Circuit Description and Elementary Hierarchical Circuit Simulation Using C++ and the Object-Oriented Programming Paradigm

Donald C. Craig

Department of Computer Science

Memorial University of Newfoundland

St. John's, Newfoundland

Canada, A1C 5S7

A dissertation submitted to the Faculty of Science

in partial fulfillment of the requirements

for the degree of

Bachelor of Science (Honours)

April 1991

Acknowledgements

I would like to thank Dr. Paul Gillard for his patience and guidance during the course of this dissertation and for introducing me to an improved foundation upon which to base circuit simulation.

Thanks also go to Dr. Karl Posch whose enlightening discussions often forewarned me of the finer points of circuit description and simulation which I might otherwise have overlooked.

I am also indebted to Elaine Boone, whose proofreading skills helped eliminate most, if not all, of the inconsistencies and grammatical errors present in an earlier draft of this report.

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. In addition, a method for modelling and simulating hardware via the construction and traversal of a three-dimensional graph over time will be addressed. This method of simulation will challenge the techniques traditionally used by many hardware digital simulators as the concept of time is distributed over the modules being simulated; all time is local to the modules themselves.

Contents

List of Figures

Chapter 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.

Using general purpose languages for hardware description is by no means a novel concept. Several attempts have been made to use languages such as LISP [10], Flavors [21], LOOPS [2], Mainsail [4],[15] and even Ada [7] for the purpose of hardware description and simulation. The underlying theme of nearly all these attempts is to treat hardware devices as a collection of abstract, interconnected objects which pass messages, in the form of signals, to one another. This, of course, is a very natural and intuitive way to think about hardware and, as will be discussed later, the very heart of the object-oriented programming paradigm.

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.

The several languages mentioned satisfy most, but not all, of these requirements. Languages such as LOOPS and Flavors are LISP-like in their structure and are cumbersome and unintuitive to use due to syntactic deficiencies, such as the gratuitous use of parenthesis. Also, these languages appear to lack a standard which is acceptable by all. Other high-level languages commonly used to describe hardware, such as Ada, are more standardized and also somewhat more popular. However, these languages are usually object-based rather than object-oriented. This report will attempt to demonstrate that languages which provide support for object-oriented programming are very amicable to the description and simulation of computer hardware.

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.

This dissertation is separated into three major sections dealing with C++ and the object-oriented paradigm, circuit description and circuit simulation using C++. 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. The remainder of the dissertation will discuss ways to 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. In addition, an argument will be made against the traditional approach of using a so-called global event queue for simulating hardware and an attractive alternative, namely the concept of a distributed event queue, will be introduced and discussed.

Chapter 2

C++ and Object-Oriented Programming

Prior to a discussion of the use of C++ as a language for circuit description and simulation, one must understand the basic mechanics of the C++ language and, in particular, how this language supports the object-oriented paradigm [5], [11], [20]. Since this programming methodology is not yet as popular or as well known as the structured programming paradigm, it is felt that devoting a chapter to object-oriented programming and C++ will enhance one's understanding of how this paradigm and the language itself can benefit the areas of hardware description and simulation. Many of the topics discussed in this chapter will be used extensively in the following chapters on hardware description and simulation; therefore it is important that these features be clearly addressed.

2.1 The C++ Programming Language

This section will give a brief discussion of the C++ language; where it came from and where it is going. Other languages which influenced the creation and development of C++ are discussed. Several of the features which make C++ different and in many ways better than the C language are also mentioned.

2.1.1 C++ Background and History

C++ is a programming language developed at AT&T Bell Laboratories by Bjarne Stroustrup [17] 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 [6]

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. Comparing C to C++ is like comparing checkers to chess. Although both games are played on the same board, one must realize that the game of chess cannot be played using the same strategy as that used in the game of checkers. 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.

2.1.2 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. Apart 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. As mentioned, the newer C++ compilers will provide support for parameterized types and exception handling, concepts borrowed from Ada and Clu.

2.1.3 C++ Versus C

To demonstrate the distinctions between C and C++, numerous features supported by C++ which have no equivalent in C are provided below. Some of these features will be discussed in more detail in the section on object-oriented programming.

