomponents; then the component's delay time is validated. If the delay is less than or equal to zero, an error message is produced and the program terminates, otherwise the `process()` method is executed. This method is responsible for reading inputs from the incoming distributed queues via the input ports, processing them and then placing the results on the outgoing distributed queues via the output ports. If the component is not at the lowest level of abstraction, then the code descends a level and sends the same `simulate()` message to all the subcomponents. This method of recursively descending a hierarchical circuit description has been done with other object-oriented simulators as well.

The three methods inputs_are_ready(), is_lowest_level(), and descend() are all members of the Port_List class. The inputs_are_ready() method scans all the wires that feed the input ports of the component and determines if each of the wires has an input that corresponds to the local time of the component.

```
boolean
Port::inputs_are_ready(ckt_time t)
{
        for (Port_Node *list = prt_list; list != 0;
             list = list->next)
        {
                Signal s = list->prt->get_Signal(t);
                if (s.get_time() == UNDEF_TIME && s ==
                    UNDEF_SIG)
                        return FALSE;
        }
        return TRUE;
}
```

The is_lowest_level() method, as invoked by simulate(), scans all the input ports and determines if their fan-outs are empty. If they are all empty, then the component is not made up of any subcomponents.

```
boolean
Port_List::is_lowest_level()
{
        for (Port_Node *list = prt_list; list != 0; list =
             list->next)
        {
                if (!list->prt->fan_out.is_empty())
                        return FALSE;
        }
        return TRUE;
}
```

The is_empty() method is defined by Component_List. It simply returns a boolean indicating whether or not the linked list of components that it maintains is empty.

The descend() method of Port_List again scans the linked list of ports. However, this time, a propagate() message is sent to the fan-out of each port. It is implemented as follows:

```
void
Port_List::descend()
{
        for (Port_Node *list = prt_list; list != 0; list =
             list->next)
        {
                if (!list->prt->fan_out.is_empty())
                        list->prt->fan_out.propagate();
        }
}
```

The propagate() method is implemented in the Component_List class. This method scans all the components in the linked list and sends simulate() messages to them:

```
void
Component_List::propagate()
{
        for (Component_Node *lst = comp_list; lst != 0; lst =
             lst->next)
        {
                lst->cmp->simulate();
        }
}
```

6.4 Virtual process() function

Object-oriented programming offers a clean, extensible solution to the problem through the use of virtual functions. When the simulate() method sends a process() message, the process routine that is called depends upon which component the simulate() message was sent to. Hence the process() message is resolved during runtime. For example, if the simulate() message was sent to a NAND gate, simulate() would invoke NAND's process() method. If, however, the simulate() message was sent to a NOR gate, NOR's process() method would be invoked instead. The process() function is initially defined in the Component class to output an error message and terminate the execution of the program. Should a user forget to define a process() method for a low-level component, then the component will inherit the process() function from the base class. When the simulate() method attempts to invoke the process() function for that component, an error message is displayed telling the user that it was unable to invoke the component's process() method. The name of the component is also displayed.

The purpose of the process() method is to read inputs from the input ports, process them and generate the appropriate output signals to the output ports. For example, a two-input AND gate would define its process() function as follows:

```
void And2::process(ckt_time t)
{
   if (I1.get_Signal(t) == HIGH && I2.get_Signal(t) == HIGH)
               O1.send_Signal(Signal(HIGH, t + delay));
   else if (I1.get_Signal(t)== LOW || I2.get_Signal(t) == LOW)
               O1.send_Signal(Signal(LOW, t + delay));
   else
               O1.send_Signal(Signal(X, t + delay));
}
```

When getting a signal from a port, there is no need to call an access function in order to determine the value of the signal. This is because the user defined conversion, `operator Sig_Val()`, automatically converts a signal to a signal value when used in the context of a signal value.

6.5 Code Modules/files

There are six modules that are created to implement the basic circuits. The modules are:

a) comp.cpp

This module contains all the definitions necessary for the manipulation of abstract components.

b) complib.cpp

This module contains the behavioral definitions of all the actual components in the predefined component library. These components are instantiated and simulated by the simulator engine.

c) connect.cpp

This module contains the method definitions for the abstract Connector class.

d) port.cpp

This module contains the method definitions for ports and its input and output sub-objects.

e) signal.cpp

This module contains all the methods for signal manipulation.

f) wire.cpp

This module contains the method definitions for wire objects.

g) main.cpp

Main driver program for initializing the primary inputs, constructing and simulating the circuit and displaying the primary outputs

CHAPTER VII

TESTING AND SAMPLE SIMULATIONS

7.1 Introduction

This chapter will trace through some sample simulations to demonstrate how the algorithm works. In the following diagrams, the symbol `*' represents the initial time. The time when signals occur will be represented as numbers beneath the signal in the wire queue. The signals themselves will be represented as H, L and X, representing high, low and unknown respectively. The time delay of the encapsulated subcomponents is represented by a number inside the component's box. The time delay for the enclosing component depends upon the delay for each of its encapsulated subcomponents and their connectivity. The local time for all components at the start of the simulation is set at `*', so when the local time of a component is first incremented, it becomes zero.

7.2 Sample Simulation of Three-Input AND Gate

We first start with a trace of a simple combinational circuit, a three-input AND gate. The initial configuration of the circuit and the external wires is shown in Figure 7.1. Recall that the `Wire` constructor places an `X` as the initial signal of all wires.

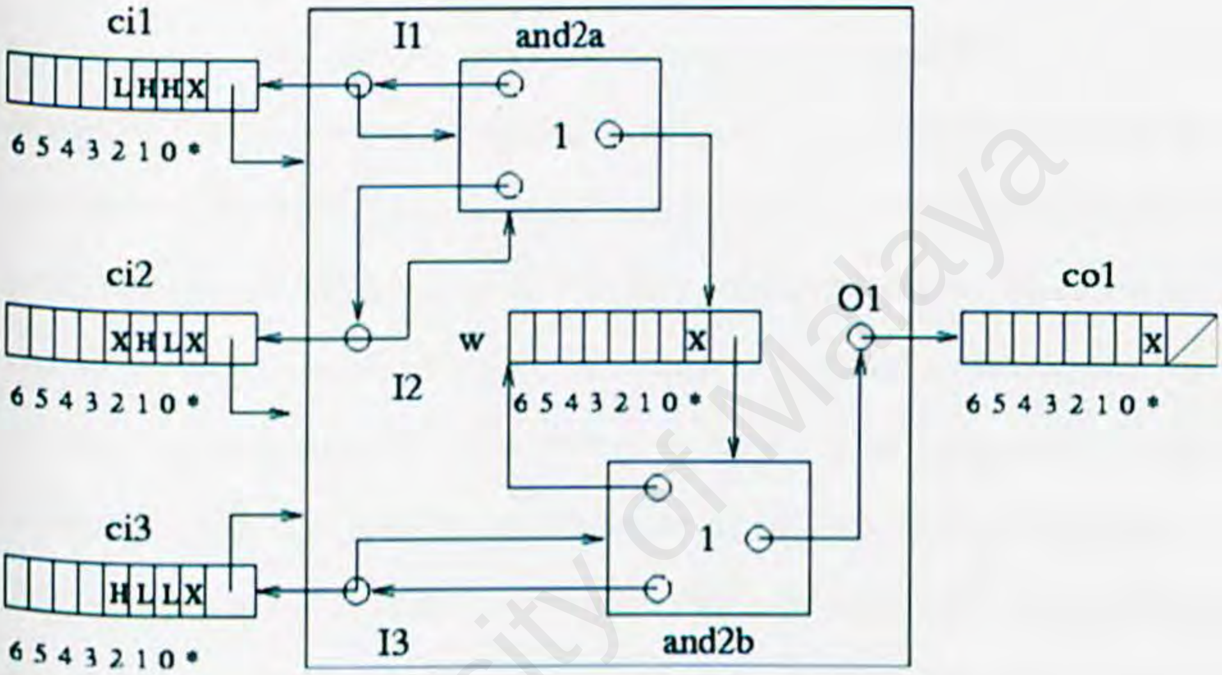


Figure 7.1: Three-Input AND Gate Before Simulation

When the three-input AND gate receives the `simulate()` message, the first thing the component does is to determine if all of its inputs are ready. Since the component is at time '*' and because all wires initially have an unknown stored at that initial time, the algorithm enters the body of the **while** loop and increments the local time of the component. Hence the local time of the three-input AND gate is now zero. The component then sends the `is_lowest_level()` message to its input port list to determine if the component does not have any nested subcomponents. In this case, the three-input AND gate *does* have subcomponents, so the `descend()` message is sent to the input ports list. This method will start scanning all the input ports, sending `simulate()` messages to all the components which are directly connected to the inputs.