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. These features are discussed in detail in the next section.

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 [1] 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 [1] (See Figure 2.1).

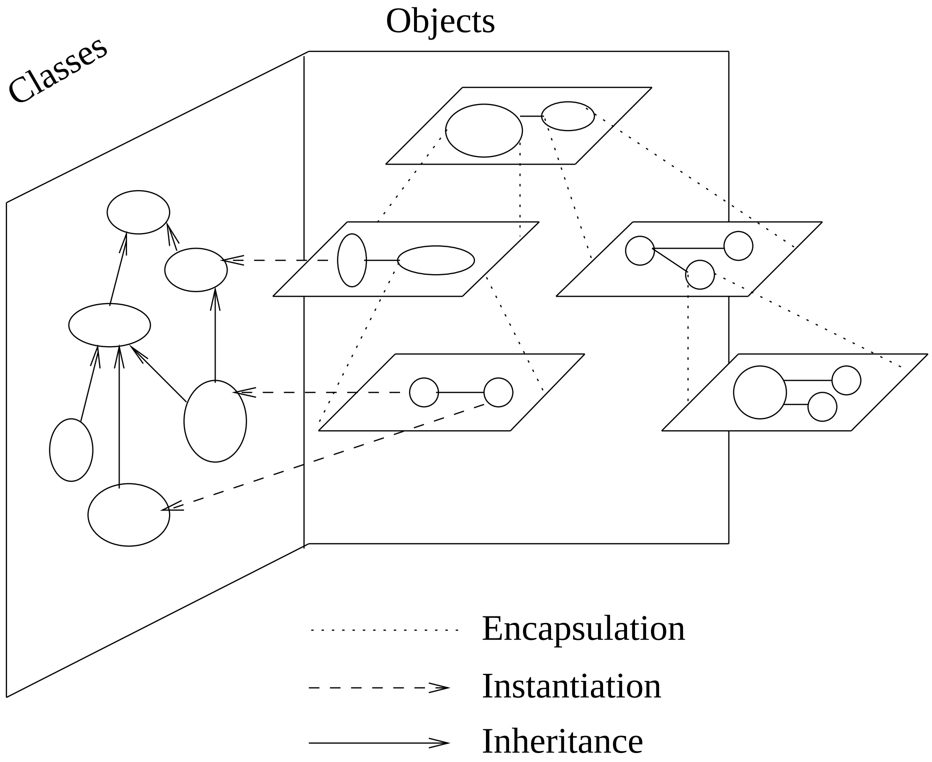

Figure 2.1: The Problem of Complexity

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* which will be discussed in detail later.

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.

Examples of these two types of hierarchies will be discussed later when we attempt to describe hardware using these relationships.

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.

This is a problem faced by other members of the scientific community as well. For example, biologists have traditionally divided living beings into two classes: plants and animals; every living entity must belong to one, and only one, of these classes. However, when the *euglena*, a single-celled being with chloroplasts, was discovered, it seemed perfectly legitimate to place it in both classes; hence it defied classification. Also, viruses cannot accurately be portrayed as either a plant or animal; so they too defy classification under the plant/animal taxonomy.

2.3 Features of Object-Oriented Programming Languages

There is no formal definition of object-oriented programming. Hence there is some confusion surrounding what features a programming language must support in order to claim that it is object-oriented. Despite this, however, most agree that in order for a language to claim that it is object-oriented, it must provide support for three major concepts, as inferred from Booch's definition.

- Data encapsulation or data abstraction
- Inheritance or derivation
- Dynamic or runtime binding

The following subsections will explain these features and show how C++ provides support for them through its concept of a *class*; the underlying mechanism upon which all good C++ programs are based. Examples of how these features can be used will be seen later in the dissertation.

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. For example, to create a stack class which can contain integers, the designer may choose to implement it with an array, which is hidden from the user of the class. The designer then writes the `push()` and `pop()` methods which puts integers into the array and removes them from the array respectively. These methods are made accessible to the user. Should an attempt be made by the user to access the array directly, a compile time error will result. Now, should the designer decide to change the stack's implementation to a linked list, the array can simply be replaced with a linked list and the `push()` and `pop()` methods rewritten so that they manipulate the linked list instead of the array. The code which the user has written to manipulate the stack is still valid because it was not given direct access to the array to begin with.

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. For example, consider a `graph` class which is represented by edges and vertices and some (abstract) method of traversal. Next, consider a `tree` class which is a special form of a graph. We can simply derive `tree` from `graph` and the `tree` class automatically inherits the concept of edges, vertices and traversal from the `graph` class. We can then restrict how edges and vertices are connected within the `tree` class so that it represents the true nature of a tree.

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. Examples of inheritance are presented later.

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*.

Going back to our previous example, we may decide to derive two tree classes from our `graph` class, the first class, `in_order_tree` would be traversed in an "in order" fashion when it received a `traverse()` message, whereas `post_order_tree` would be traversed in a "post order" manner. The different traversal algorithms could be incorporated into a dynamically bound `traverse()` method. Now, when one of these trees is passed to a function which accepts a reference to a `graph` class, the invocation of the `traverse()` method via the `graph` parameter would call the correct traversal algorithm at runtime depending upon which tree was passed to the function. This reduces the burden on the programmer since a tag does not have to be associated with each class derived from `graph` to distinguish it from other graphs. In addition, the programmer would not have to maintain an unwieldy `switch` statement to determine which traversal algorithm to invoke since this is all handled automatically by the compiler.

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 [18] 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."

As will be shown in the next two chapters, the object-oriented paradigm can be very easily applied to the problem of hardware description and simulation.

Chapter 3

Hardware Description Using C++

This chapter will explore one possible way to describe hardware using C++. The first matter to be addressed will be the problem of classification. Once the major classes have been established and described in detail, methods of expressing connectivity will be examined and a technique for describing hardware in a hierarchical fashion will be presented. Since the C++ code required to describe hardware in this manner is relatively small, relevant portions of the actual source will be presented, accompanied with annotations explaining the particular methods. Source code which is specific to circuit simulation will be presented in the next chapter. At the end of this chapter, several examples are presented to demonstrate how more complex devices can be created from smaller ones via encapsulation.

3.1 Classification of Circuit Components

When one thinks of circuit elements, a myriad of objects immediately comes to mind. These devices include wires, AND gates, transistors, input and output ports, RS-latches, HIGH and LOW signals and so on. 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. As will be shown later, a list of signals 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. Since signals are used almost exclusively during the simulation of circuitry, a detailed analysis of this class will be postponed until the next chapter.

As alluded to above, linked list classes are required for components and signals. As will be shown in the next section, 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.

3.1.1 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. Note, however, that the operations that can be performed on this linked list are limited by the public interface of the `Port_List` class, which is discussed later.

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. This concept of local time will be further explained in the next chapter. 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, `process()` and `simulate()` are discussed in greater detail in the next chapter.

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 final class declaration for `Component` looks as follows:

```
class Component
{
public:
        Port_List       I_List;
        Port_List       O_List;
        virtual void    process(ckt_time);
        void            simulate();
protected:
        Component(ckt_time = UNDEF_TIME, char* = "Component");
        ckt_time        delay;
private:
        char            *name;
        ckt_time        local_time;
};
```

Note that the `Component` constructor accepts two parameters, a `ckt_time` argument, which is used to initialize the delay of the component (ckt_time is simply typedef'fed to long), and a char * which is used to initialize the name of the component. Both of these arguments have default values, so the derived class does not have to call this constructor explicitly when constructing its Component base. The constructor for Component is simply:

```
Component::Component(ckt_time t, char *nm) :
        delay(t), name(nm), local_time(INIT_TIME)
{ }
```

This constructor assigns the arguments t and nm to data members delay and name respectively. The local_time data member is set to some constant value representing initial time. Note that the I_List and O_List data members are initialized implicitly by the Port_List default constructor, as discussed later in this chapter.

As is the convention among hardware designers, boxes will be used to represent components in the diagrams following in this report.