It will be assumed that the first subcomponent to receive the `simulate()` message is the `and2a` gate. As with the three-input AND gate, this component now determines if all of its inputs are ready. Obviously, they are, so the local time of the `and2a` gate is increased to zero and the `is_lowest_level()` message is sent to its input ports to determine if the component does not have any subcomponents. This is now the case, so its virtual `process()` function is executed after the delay time is validated. The `process()` message reads the `x` signals from the wires `ci1` and `ci2` at time `'*'` and performs a logical AND operation on them producing another unknown `x` at time zero. This signal is then sent and stored in the nested wire in the three-input AND gate. The wire then takes control and sends `simulate()` messages to all the components to which it is connected. The only component in the wire's fan-out is the `and2b` gate.

Now the inputs to the `and2b` gate are ready, so its local time is incremented to zero and its initial unknown inputs are read from the nested wire and the `ci3` wire by the `process()` method. The resulting signal, `x`, is sent to the output wire `co1` at time $t = 0$. Since this wire has no fan-out, any attempt it makes to propagate its signals is immediately rejected. Control returns back to the `and2b` gate, where it determines that it still has inputs waiting to be processed. Its local time increases to one and its virtual `process()` method is invoked. The `X` input of the nested wire and the `L` input of the `ci3` wire is read and an `L` signal is sent to the output wire at time $t = 1$. When control is returned back to the `and2b` gate, it realizes that all of its inputs are gone, and control eventually returns back to `and2a`.

Since `and2a` has inputs ready at time $t = 0$, its local time is incremented to one and the `H` and the `L` signals are read from `ci1` and `ci2` respectively. The resulting `L` signal is sent to the nested wire at time $t = 1$. The wire once again passes control to the `and2b` gate which increments its local time to two and reads the two `L` signals from its input wires. It sends an `L` signal to the output wire which returns control back to the `and2b` gate. Upon realizing that all its inputs have once again been consumed, control returns yet again to the `and2a` gate.

The `and2a` gate determines that it has two signals waiting for it at time $t = 1$ and therefore increments its local time to two. The two `H` signals are read from the wire and processed with the resulting `H` signal being sent to the nested wire at time $t = 2$. The wire passes control to `and2b` which increments its local time to three and processes the two `H` signals from `ci3` and the nested wire. It then sends the resulting `H` signal to the output wire at time $t = 3$. Control is returned back to `and2a` since `and2b` can go no further.

After incrementing its local time to three, the `and2a` gate reads the final `L` and `X` signals from its two input wires and sends the resulting `L` to the nested wire. When this wire passes control to the `and2b` gate, `and2b` notices that it has a signal `L` waiting for it on the nested wire at time $t = 3$, but there is no corresponding signal on `ci3`. Therefore, control is eventually returned back to `and2a` which realizes that it too has consumed all of its inputs. Control is finally passed back to the encompassing three-input AND gate. This component continues to trivially execute its outer **while** loop, incrementing its local time through each iteration. Every time it tries to send a message to its subcomponents, telling them to simulate, control is immediately returned back since all of its subcomponents are finished. Eventually, the local time of the three-input AND gate reaches three, and it realizes that all of its inputs have been processed, hence terminating the simulation. The final configuration of the internal and external wires after the simulation is shown in Figure 7.2.

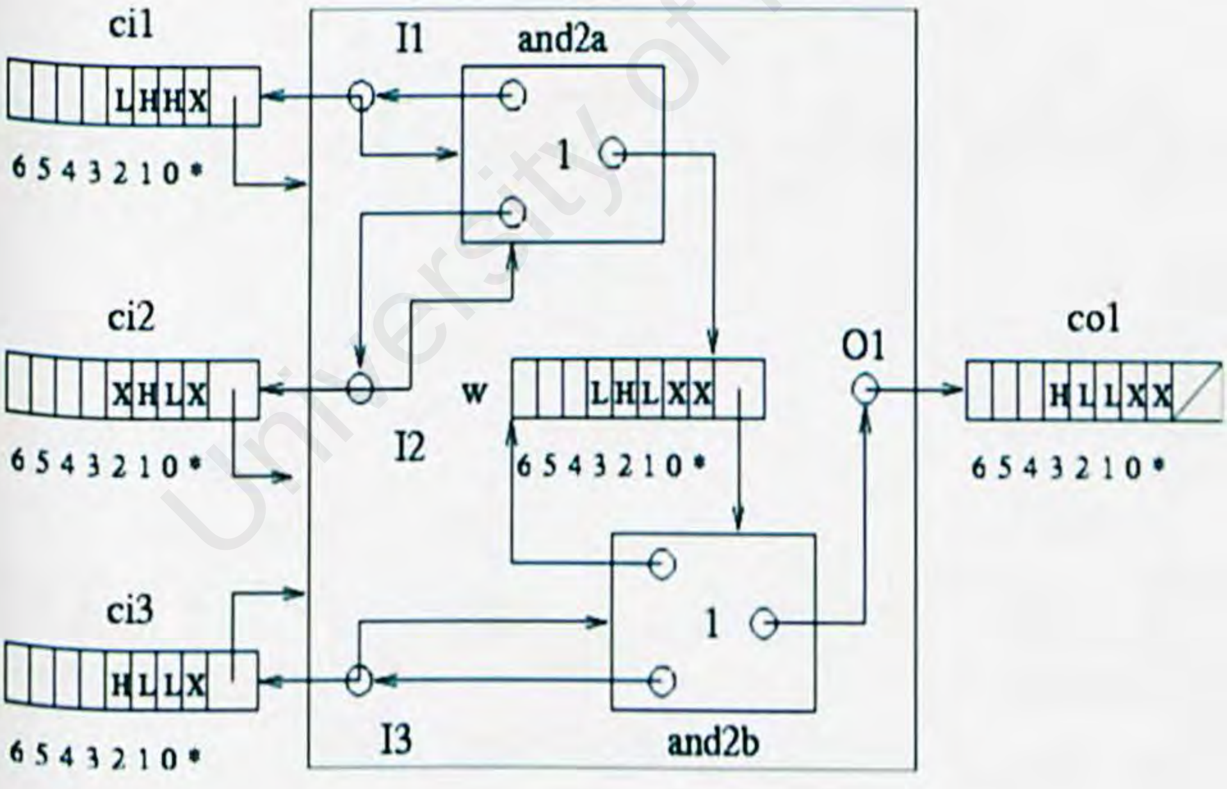


Figure 7.2: Three-Input AND Gate After Simulation

7.3 Sample Simulation of RS-Latch

Next, the simulation of a component with feedback is traced. This trace will demonstrate why it is necessary to increment the local time at the top of the **while** loop and why it is necessary for the `Wire` constructor to place an `X` at the initial time in the wire queue. The initial configuration of the circuit and its corresponding wires is shown in Figure 7.3. Note that the input wires have some empty time slots. This means that the signal at that time is the same as the signal that was before it. For example, in `ci1`, the signal at time $t = 1$ is the same as the signal at time $t = 0$, that is, `L`. It was necessary to space out the input signals in order to avoid resonance during the simulation of the RS-latch.

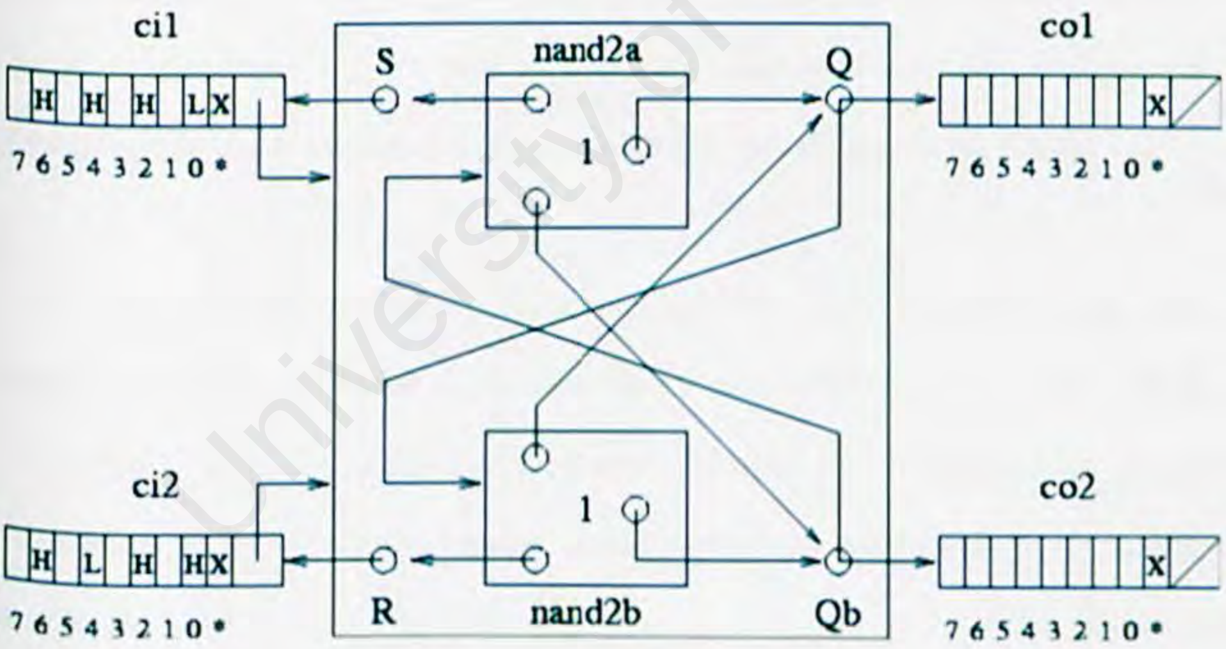


Figure 7.3: RS-Latch Before Simulation

As was the case with the three-input AND gate, the RS-latch receives a message telling it to simulate. The RS-latch, being at its initial time, has inputs ready waiting for it. It then increments its local time to zero and sends `simulate()` messages to all its subcomponents via the `descend()` method.