The Component_List Class

The Component_List class, as used later, maintains a linked list of components which may be present in the fan-out of a connector. The definition of the class is as follows:

```
class Component_List
{
public:
        Component_List();
        void            add(Component *);
        boolean         is_empty();
        void            propagate();
```

```
private:
        Component_Node  *comp_list;
};
```

This class contains a public constructor which has no parameters (a *default constructor*). The constructor sets the pointer to the head of the list, `comp_list`, to null.

```
Component_List::Component_List() :
        comp_list(0)
{ }
```

The `add()` method accepts a pointer to a component and simply adds it to the linked list of component pointers.

```
void
Component_List::add(Component *c)
{
        Component_Node       *nc = new Component_Node(c);

        nc->next = comp_list;
        comp_list = nc;
}
```

The `Component_Node` class is simply a class which contains a pointer to a component and a pointer to the next element in the list. All the data elements in this class are made private. `Component_List` is made a so-called `friend` of `Component_Node` so that `Component_List` is the only class which may access the members of a `Component_Node` class.

The `is_empty()` and `propagate()` methods, being related more closely to circuit simulation, are discussed in the next chapter.

3.1.2 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, namely 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. These derived classes will be covered in a later section. The class declaration for Connector is as follows:

```
class Connector
{
public:
        Connector(char* = "Connector");
        void            connect(Component *);
        virtual Signal  get_Signal(ckt_time) = 0;
        virtual void    send_Signal(Signal) = 0;
protected:
        Component_List  fan_out;
        char            *name;
};
```

The constructor for Connector trivially assigns the character pointer passed in to its protected name data member:

```
Connector::Connector(char *nm) :
        name(nm)
{ }
```

The connect() method is responsible for taking a pointer to a component and adding it to the fan-out of the connector.

```
void
Connector::connect(Component *cmp)
{
        fan_out.add(cmp);
}
```

The `get_Signal()` and `send_Signal()` virtual methods are related to the transmission of signals which occurs during simulation, and are therefore discussed in the next chapter. These two virtual methods are defined as *pure virtual functions* by assigning them to zero. Doing this prevents an object of type `Connector` from being instantiated in the code. Note that this does not preclude the creation of objects from classes derived from `Connector`, just as long as they override *both* virtual methods in their class declaration.

Since a connector can supply signals to one or more components, the `Connector` class is responsible for maintaining a linked list of pointers to components, referred to by the `fan_out` data member. Since the `Wire` and `Port` class require access to the fan-out of the connector, `fan_out` is made a protected member of the class. The `Component_List` class which is used to instantiate `fan_out` was described in the previous section.

3.1.3 The `Wire` Class

As mentioned, the `Wire` class is derived from the `Connector` class. In our model, 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 `Signal` and `Signal_List` classes are described later in the next chapter.

The `Wire` class itself is as follows:

```
class Wire : public Connector
{
public:
        Wire(char* = "Wire");
        Wire(Signal s[], int, char* = "Wire");
        virtual Signal  get_Signal(ckt_time);
        virtual void    send_Signal(Signal);
        void            display();
private:
        Signal_List     signals;
};
```

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.

```
Wire::Wire(char *name) :
        Connector(name)
{
        signals.add(Signal(X, INIT_TIME));
}
```

The second constructor is used to actually place signals onto the wire. This constructor is invoked when a user wants to create a wire which has a series of test inputs which will later be "hooked-up" to a circuit. The signals are stored in an array of signals of size `num`. Its constructor is as follows:

```
Wire::Wire(Signal s[], int num, char *name) :
        Connector(name)
{
        signals.add(Signal(X, INIT_TIME));
        for (int i = 0; i < num; i++)
                signals.add(s[i]);
}
```

Note that in both cases the default constructor for `Signal_List` is implicitly called to initialize the linked list before either of the `Wire` constructor bodies are executed.

The `get_Signal()` and `send_Signal()` virtual methods inherited from `Connector` are redefined by this class. Doing this makes it possible to instantiate objects of type `Wire`. These two methods are discussed in the next chapter.

The final method in the `Wire` class is `display()` which provides the user with a means of seeing what values are currently in the wire. This method sends a message to the signal list asking it to dump all it signal values to the standard output.

```
void
Wire::display()
{
        cout << "Wire: " << name << "\n";
        signals.dump();
}
```

Wires will be represented by elongated rectangles in the diagrams of this report.