At this point, it is assumed that `nand2a` is the first component to receive the `simulate()` message. It examines its two input wires, `ci1` and `co2` and discovers that they both have inputs waiting at times corresponding to its local initial time. Note that if the `Wire` constructor had not placed an initial `X` input on the `co2` wire, the simulation would essentially stop at this point producing no output. The `nand2a` gate increases its local time to zero and sends its processed result (`X`) to the `Q` output port at time $t = 0$. `Q` takes this signal and sends it to the wire `co1`. Since `co1` is not connected to anything, control is passed back to `Q`. `Q` then sends `simulate()` messages to all the components in its fan-out. Control is passed to `nand2b` since it is the only element in the fan-out.

Upon receiving control, `nand2b` discovers that its input signals are ready from wires `ci2` and `co1`. It increments its local time to zero and sends an `X` signal to `Qb` at time $t = 0$. Upon receipt of this signal, `Qb` sends the signal to `co2` which passes control back to `Qb`. `Qb` then sends the simulate message to all the components in its fan-out; namely `and2a`.

It is at this point that we realize the importance of increasing a component's local time immediately after determining whether all its inputs are ready. Had we postponed the incrementing of the local time until after the `process()` method, `nand2a` would still be at its initial local time, causing it to re-read the signals it read previously, hence resulting in an infinite loop. However, because `nand2a` previously incremented its time to zero, it is able to look ahead at the next set of inputs. Upon discovering that more inputs are available, the local time is increased to one. The `L` from `ci1` and the `X` from `co2` are nanded together to produce `H` which is sent to `Q` at time $t = 1$. After this signal is sent to the wire `co1`, `Q` sends a `simulate()` message to `nand2b`, at which point `nand2b` increments its local time to one, reads the `H` and the `X` from `ci2` and `co1` respectively and sends the resulting `X` to `Qb` at time $t = 1$.

After placing this signal in the wire, control is again passed to `nand2a` which determines that all its inputs are ready and therefore increases its local time to two. An `L` signal is read from `ci1` (unchanged from the previous time) and is processed with the `X` read from `co2`. The resulting `H` signal is sent to `Q` at time $t = 2$.

After `Q` places the signal on the wire, control is passed yet again to `nand2b`. All its inputs are ready, so it increments its local time to two and processes the `H` signal from `ci2` and the `H` signals from `co1` to produce an `L` output at time $t = 2$.

This method of passing control back and forth continues until all the inputs on `ci1` and `ci2` have been exhausted. Then, like in the three-input AND gate, the RS-latch's local time catches up to the local time of its subcomponents. All the messages which are sent to the two NAND gates are essentially ignored and the simulation ends when the local time of the RS-latch exceeds the time of the last input on the wire. In this particular example, the simulation will terminate when the local time of the RS-latch reaches seven.

A diagram illustrating the state of the RS-latch after the simulation is shown in Figure 7.4.

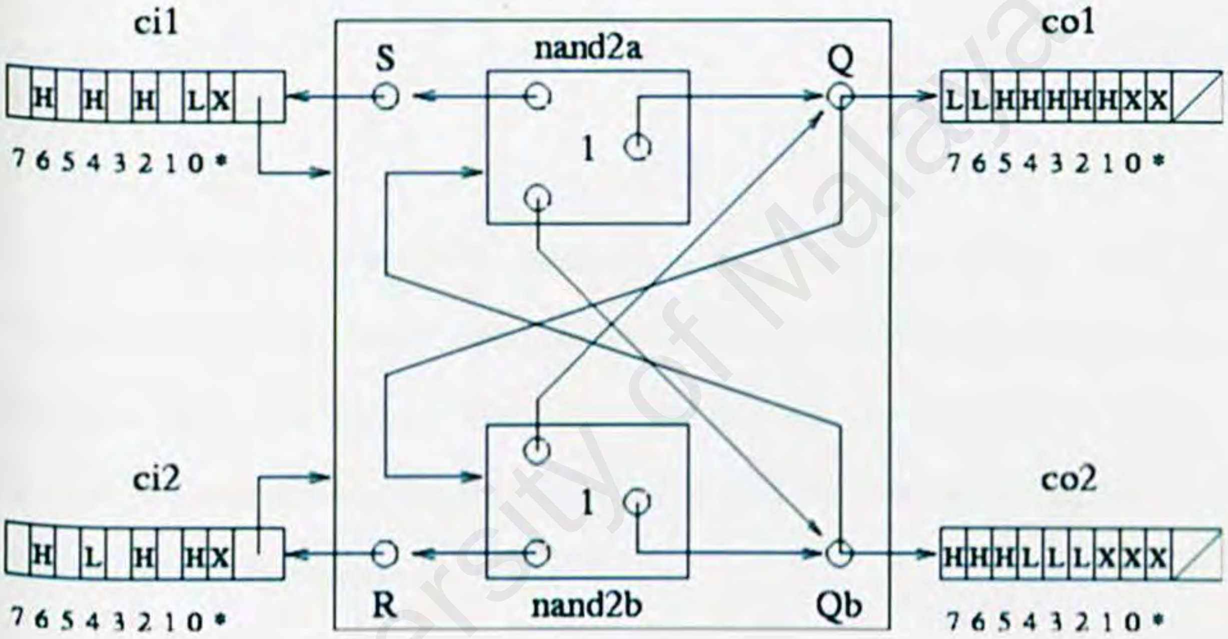


Figure 7.4: RS-Latch After Simulation

CHAPTER VIII

DISCUSSION

8.1 Introduction

I have experienced many challenges during doing this project which is "Hardware Description Using Object-Oriented Paradigm". The most important part is I have to be clear in the concept of object-oriented language, the advantages of object-oriented language and how this language can be used for both the description and simulation of hardware modules.

Nowadays as we know that most of the students taking course in computer field will learn C++ language which is the powerful language in object-oriented paradigm. So in this project I try to use the capabilities of C++ in order to support for circuit designs at several levels of abstraction. Generally my objectives that stated in the proposal for this project were achieved.

8.2 Problem Encountered and Solutions

Throughout the development process, a few problem were encountered that were eventually resolved. Some of the solution came easily but there were those that required an alternative solution.

8.2.1 Problems in getting the resources

There was some difficulty in searching the information and reference book on how to program using C++ for hardware description. However with the limited resources by surfing the internet, reading reference book, discussion with some expertise in this field and depth discussion with my supervisor I coming out with the earlier stages of development.

8.2.2 Lack of Knowledge in the Language

Due to the time constraint, the learning and developing process was done in parallel. Although I have studied the language before but there are still a lot of part in C++ language that I have to concentrate such as data structure and Standard Template Library (STL). Besides that I also have to study on how the hardware and simulation description can be program in C++. With this limited understanding of the programming language, a lot of time was spent in looking for the solutions that occurred while coding the program.

8.2.3 Time Constraint

This project had to be built in a semester's time. A lot of time was needed to learn new thing. So here I just concentrated on basic circuits such as AND gate. The basic modules which are important already included in the workspace and for most complicated circuit can be added and recommended for future enhancements.

8.3 Project Strength

8.3.1 Predefined Library

All the basic modules that are needed in running the hardware and simulation are included in the program. There are also a module contains the behavioral definitions of all the actual components in the predefined component library. These components are instantiated and simulated by the simulator engine.

8.3.2 Easy to Create Sample Gate

We can test the sample gate with some test inputs and then simulate and display the resultant output in the main program. We can change the inputs in order to be more understand how it's work.

8.4 Project Limitation

The program is only running under DOS and the system only concentrate on basic circuit to simulate the hardware which are AND gate and RS-Latch.

8.5 Future Enhancement

For future enhancement, the coding can be integrates to Graphic User Interface (GUI) so that it can be more user-friendly. Current stages, the user only can see the output in signal HIGH, LOW or unknown. In future I hope that user can see the signal in wave format.

CONCLUSIONS

The object-oriented programming paradigm appears to be better suited for hardware description and simulation than the structured programming paradigm. As shown throughout the report, the concepts of encapsulation, inheritance and runtime binding are indispensable when attempting to describe and simulate hardware using the same language. Since C++ can be used for the modeling and testing of hardware designs, the language creates a uniform environment for hardware description and simulation.

The C++ language is much more modular and powerful than traditional structured programming languages. The source code for the description and simulation consisted of only about 800 lines, with the most complicated method requiring less than a dozen lines of code. C++ also permits new components to be added to the library quickly and easily with little threat of error. This makes C++ and the object-oriented programming paradigm very amicable to the fields of hardware design and simulation.

APENDICES A

.CPP FILES

```
1. comp.cpp
2. complib.cpp
3. connect.cpp
4. port.cpp
5. signal.cpp
6. wire.cpp
7. main.cpp
```

```
/************************************************************************
 * comp.cpp
 * ----------------------------------------------------------------------
 *
 * This module contains all the definitions necesary for the manipulation
 * of abstract components
 *
 ************************************************************************/

#include        <iostream>
#include        <list>

#include        "sim.h"              /* For generate_tabs() decl        */
#include        "port.h"             /* Req'd by inputs_are_ready()     */
#include        "signal.h"           /* Req'd by inputs_are_ready()     */
#include        "comp.h"

using namespace std;

using std::endl;
using std::list;

/*
 * Method definitions of the abstract base class for Component.
 */

/*
 * Create a component object, initialize the delay time, name, and
 * local time of the object. This constructor is almost always
 * called from the member initialization list of a component
 * derived from Component.
 */
Component::Component(ckt_time t, const char *nm) :
        delay(t), local_time(CKT_TIME_INIT)
{
        name = new char[strlen(nm) + 1];
        (void) strcpy(name, nm);
}

/*
 * Deallocate the storage used to store the name of the component.
 */
Component::~Component()
{
        delete [] name;
}

/*
 * The main method responsible for simulating a component. It's inherited
 * by all components. Basic approach is to continue executing this method
 * while the component has inputs available.  When the component has all
 * of its inputs ready at its given local time, invoke its process() message
 * passing it the time the inputs were discovered to exist (i.e. at time
 * local_time - 1).
 */
void
Component::simulate()
{

        // Are ALL the inputs ready at the local time of the Component?
        while (inputs_are_ready() == TRUE)
        {
                // If so, then increment local time here.
                // Otherwise, circuits with feedback go on forever.
                local_time++;

                // Send a process method to the component.
                // This may trigger further simulate()
                // messages depending upon the connectivity
                // of the component.
                process(local_time - 1);
        }
}

/*
 * This method determines if the signals which are directed to the input
 * ports of the component are available at the given time.  If they are all
 * ready, return TRUE otherwise return FALSE.
 */
boolean
Component::inputs_are_ready() const
{
        list<Port *>::const_iterator        p;

        // Scan the linked list of port pointers.
        for (p = I_List.begin(); p != I_List.end(); p++)
        {
                // Get the signal directed to the port at localtime
                Signal s = (*p)->get_signal(local_time);

                // If the signal does not exist at that time then
                // return FALSE.
                if (s.get_time() == CKT_TIME_NULL && s.get_value() == SIG_NULL)
                        return FALSE;
        }

        // Otherwise, all signals are ready.
        return TRUE;
}

/*
 * All nonleaf components should inherit this process() function.  If a
 * component is composed of sub-components, then descend the three
 * dimensional hierarchy, sending simulate() messages to all the
 * component's subcomponents.  The simulate() is sent by the propagate()
 * message of the Connector class.  This method triggers the subcomponents
 * of the enclosing component.  It esentially descends the
 * three-dimensional hierarchy of components.
 */

void
Component::process(ckt_time)
{
        list<Port *>::const_iterator        p;

        // Scan all the port pointers in the port list
        // and activate all the subcomponents.
        for (p = I_List.begin(); p != I_List.end(); p++)
                (*p)->propagate();
}

/*
 * Display all the output signals.
 */
void
Component::show_outputs() const
```

```
{
	list<Port *>::const_iterator		p;

	for (p = O_List.begin(); p != O_List.end(); p++)
		(*p)->show_signals();
}

/*
 * Display the component's name.
 */
void
Component::display(ostream &os, int tabs) const
{
	char	*indent = generate_tabs(tabs);

	os << indent << "Component Name: " << name << endl;

	os << indent << "Component's Input Port List:\n";
	display_ports(os, I_List, tabs + 1);
	os << indent << "--- end component's input ports ---\n";

	os << indent << "Component's Output Port List:\n";
	display_ports(os, O_List, tabs + 1);
	os << indent << "--- end component's output ports ---\n";

	delete [] indent;
}

/*
 * Display the ports in the list.
 */
void
Component::display_ports(ostream &os, list<Port *> ports, int tabs) const
{
	list<Port *>::const_iterator		p;

	char		*indent = generate_tabs(tabs);
	int		 counter = 0;

	for (p = ports.begin(); p != ports.end(); p++)
	{
		os << indent << "Port #" << ++counter << endl;
		(*p)->display(os, tabs + 1);
	}

	delete [] indent;
}
```

```
/***********************************************************************
 * complib.cpp
 * ---------------------------------------------------------------------
 *
 * This module contains the behavioural definitions of all the actual
 * components in the predefined component library.  These components
 * are instantiated and simulated by the simulator engine.
 *
 ***********************************************************************/

#include      "complib.h"

using namespace std;


/*
 * Component library. All user defined components go here.
 * Only components at the lowest level of abstract need to
 * redefine the virtual process() function.
 */

#define CONNECT(Port, Wire, name) Port(*this, Wire, name)

And2::And2(Connector &ci1, Connector &ci2, Connector &co1,
        ckt_time dly, char *n) :
       Component(dly, n),
       CONNECT(I1, ci1, "And2 I1"),
       CONNECT(I2, ci2, "And2 I2"),
       CONNECT(O1, co1, "And2 O1")
{ }

void
And2::process(ckt_time t)
{
       Sig_Val                 sigval1 = I1.get_signal(t).get_value();
       Sig_Val                 sigval2 = I2.get_signal(t).get_value();

       if (sigval1 == SIG_HIGH && sigval2 == SIG_HIGH)
              O1.send_signal(Signal(t + delay, SIG_HIGH));
       else if (sigval1 == SIG_LOW || sigval2 == SIG_LOW)
              O1.send_signal(Signal(t + delay, SIG_LOW));
       else
              O1.send_signal(Signal(t + delay, SIG_X));
}

Nand2::Nand2(Connector &ci1, Connector &ci2, Connector &co1,
         ckt_time dly, char *n) :
       Component(dly, n),
       CONNECT(I1, ci1, "Nand2 I1"),
       CONNECT(I2, ci2, "Nand2 I2"),
       CONNECT(O1, co1, "Nand2 O1")
{ }

void
Nand2::process(ckt_time t)
{
       Sig_Val                 sigval1 = I1.get_signal(t).get_value();
       Sig_Val                 sigval2 = I2.get_signal(t).get_value();

       if (sigval1 == SIG_LOW || sigval2 == SIG_LOW)
              O1.send_signal(Signal(t + delay, SIG_HIGH));
       else if (sigval1 == SIG_HIGH && sigval2 == SIG_HIGH)
              O1.send_signal(Signal(t + delay, SIG_LOW));
```

```
      else
            O1.send_signal(Signal(t + delay, SIG_X));
}