3.1.4 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 declaration of the `Port` class is as follows:

```
class Port : public Connector
{
friend  class Port_List;
public:
        virtual Signal  get_Signal(ckt_time);
        virtual void    send_Signal(Signal);
protected:
        Port(Component *, Connector &, char* = "Port");
        Connector       *external;
};
```

The constructor accepts a pointer to the component which encapsulates the port, a reference to the connector that feeds the port and an optional name parameter. It then passes the `name` argument to its base class, `Connector`; assigns the address of the feeding connector to the protected `external` data member and finally adds the component to the fan-out list of the feeding connector.

```
Port::Port(Component *cmp, Connector &con, char *name) :
        Connector(name), external(&con)
{
        con.connect(cmp);
}
```

Note that as in the `Wire` class, the `get_Signal()` and `send_Signal()` methods have been redefined by this class. Their definitions are explained in detail in the next chapter.

The `Port_List` class, described in the next section, is made a `friend` of the `Port` class. This is done so the methods of `Port_List` can efficiently access the fan-out of the `Port` class during the simulation. This will be explained in more detail in the next chapter.

Diagrams portray ports as small circles embedded within the boxes representing components. Where possible, input ports are placed on the left of the box, whereas output ports are usually placed on the right side of the box.

The `Input` and `Output` Class

The `Input` and `Output` classes are almost the same with just one minor difference, as can be seen from the following class declarations:

```
class Input : public Port
{
public:
        Input(Component *, Connector &, char* = "Input");
        virtual void    send_Signal(Signal);
};

class Output : public Port
{
public:
        Output(Component *, Connector &, char* = "Output");
};
```

`Input` inherits `Port`'s virtual `get_Signal()` method, but defines its own `send_Signal()` method. This is because a signal cannot be sent out through an input port. Therefore, this method outputs a warning whenever an attempt is made to send a signal from an input port. `Output`, however, inherits both virtual methods from `Port`, so it is possible to both send a signal to and get a signal from an output port. This is useful to model bidirectionality.

The constructors for the two ports are trivially implemented. The `Port` base constructor is first called with the appropriate arguments and the I/O port is added to the corresponding `Port_List` of the component.

```
Input::Input(Component *cmp, Connector &con, char *name) :
        Port(cmp, con, name)
{
        cmp->I_List.add(this);
}
```

```
Output::Output(Component *cmp, Connector &con, char *name) :
        Port(cmp, con, name)
{
        cmp->O_List.add(this);
}
```

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. As mentioned earlier, this similarity could be exploited using generic types or templates. Since C++ does not yet support templates, some code replication will be necessary.

The constructor and the `add()` methods are identical to the corresponding methods of the `Component_List` class with the exception that a pointer to a port rather than a pointer to a component is added to the list. The `Port_List` class is defined as follows:

```
class Port_List
{
public:
        Port_List();
        void            add(Port *);
        boolean         inputs_are_ready(ckt_time);
        boolean         is_lowest_level();
        void            descend();
private:
        Port_Node       *prt_list;
};
```

The default constructor is automatically called to initialize `I_List` and `O_List` when a `Component` object is constructed.

The other three methods, `inputs_are_ready()`, `is_lowest_level()` and `descend()` are specific to the simulation of the circuit and not to its description. As such, elaboration upon these methods will be deferred until the next chapter.

3.2 Describing Hardware

This section will describe how one can use the above classes to actually build useful hardware components. A brief outline of the general technique used to build components is presented. Examples demonstrating how to express connectivity and the hierarchical nature of hardware are also described.

3.2.1 The Method

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:

```
#define CONNECT(Port, Wire, name)    Port(this, 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.

3.2.2 Examples

This section will demonstrate how to build a two-input AND gate component using the classes described previously. Then, using two two-input AND gates and a wire, a three-input AND gate will be built to demonstrate how smaller components can be encapsulated together to form larger, more complicated components. Following that, an RS-latch will be described to demonstrate how components with feedback can be designed. These last two examples show how hierarchical descriptions can be achieved using encapsulative techniques.

Two-Input AND Gate

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. 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;
};
```

As required, 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 constructor for the two-input AND gate is presented below. 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.

```
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")
{ }
```

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. Describing the functionality of fundamental circuit components is presented in the next chapter.