Or2::Or2(Connector &ci1, Connector &ci2, Connector &co1,
         ckt_time dly, char *n) :
      Component(dly, n),
      CONNECT(I1, ci1, "Or2 I1"),
      CONNECT(I2, ci2, "Or2 I2"),
      CONNECT(O1, co1, "Or2 O1")
{ }

void
Or2::process(ckt_time t)
{
      Sig_Val             sigval1 = I1.get_signal(t).get_value();
      Sig_Val             sigval2 = I2.get_signal(t).get_value();

      if (sigval1 == SIG_HIGH || sigval2 == SIG_HIGH)
            O1.send_signal(Signal(t + delay, SIG_HIGH));
      else if (sigval1 == SIG_LOW && sigval2 == SIG_LOW)
            O1.send_signal(Signal(t + delay, SIG_LOW));
      else
            O1.send_signal(Signal(t + delay, SIG_X));
}


Xor2::Xor2(Connector &ci1, Connector &ci2, Connector &co1,
         ckt_time dly, char *n) :
      Component(dly, n),
      CONNECT(I1, ci1, "Xor2 I1"),
      CONNECT(I2, ci2, "Xor2 I2"),
      CONNECT(O1, co1, "Xor2 O1")
{ }

void
Xor2::process(ckt_time t)
{
      Sig_Val             sigval1 = I1.get_signal(t).get_value();
      Sig_Val             sigval2 = I2.get_signal(t).get_value();

      if ((sigval1 == SIG_LOW  && sigval2 == SIG_HIGH) ||
          (sigval1 == SIG_HIGH && sigval2 == SIG_LOW))
            O1.send_signal(Signal(t + delay, SIG_HIGH));
      else if ((sigval1 == SIG_LOW  && sigval2 == SIG_LOW) ||
               (sigval1 == SIG_HIGH && sigval2 == SIG_HIGH))
            O1.send_signal(Signal(t + delay, SIG_LOW));
      else
            O1.send_signal(Signal(t + delay, SIG_X));
}

And3::And3(Connector &ci1, Connector &ci2,
         Connector &ci3, Connector &co1,
         ckt_time dly, char *n) :
      Component(dly, n),
      CONNECT(I1, ci1, "And3 I1"),
      CONNECT(I2, ci2, "And3 I2"),
      CONNECT(I3, ci3, "And3 I3"),
      CONNECT(O1, co1, "And3 O1"),
      w("And3 wire"),
      and2a(I1, I2, w, 1L, "And2a"),
      and2b(w, I3, O1, 1L, "And2b")
{ }
```

```
RS_Latch::RS_Latch(Connector &ci1, Connector &ci2,
                  Connector &co1, Connector &co2,
                  ckt_time dly, char *n) :
       Component(dly, n),
       CONNECT(R, ci1, "R"),
       CONNECT(S, ci2, "S"),
       CONNECT(Q, co1, "Q"),
       CONNECT(Qb, co2, "Qb"),
       nand2a(R, Qb, Q, 1L, "Nand2a"),
       nand2b(Q, S, Qb, 1L, "Nand2b")
{ }


Half_Adder::Half_Adder(Connector &ci1, Connector &ci2,
                  Connector &co1, Connector &co2,
                  ckt_time dly, char *n) :
       Component(dly, n),
       CONNECT(X, ci1, "X"),
       CONNECT(Y, ci2, "Y"),
       CONNECT(S, co1, "S"),
       CONNECT(C, co2, "C"),
       xor2a(X, Y, S, 1L, "Xor2a"),
       and2a(X, Y, C, 1L, "And2a")
{ }
```

```
/*********************************************************************
 * connect.cpp
 *
 * --------------------------------------------------------------------
 *
 * This module contains the method definitions for the abstract
 * Connector class.
 *
 *********************************************************************/

#include      <list>
#include      <iostream>

#include      "sim.h"                     /* For generate_tabs() decl      */
#include      "comp.h"             /* For propagate() & display()    */
#include      "connect.h"

using namespace std;


using std::list;
using std::endl;

/*
 * Method definitions for the Connector class.  Connectors are responsible
 * for connecting components with one another (Wires) and the outside world
 * (Ports). It is through connectors that the transmission of signals
 * occurs.
 */

/*
 * Trivial Connector constructor. Assign name passed
 * in to the name of the Connector.
 */
Connector::Connector(const char *nm)
{
       name = new char[strlen(nm) + 1];
       (void) strcpy(name, nm);
}

/*
 * Deallocate the storage used to store the name of the connector.
 */
Connector::~Connector()
{
       delete [] name;
}

/*
 * Return a constant pointer to the name.
 */
const char *
Connector::get_name() const
{
       return name;
}

/*
 * When a Connector object receives a connect message, it adds the
 * component pointer passed in to the fan-out of the connector.  This
 * method is used exclusively by the Port constructor which is passed a
 * pointer to the component via the CONNECT macro. This component is added
 * to the fan-out of the connector passed into the Port constructor.
 */
void Connector::connect(Component &cmp)
{
	// Add the component pointer to the fan-out list.
	fan_out.push_back(&cmp);
}

/*
 * Inform each of the component's in the connector's fan-out list
 * to simulate, thereby propagating signals into the circuit.
 */
void
Connector::propagate() const
{
	list<Component *>::const_iteratorc;

	// Scan all the elements in the component pointer list.
	for (c = fan_out.begin(); c != fan_out.end(); c++)
	{
		// Send simulate() messages to each component in the list.
		(*c)->simulate();
	}
}

/*
 * Display the connector name and its fan-out.
 */
void
Connector::display(ostream &os, int tabs) const
{
	char   *indent = generate_tabs(tabs);

	os << indent << "Connector Name: " << name << endl;

	os << indent << "Connector Fanout List:\n";
	list<Component *>::const_iterator  cmp;
	for (cmp = fan_out.begin(); cmp != fan_out.end(); cmp++)
		(*cmp)->display(os, tabs + 1);

	os << indent << "--- end fanout list ---\n";

	delete [] indent;
}
```

```
/**********************************************************************
 * port.cpp
 * --------------------------------------------------------------------
 *
 * This module contains the method definitions for ports and its
 * input and output subobjects.
 *
 **********************************************************************/

#include        <iostream>

#include        "sim.h"                     /* For generate_tabs() decl      */
#include        "port.h"
#include        "comp.h"     /* Component's methods are called here  */

using namespace std;


using std::endl;
using std::cerr;

/*
 * Constructor for a port. Set the name of the port by passing the name to
 * the Connector base class constructor.  Assign the encapsulated external
 * Connector pointer member to the address of the connector passed to the
 * constructor.  External will point to the Connector that feeds the port
 * (if the port is an inport port) or to the Connector that the port feeds
 * (if the port is an output port).
 */
Port::Port(Connector &con, const char *nm) :
       Connector(nm), external(&con)
{ }

/*
 * Method used to get and return a signal from a port which occured at time
 * t. Like the preceding method, this method will recurse until the
 * get_signal() messages is sent to a wire, after which the recursion
 * unfolds.
 */
Signal
Port::get_signal(ckt_time t) const
{
       // Send message to the external feeder to get its signal.
       return external->get_signal(t);
}

/*
 * Method sends a supplied signal to the external world.  This method will
 * recurse until a send_signal() message is sent to a Wire. Since every
 * port connects eventually to a wire, this recursion will eventually
 * terminate. After sending the signal to the outside world, an attempt is
 * made to propagate the signal to any components which are at the same
 * level of hierarchy and are in the fanout list of the port.
 */
void
Port::send_signal(Signal s)
{
       // Send the signal to the external Connector.
       external->send_signal(s);

       // Propagate the signal at the current level of the hierarchy.
       propagate();
}
```

```
/*
 * Send a message to the external connector to display all the signals
 * that have travelled along it.
 */
void
Port::show_signals() const
{
	// Instruct the external connector to show its signal list.
	external->show_signals();
}

/*
 * Display the port information.
 */
void
Port::display(ostream &os, int tabs) const
{
	char   *indent = generate_tabs(tabs);

	os << indent << "Port's external connector: "
		<< external->get_name() << endl;
	Connector::display(os, tabs);

	delete [] indent;
}

/*
 * Input constructor. Invoke Port's constructor to do most of the work.
 * Add the component passed in to the fan-out list external connector.
 * Then add the Input port to the linked list of input ports already in the
 * component.
 */
Input::Input(Component &cmp, Connector &con, const char *n) :
	Port(con, n)
{