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 3.1 is produced.

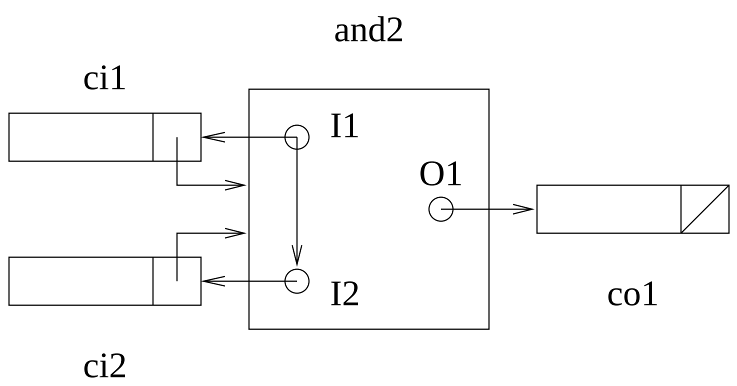

Figure 3.1: Two-Input AND Gate with External Wires

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 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;
};
```

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.

```
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")
{ }
```

Figure 3.2 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.

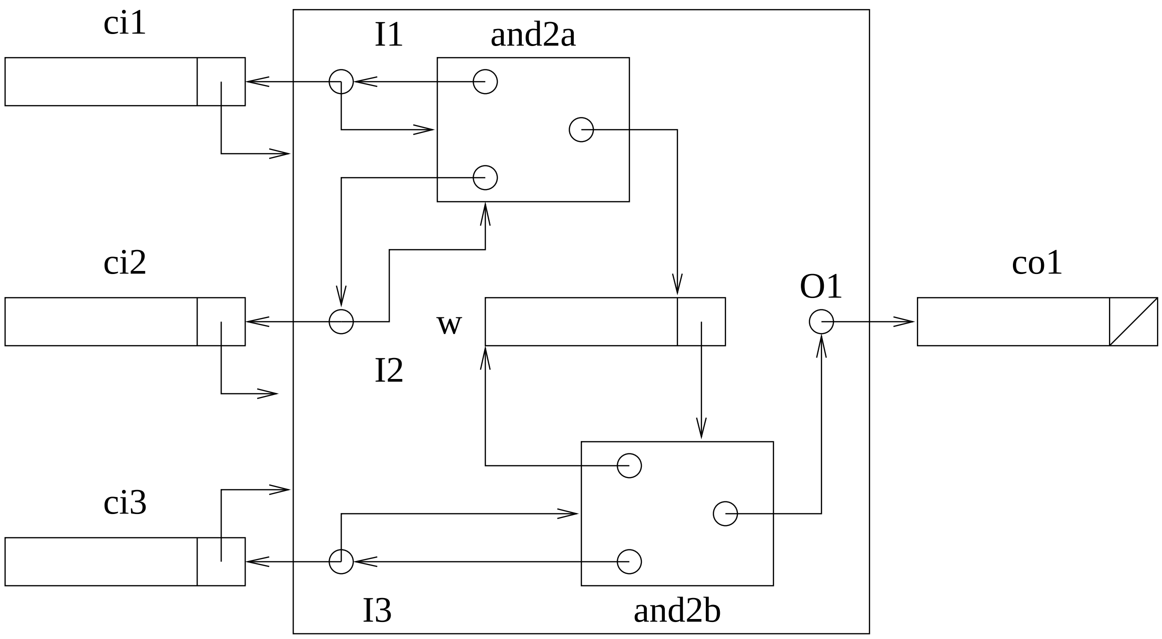

Figure 3.2: Three-Input AND Gate with External Wires

RS-Latch

The following code illustrates how an RS-latch can be described. 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;
};
```

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.

```
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")
{ }
```

A diagram showing the interconnectivity of the components within the RS-latch is presented in Figure 3.3. Again, the pointers connecting the input and output ports are omitted for clarity. Note how the feedback is achieved.