	// Send the connect message to the external feeding connector.
	// Note: The message fan_out.add() could not be sent directly
	// to the connector, since fan_out is an inaccessible member
	// of Connector con -- this port constructor can only access its
	// own fan-out, not the fan-out of another Connector.
	external->connect(cmp);

	// Add Input port to component's linked list of Input ports
	// pointers.  Note that 'add' takes a constant reference
	// parameter.  Therefore, we cannot pass 'this' explicity.
	cmp.I_List.push_back((Port *)this);
}

/*
 * Method to warn about send_signal() messages being sent to an Input port.
 */
void
Input::send_signal(Signal)
{
	// Output a warning message if an input signals tries to send a signal.
	cerr << "Warning: Cannot send a signal via Input port.\n";
}

/*
 * Output constructor. Invoke Port's constructor to do most of the work.
 * Add the Output port to the linked list of output ports already in the
 * component.
 */
```

```
Output::Output(Component &cmp, Connector &con, const char *n) :
        Port(con, n)
{
        // Add Output port to component's linked list of Output ports
        // pointers.  Note that add takes a constant reference
        // parameter.  Therefore, we cannot pass 'this' explicity.
        cmp.O_List.push_back((Port *)this);
}
```

```
/**********************************************************************
 * signal.cpp
 *
 * ----------------------------------------------------------------------
 *
 * This module contains all the methods for signal manipulation.
 *
 ***********************************************************************/

#include      "signal.h"


/*
 * Method definitions for Signal class.
 */

/*
 * Signal constructor. Assign the supplied signal value and signal time to
 * the private members of the signal class. This is the only place where
 * such an assignment can be made.
 */
Signal::Signal(ckt_time ct, Sig_Val sv) :
        t(ct), value(sv)
{ }

Signal::Signal(const Signal &sig)
{
       value = sig.value;
       t = sig.t;
}

Signal &
Signal::operator=(const Signal &sig)
{
       if (&sig == this)
              return *this;

       value = sig.value;
       t = sig.t;
       return *this;
}

/*
 * User defined conversion. Used to automatically convert a signal object
 * into its signal value.  This method is very useful in the process()
 * methods of components.
 */
Sig_Val
Signal::get_value() const
{
       // Simply return the signal value.
       return value;
}

/*
 * Access method for getting the time of the signal.  Could not be
 * implemented as a user defined conversion because its return value (long)
 * clashes with the return value of operator Sig_Val() (int).
 */
ckt_time
Signal::get_time() const
{
       // Simply return the signal time.
```

```
        return t;
}

/*
 * Modify the signal value.  This method is required as zero-delay
 * components with feed back stabilize their outputs.
 */
void
Signal::set_value(Sig_Val sv)
{
        value = sv;
}

/*
 * Overloaded << operator. This enables signals to be output just like any
 * other built in type in C++. Somewhat long but very straightforward.
 */
ostream &
operator <<(ostream &os, const Signal &s)
{
        os << "{";

        // Output special times as strings instead of numbers.
        switch (s.t)
        {
                case CKT_TIME_INIT:
                        os << "_";
                        break;
                case CKT_TIME_NULL:
                        os << "?";
                        break;
                default:
                        // Get output times to line up nicely.
                        os << s.t;
                        break;
        }

        os << " ";

        // Output Sig_Val enumerated type.
        switch (s.value)
        {
                case SIG_LOW:
                        os << "L";
                        break;
                case SIG_HIGH:
                        os << "H";
                        break;
                case SIG_X:
                        os << "X";
                        break;
                case SIG_NULL:
                        os << "?";
                        break;
                default:
                        os << "BAD SIGNAL!";
                        break;
        }

        // Return ostream in case <<'s are used in succession.
        return os << "} ";
}
```

```
/***********************************************************************
 * wire.cpp
 *
 * ---------------------------------------------------------------------
 *
 * This module contains the method definitions for wire objects.
 *
 ***********************************************************************/

#include      <iostream>
#include      <list>

#include      "sim.h"                      /* For generate_tabs() decl       */
#include      "wire.h"

using namespace std;


using std::cerr;
using std::cout;
using std::endl;
using std::list;

/*
 * Trivial wire constructor. Pass the name to the base Connector constructor
 * and initialize the Signal list by adding an unknown input at the initial
 * time to the list.
 */
Wire::Wire(const char *nm) :
       Connector(nm)
{
       // Add initial signal.
       add_signal(Signal(CKT_TIME_INIT, SIG_X));
}

/*
 * Wire constructor used to add a series of signals to a wire. Useful for
 * placing a set of initial signals on, say, an input wire for circuit
 * testing. A wire constructor should also be provided which reads these
 * signals from an input file.
 */
Wire::Wire(Signal s[], int num, char *nm) :
       Connector(nm)
{
       // Add initial signal.
       add_signal(Signal(CKT_TIME_INIT, SIG_X));

       // Add signals which are in the signal array to the wire.
       for (int i = 0; i < num; i++)
              add_signal(s[i]);
}

/*
 * Add the specified signal to the wire.  This function should be
 * called to put initial signals on an input wire -- the signals
 * will not actually be propagated until the simulation begins.
 */
void
Wire::add_signal(Signal sig)
{
       // Append the signal to the signal list.
       signals.push_back(sig);
}
```

```
/*
 * Method used to find a signal in the wire's encapsulated signal list.  If
 * the signal is not found, an undefined signal (undefined value, undefine
 * time) is returned.
 */
Signal
Wire::get_signal(ckt_time t) const
{
      // If the signal list is empty or if the time we are looking for
      // is greater than the time the last signal came into the wire,
      // then assume an undefined signal.  We should also inform the
      // requesting component to not update its local time until it can
      // be verfied that this assumption is correct.  (For now, we return
      // an undefined signal until we can find a way to inform the
      // component to not update its local time).
      if (signals.empty() || signals.back().get_time() < t)
            return Signal(CKT_TIME_NULL, SIG_NULL);

      list<Signal>::const_iterator s (signals.begin());
      Signal found;

      // Otherwise, start the scan of the signal linked list.
      while (s != signals.end() && (*s).get_time() <= t)
            found = *s++;
      return found;
}

/*
 * Method used to send an add() message to the encapsulated signal list of
 * Wire. After adding the signal to the wire, a message is sent to the
 * fan-out of the wire (inherited from Connector) to propagate the signal
 * as far as possible.
 */
void
Wire::send_signal(Signal sig)
{
      add_signal(sig);
      propagate();
}

/*
 * Display all the signals in the wire.  The output of this method
 * is suitable for parsing by another application.
 */
void
Wire::show_signals() const
{
      cout << "output:" << endl << "\tid: " << get_name() << endl;
      cout << "\tvalues: ";

      // Dump the signals on the wire.
      display_signals();
      cout << endl;
}

/*
 * This method is used to show the user signals (value and time) on the
 * wire.  In addition, information pertaining to the wire's connector
 * information is also displayed.
 */
void
Wire::display(ostream &os, int tabs) const
```

```
{
        char   *indent = generate_tabs(tabs);

        // Output the name of the wire
        os << indent << "Wire values: (";

        // Output the signal values in the wire.
        display_signals();
        os << ")" << endl;

        // Dump the connector information.
        Connector::display(os, tabs + 1);

        delete [] indent;
}

/*
 * Replace a signal in the wire's signal list.  The time of the signal to
 # replace and the (possibly) new value is given by sig.
 */
void
Wire::replace(Signal sig)
{
        list<Signal>::iterator     s;

        for (s = signals.begin(); s != signals.end(); s++)
                if ((*s).get_time() == sig.get_time())
                        (*s).set_value(sig.get_value());
}

/*
 * Method used to loop through all the signals in the encapsulated signal
 * list, displaying them one by one.  Output of the signals is of the
 * form {0 1} {5 X} {6 1} {9 0} ... (i.e. {time1 value1} {time2 value2} ...).
 * Only the deltas are reported.
 */
void
Wire::display_signals() const
{
        list<Signal>::const_iterator      sig;
        Sig_Val                          prev_val = SIG_X;
        int                          num = signals.size();
        boolean                          first = TRUE;

        // Scan all the signals in the list.
        // Dump the signals on the wire.
        for (sig = signals.begin(); sig != signals.end(); sig++)
        {
                num --;
                if (first || (*sig).get_value() != prev_val || num == 0)
                {
                        cout << *sig;
                        prev_val = (*sig).get_value();
                }
                first = FALSE;
        }
}
```

```
/*********************************************************************
 * main.cpp
 *                          2-INPUT AND GATE
 * -------------------------------------------------------------------
 *
 * Main driver program for intializing the primary inputs, constructing
 * and simulating the circuit and displaying the primary outputs.
 *
 *********************************************************************/

#include      "sim.h"                      /* For generate_tabs() decl      */
#include      "signal.h"
#include      "wire.h"
#include      "complib.h"

using namespace std;


/*
 * Generate a string containing the specified number of tab characters
 * for indenting output purposes.
 */
char *
generate_tabs(int num)
{
       char   *tabs = new char[num + 1];
       int     t;

       for (t = 0; t < num; t++)
              tabs[t] = '\t';
       tabs[t] = '\0';
       return tabs;
}


/*
 * Main driver function:
 * The main program will programmatically
 * create a 2-input AND gate with some test inputs and then simulate
 * and display the resultant output.
 */

int
main()
{
       Signal  Signal1[] =
       {
              Signal(0, SIG_HIGH),
              Signal(1, SIG_LOW),
              Signal(2, SIG_HIGH),
              Signal(3, SIG_LOW),
              Signal(4, SIG_LOW),
              Signal(5, SIG_HIGH),
              Signal(6, SIG_LOW),
              Signal(7, SIG_LOW),
              Signal(8, SIG_HIGH),

       };

       Signal  Signal2[] =
       {
              Signal(0, SIG_LOW),
              Signal(1, SIG_HIGH),
        Signal(2, SIG_HIGH),
        Signal(3, SIG_LOW),
        Signal(4, SIG_HIGH),
        Signal(5, SIG_LOW),
        Signal(6, SIG_HIGH),
        Signal(7, SIG_HIGH),
        Signal(8, SIG_HIGH),

    };


    Wire    w1(Signal1, sizeof(Signal1) / sizeof(Signal), "Main in_w1");
    Wire    w2(Signal2, sizeof(Signal2) / sizeof(Signal), "Main in_w2");
    Wire    w3("2_input_AND");
    And2    and2(w1, w2, w3, CKT_TIME_NULL, "and2");

    and2.simulate();

    and2.show_outputs();


    return 0;

}
```

```
/*********************************************************************
 * main.cpp
 *                          3-INPUT AND GATE
 * -------------------------------------------------------------------
 *
 * Main driver program for intializing the primary inputs, constructing
 * and simulating the circuit and displaying the primary outputs.
 *
 *********************************************************************/

#include      "sim.h"                       /* For generate_tabs() decl      */
#include      "signal.h"
#include      "wire.h"
#include      "complib.h"

using namespace std;


/*
 * Generate a string containing the specified number of tab characters
 * for indenting output purposes.
 */
char *
generate_tabs(int num)
{
        char   *tabs = new char[num + 1];
        int     t;

        for (t = 0; t < num; t++)
                tabs[t] = '\t';
        tabs[t] = '\0';
        return tabs;
}


/*
 * Main driver function:
 * The main program will programmatically
 * create a 3-input AND gate with some test inputs and then simulate
 * and display the resultant output.
 */

int
main()
{
        cout<<"3 Input AND Gate\n";
        cout<<"----------------\n\n";
        Signal  Signal1[] =
        {
                Signal(0, SIG_HIGH),
                Signal(1, SIG_HIGH),
                Signal(2, SIG_LOW),
        };

        Signal  Signal2[] =
        {
                Signal(0, SIG_LOW),
                Signal(1, SIG_HIGH),
                Signal(2, SIG_X),

        };
```

```
    Signal  Signal3[] =
    {
           Signal(0, SIG_LOW),
           Signal(1, SIG_LOW),
           Signal(2, SIG_HIGH),

    };

    Wire   w1(Signal1, sizeof(Signal1) / sizeof(Signal), "Main in_w1");
    Wire   w2(Signal2, sizeof(Signal2) / sizeof(Signal), "Main in_w2");
    Wire   w3(Signal3, sizeof(Signal3) / sizeof(Signal), "Main in_w3");
    Wire   w4("Main out_w4");
    And3   and3(w1, w2, w3, w4, CKT_TIME_NULL, "and3");

    and3.simulate();

    and3.show_outputs();


    cout<<"\n\n";
    return 0;
}
```

```
/***********************************************************************
 * main.cpp
 *                              RS-LATCH
 * ---------------------------------------------------------------------
 *
 * Main driver program for intializing the primary inputs, constructing
 * and simulating the circuit and displaying the primary outputs.
 *
 ***********************************************************************/

#include      "sim.h"                    /* For generate_tabs() decl      */
#include      "signal.h"
#include      "wire.h"
#include      "complib.h"

using namespace std;


/*
 * Generate a string containing the specified number of tab characters
 * for indenting output purposes.
 */
char *
generate_tabs(int num)
{
       char   *tabs = new char[num + 1];
       int     t;

       for (t = 0; t < num; t++)
              tabs[t] = '\t';
       tabs[t] = '\0';
       return tabs;
}


/*
 * Main driver function:
 * The main program will programmatically
 * create a RS_Latch gate with some test inputs and then simulate
 * and display the resultant output.
 */

int
main()
{
       cout<<"\n\n";
       cout<<"Reset-set Latch (RS-Latch)\n";
       cout<<"------------------------\n\n\n";

       Signal  Signal1[] =
       {
              Signal(0, SIG_LOW),
              Signal(2, SIG_HIGH),
              Signal(4, SIG_HIGH),
              Signal(6, SIG_HIGH),
       };

       Signal  Signal2[] =
       {
              Signal(0, SIG_HIGH),
              Signal(2, SIG_HIGH),
              Signal(4, SIG_LOW),
              Signal(6, SIG_HIGH),
    };

    Wire  r(Signal1, sizeof(Signal1) / sizeof(Signal), "Main in_w1");
    Wire  s(Signal2, sizeof(Signal2) / sizeof(Signal), "Main in_w2");
    Wire  Q("Q");
    Wire  Qb("Qb");
    RS_Latch      rs_latch(r, s, Q, Qb, CKT_TIME_NULL, "rs_latch");

    rs_latch.simulate();

    rs_latch.show_outputs();

    cout<<"\n\n";


    return 0;


    return 0;
}
```

```
Signal  Signal2[] =
{
        Signal(0, SIG_LOW),
        Signal(1, SIG_HIGH),
        Signal(2, SIG_HIGH),
        Signal(3, SIG_LOW),
        Signal(4, SIG_HIGH),
        Signal(5, SIG_LOW),
        Signal(6, SIG_HIGH),
        Signal(7, SIG_HIGH),
        Signal(8, SIG_HIGH),

};


Wire    w1(Signal1, sizeof(Signal1) / sizeof(Signal), "Main in_w1");
Wire    w2(Signal2, sizeof(Signal2) / sizeof(Signal), "Main in_w2");
Wire    w3("main out_S");
Wire    w4("main out_C");
Half_Adder   half_adder2(w1, w2, w3, w4, CKT_TIME_NULL, "half_adder2");

half_adder2.simulate();

half_adder2.show_outputs();
cout<<"\n\n";



return 0;
}
```