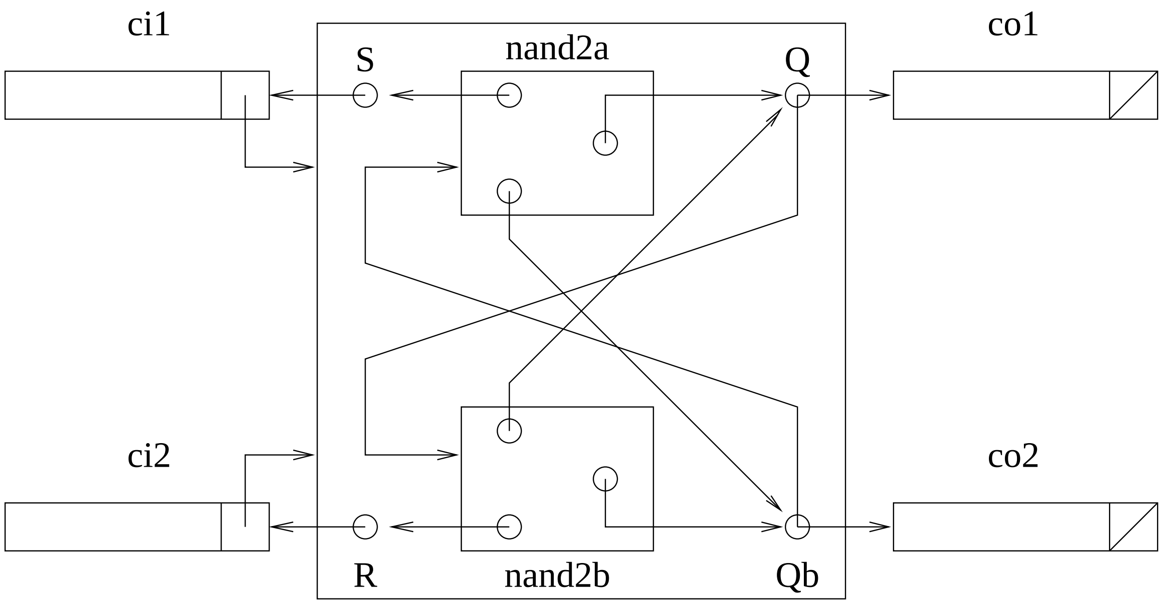

Figure 3.3: RS-Latch with External Wires

Chapter 4

Hardware Simulation Using C++

The problem of simulating hardware originates from trying to mimic the inherent parallelism of circuitry using programming languages which are sequential in nature and design. This chapter will briefly examine the traditional approach to implementing a circuit simulator and introduce a new method of simulating hardware which takes advantage of the features offered by the object-oriented programming paradigm. It will be shown how this new technique of simulation attempts to rectify some problems associated with the classical method of simulating digital devices.

This chapter will examine a new way of treating time during simulation. An algorithm which implements the new simulation technique will also be examined and the `Signal` class will be analyzed in detail. This method of simulation has also been used in various forms in other simulators [7]. A few examples describing how the lowest level components should be specified so that they will simulate correctly in the implementation will also be presented.

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. This section will look at two methods of treating time and how that concept of time is manifested in the implementation. This section will explain the deficiencies of the first method and how the second method attempts to correct these problems.

4.1.1 The Global Event Queue

The traditional approach to simulating parallel devices is to treat all components as being at the same time with respect to a so-called *global event queue* [9]. This structure basically consists of a linked list of time nodes, with each node containing a pointer to a linked list of components, or events, which have to be processed at that time. All events which are linked to the same time node are essentially in parallel with one another. When a component produces an output, all the components which have an input that depends on the output are placed on the global event queue at the node representing the time that the output signal was produced. As the time nodes are traversed sequentially, the components connected to each of the nodes are simulated and more events are scheduled later in time, reflecting the output of these components. This continues until the end of the timing queue is reached or the time allotted for the simulation expires.

Even though many simulators are built based upon the concept of a global event queue [3],[10], [16]; there are some fundamental problems associated with these data structures and, in particular, how they are implemented using structured programming languages.