APPENDICES B

HEADER FILE

1. comp.h
2. complib.h
3. connect.h
4. port.h
5. signal.h
6. wire.h
7. main.h

```
/*************************************************************************
 * comp.h
 * -----------------------------------------------------------------------
 * Class declaration for component class.
 *************************************************************************/

#if !defined(COMP_H_)      /* protect from multiple inclusion */
#  define COMP_H_

#include     <iostream>    /* For ostream declaration        */
#include     "sim.h"            /* For ckt_time/boolean    typedef
        */
#include     <list>        /* Input/Output are linked lists */

class        Port;         /* Forward declaration for I/O_list      */

/* Component objects are the functional units of the circuit that process
 * inputs and produce outputs. */

class Component
{
public:
      virtual            ~Component();

      std::list<Port *>    I_List;      // List of input ports.
      std::list<Port *>    O_List;      // List of output ports.
      virtual void  process(ckt_time); // Method responsible for
                                       // processing component inputs.
                                       // In general, this is
                                       // different for all cmps.
      void  simulate();                // Simulate the component.
      void  show_outputs() const;      // Display the output signals.

      virtual void  display(std::ostream &, int) const;// Display cmp.
                                                 // information.

protected:
      Component(ckt_time = 1L, const char* = "Component");
                                       // Only derived components
                                       // can be created.

      ckt_time      delay;             // Transport delay of component.

private:
      void          display_ports(     std::ostream &,
                                 std::list<Port *>, int) const;
                                       // Used twice by display().

      boolean              inputs_are_ready() const;
                                       // Is cmp ready for simulation?
                                       // Used by simulate().

      char          *name;        // Name of the component.
      ckt_time      local_time; // Local time of component.
};

#endif /* !COMP_H_ */
```

```
/***********************************************************************
 * complib.h
 * -----------------------------------------------------------------------
 * Class definitions for the actual components to be simulated.
 * Each of these components must be derived from the 'Component' class.
 ***********************************************************************/

#if !defined(COMPLIB_H_)         /* protect from multiple inclusion */
#  define COMPLIB_H_

#include     "sim.h"             /* For ckt_time typedef            */
#include     "comp.h"     /* And2 etc. are derived from Component */
#include     "port.h"     /* For definition of Input/Output ports */
#include     "wire.h"     /* For definition of wire class         */

class And2 : public Component
{
public:
        And2(Connector &, Connector &, Connector &,
             ckt_time = 1L, char* = "And2");
        void process(ckt_time);
private:
        Input  I1, I2;
        Output O1;
};

class Nand2 : public Component
{
public:
        Nand2(Connector &, Connector &, Connector &,
              ckt_time = 1L, char* = "Nand2");
        void process(ckt_time);
private:
        Input  I1, I2;
        Output O1;
};

class And3 : public Component
{
public:
        And3(Connector &, Connector &,
             Connector &, Connector &,
             ckt_time = CKT_TIME_NULL, char* = "And3");
private:
        Input  I1, I2, I3;
        Output O1;
        Wire   w;
        And2   and2a, and2b;
};

class RS_Latch : public Component
{
public:
        RS_Latch(Connector &, Connector &,
                 Connector &, Connector &,
                 ckt_time = CKT_TIME_NULL, char* = "RS_Latch");
private:
        Input  R, S;
        Output Q, Qb;
        Nand2  nand2a, nand2b;
};
```

```
class Half_Adder : public Component
{
public:
        Half_Adder(Connector &, Connector &,
                Connector &, Connector &,
                ckt_time = CKT_TIME_NULL, char* = "Half_Adder");
private:
        Input  X, Y;
        Output S, C;
        Xor2    xor2a;
        And2    and2a;
};


#endif /* !COMPLIB_H_ */
```

```
/*************************************************************************
 * connect.h
 * -----------------------------------------------------------------------
 * Class declaration for the connector class.
 *
 *************************************************************************/

#if !defined(CONNECT_H_)         /* protect from multiple inclusion */
#  define CONNECT_H_

#include     <iostream>    /* For ostream declaration       */
#include     "signal.h"    /* Signal size req'd by Connector*/
#include     <list>        /* 'fan_out' is a linked list         */

class        Component;    /* Forward declaration for 'fan_out'     */

/*
 * Connectors are responsible for connecting components with one another
 * and the outside world. It is through connectors that the transmission
 * of signals occurs.
 */
class Connector
{
public:
       virtual              ~Connector();
       virtual      Signal  get_signal(ckt_time) const = 0;
                                                        // Retrieve signal
       virtual      void    send_signal(Signal) = 0;  // Generate signal
       virtual      void    show_signals() const = 0; // Display signals.

       virtual void  display(ostream &, int) const;  // Display info.

       void          connect(Component &);       // Add comp. to fan-out
       const char   *get_name() const;           // Return the name
       void          propagate() const;          // Simulate fanout cmps

protected:
       Connector(const char* = "Connector");     // Only derived
                                                 // connectors can be
                                                 // created.
private:
       std::list<Component *> fan_out;  // Components in fan-out
       char          *name;             // Name of connector.
};

#endif /* !CONNECT_H_ */
```

```
/***********************************************************************
 * port.h *
 * -------------------------------------------------------------------
 * Class declaration for port class and its input and output port
 * subclasses.
 *
 ***********************************************************************/

#if !defined(PORT_H_)      /* protect from multiple inclusion */
#  define PORT_H_

#include     <iostream>    /* For ostream declaration        */
#include     "sim.h"       /* For ckt_time typedef               */
#include     "connect.h"   /* Port is a derived class of Connector */

/*
 * Ports connect components with the outside world.
 */
class Port : public Connector
{
public:
        Signal          get_signal(ckt_time) const;// Get signal from ext.
        void            send_signal(Signal);       // Send sig. to ext.
        void            show_signals() const;      // Display ext. signs.
        void            display(ostream &, int) const; // Display port info.
protected:
        Port(Connector &, const char* = "Port"); // Only derived ports
                                                 // can be created.

        Connector     *external;    // External connector (port or wire) to
                                    // which the port is attached.
};

/*
 * Input ports can get signals from the external world but are unable to
 * interpret a send_signal() message.
 */
class Input : public Port
{
public:
        Input(Component &, Connector &, const char* = "Input");

        void   send_signal(Signal);  // Input ports can't send signals.
};

/*
 * Output ports can receive signals and send them, so inherit both
 * send_ and get_signal methods from Port.
 */
class Output : public Port
{
public:
        Output(Component &, Connector &, const char * = "Output");
};

#endif /* !PORT_H_ */
```

```
/***********************************************************************
 * signal.h *
 * ---------------------------------------------------------------------
 * Class declaration for the signal class.
 *
 ***********************************************************************/

#if !defined(SIGNALS_H_)         /* protect from multiple inclusion */
#  define SIGNALS_H_

#include      <iostream>
#include      "sim.h"            /* For ckt_time     typedef       */

using std::ostream;

enum Sig_Val {SIG_LOW, SIG_HIGH, SIG_X, SIG_NULL};

/*
 * Signal class declaration. Every signal in this implementation is
 * comprised of two entities: a signals value (e.g. SIG_HIGH, SIG_LOW,
 * SIG_X ...) and a time at which the signal occurred during the
 * simulation. The wire class is responsible for handling the signals.The
 * Signal class is essentially a write once/read many structure. With the
 * only write being done once by the constructor. Once a signal is set, it
 * cannot be changed, either accidentally or intentionally.
 */
class Signal
{
// Overload << operator for intuitive output of signals.
friend ostream &operator <<(ostream &, const Signal &);
public:
      Signal(ckt_time = CKT_TIME_INIT, Sig_Val = SIG_X);
      Signal(const Signal &signal);
      Signal &operator=(const Signal &sig);

      Sig_Val       get_value() const;  // get the signal's value.
      ckt_time      get_time() const;   // return the time of signal.

      void          set_value(Sig_Val);
private:
      Sig_Val       value;         // Value of the signal
      ckt_time      t;             // Time that the signal occurred.
};

#endif /* !SIGNALS_H_ */
```

```
/***************************************************************************
 * wire.h *
 * ------------------------------------------------------------------------
 * Class declaration for wire class.
 *
 ***************************************************************************/

#if !defined(WIRE_H_)            /* protect from multiple inclusion */
#  define WIRE_H_

#include      <iostream>    /* For ostream declaration        */
#include      "sim.h"       /* For ckt_time     typedef                  */
#include      "connect.h"   /* Wire is a derived class of Connector */
#include      "signal.h"    /* Size of Signal required by Wire       */
#include      <list>        /* 'signals' is a linked list            */

/*
 * Wires connect components with each other. They store a history of
 * signals <value, time> in addition to a linked list of components
 * inherited from Connector.
 */
class Wire : public Connector
{
public:
       Wire(const char* = "Wire");
       Wire::Wire(Signal s[], int num, char *nm);

       Signal get_signal(ckt_time) const;      // Retrieve a signal.
       void   send_signal(Signal);             // Put & propagate sig.
       void   show_signals() const;            // Show wire signals.
       void   display(ostream &, int) const;   // Show wire info.
       void   add_signal(Signal);              // Place initial
                                               // signals on a wire.
private:
       void            display_signals() const; // Dump signals on wire.
       void            replace(Signal);         // Replace a signal.
       std::list<Signal> signals; // History of all the signals which
                                  // have travelled along the wire.
};


#endif /* !WIRE_H_ */
```

```
/*************************************************************************
 * sim.h
 * ----------------------------------------------------------------------
 * General purpose header file containing typedefs and constants
 * which are useful to the simulator engine as a whole.
 *
 *************************************************************************/

#if !defined(SIM_H_)              /* protect from multiple inclusion */
#  define SIM_H_

/*
 * Define our typedefs, signal values and a few useful consts. This
 * header file should be included by most source files in the
 * implementation.
 */
typedef int  boolean;
#if !defined(TRUE)
      const  boolean TRUE = 1;
      const   boolean FALSE = 0;
#endif /* !TRUE */

typedef long ckt_time;

const  ckt_time CKT_TIME_INIT = -1;
const  ckt_time CKT_TIME_NULL = -2;

char   *generate_tabs(int);       // Defined in main.cpp

#endif /* !SIM_H_ */
```

APPENDICES C

OUTPUT

```
Output 1: Two-input AND Gate
Output 2: Three-input AND Gate
Output 3: RS-Latch
Output 4: Half Adder
```

OUTPUT 1

```
output:

        id: 2_input_AND

        values: {_ X} {? L} {0 H} {1 L} {6 H}

Press any key to continue
```

OUTPUT 2

```
3 Input AND Gate

---------------

output:

        id: Main out_w4

        values: {_ X} {1 L} {3 H}

Press any key to continue
```

OUTPUT 3

```
Reset-set Latch (RS-Latch)

--------------------------

output:

        id: Q

        values: {_ X} {1 H} {6 L} {7 L}

output:

        id: Qb

        values: {_ X} {2 L} {5 H} {7 H}

Press any key to continue
```

OUTPUT 4

```
HALF ADDER
----------
output:
        id: main out_S
        values: {_ X} {1 H} {3 L} {5 H} {9 L}
output:
        id: main out_C
        values: {_ X} {1 L} {3 H} {4 L} {9 H}


Press any key to continue
```

REFERENCES

1. Stephen R.Schach, 2005, *Object-Oriented and Classical Software Engineering*, McGraw-Hill.

2. Deitel, Harvey M, 1998, C++ How to Program, 2nd ed, Prentice Hall.

3. Gary J. Bronson, 2000, Program Development and Design Using C++, 2nd ed, Thompson Learning.

4. Thomas L.Floyd, 2003, *Digital Fundamentals* 8th ed,, Prentice Hall.

5. Allen Dewey, 1997, *Analysis and Design of Digital Systems With VHDL*, PWS Publishing Com.

6. Douglas Perry, 1999, *VHDL*, 3rd ed., Mc-Graw Hill.

7. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley.

8. Stephen C. Dewhurst, 1989, Kathy T. Stark, *Programming in C++*, Prentice-Hall.

9. M. Morris Mano, *Digital Design*, Prentice-Hall, 1984.

10. Behrouz A.Forouzan, 2004,Richard F. Gilberg, *Computer Science: A Structured Programming Approaching Using C++*, Thompson Learning.

11. Ron Waxman, ``Hardware Design Languages for Computer Design and Test," *IEEE Computer*, pp 90-97, Vol. 3, No. 2, April 1986.

12. Wayne H. Wolf, ``How to Build a Hardware Description and Measurement System on an Object-Oriented Programming Language," *IEEE Transactions on Computer-Aided Design*, pp 288-301, Vol. 8, No. 3, March 1989.

13. Wayne H. Wolf, ``A Practical Comparison of Two Object-Oriented Languages," *IEEE Software*, pp 61-68, Vol. 6 No. 5, September 1989.

14. http://www.cse.sc.edu/~jimdavis/Tools/exsedia/nimbus_asm_models_.htm