A global event queue is, by its very name, a *global* data structure. As such, it is accessible everywhere in the program. Since structured programming languages do not support abstract data types or data encapsulation, there is nothing to prevent a user from accidentally (or intentionally) corrupting this queue. Even though the designer of the queue may have written a clean interface consisting of functions which should be called in order to modify the queue, the user may still be permitted access to the implementation of the structure directly, thereby risking abuse. If the queue is corrupted, then *every* component in the simulation is affected and the simulation is rendered invalid.

Since the time queue is sequential in nature, once a time window is passed, it can be very cumbersome to go back. Going back in time may be necessary should it be discovered that an asynchronous signal entered the system at some time prior to the current time of the simulation. Even if it were possible to traverse the queue backwards, one has to realize that because this queue applies to every single component, *all* components go back in time as well. This could be very disconcerting if the asynchronous signal affected only a small portion of the circuit.

The concept of a global event queue is also somewhat unintuitive; it has no equivalent whatsoever in the actual hardware. This means that a global queue does not correspond to the way people normally think about hardware. It will be shown in the next section that the use of distributed event queues in the form of wires makes for a more intuitive approach to hardware simulation.

4.1.2 Distributed Event Queues

This section 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 and wires essentially perform the same function and are so closely related, the concept of a distributed queue seems much more attractive and intuitive than a cumbersome global event queue.

As will be discussed later, in order for the distributed queues to be useful, each component must maintain its own concept of local time. Then, when the time comes for the component to simulate, it first queries all its input wires to determine if they have a signal which occurred at the local time of the component. If all the input wires do have signals waiting, then the component moves forward in time and then starts to simulate. It is through this mechanism that the concept of parallelism is mimicked. This algorithm is discussed in more detail later.

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.

How these distributed queues are used by the simulation algorithm is described later in this chapter. However, before the algorithm and implementation of the circuit simulator are presented, the `Signal` class will be examined in detail. Particular attention will be paid to how signals are stored and how they are transferred from one component to another via the `Wire` class and `Port` class.

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;
};
```

Note that `Sig_Val` is implemented as an enumerated type. The signal constructor trivially assigns a signal value and a time to the hidden members of the class.

```
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, and its usefulness will become apparent when the `process()` method for a component is described later in this chapter.

```
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, however, 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);
}
```

Note that the last call appears to be recursive with no terminating condition. However, it must be remembered that `get_Signal()` is a virtual function, and therefore is bound at runtime depending upon whether `external` is a pointer to a wire or a pointer to another port. As such, the call to `get_Signal()` in `Port::get_Signal()` will recurse only if `external` points to another port. Otherwise, it will call `Wire::get_Signal()` meaning that `external` points to a wire. Since every port must point directly or indirectly to a wire, it is guaranteed that this "recursion" will terminate.

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();
}
```

The details of the `propagate()` function are presented in the next next section along with the entire simulation algorithm.

4.3 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 [4].

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();
        }
}
```

The simulation algorithm basically amounts to the traversal of a three-dimensional graph much like that shown in Figure 2.1. The algorithm descends the graph of encapsulated components until it reaches a component that knows how to process its inputs (a fundamental component). When that component produces an output to a distributed queue, it is the responsibility of the queue to propagate that signal to all the components in its fan-out list. If one of the components was waiting on that input, it too will traverse through its graph of encapsulated components, looking for its base components. When control reaches a point where the components can simulate no further due to the lack of other inputs, control is eventually returned back to the component that sent the original signal. Some actual simulations are traced in Appendix A to demonstrate how the algorithm operates.

4.4 Redefining the virtual `process()` function

One problem with implementing simulators using structured programming languages has to do with determining which function to call to simulate a component. In some languages, such as Pascal, this means resorting to tag fields in records and using huge `case` statements which have to be modified each and every time a component is added or deleted from the library. C allows a slightly cleaner solution through the use of pointers to functions. However, this is still deficient since the user must remember to assign the address of the function which processes the inputs to the function pointer of the component structure. This may not seem like a major drawback; however, if one built a circuit using a thousand components and one function pointer assignment was missed, the simulation would likely terminate in error. The user would then have to check through the code making sure that every function pointer assignment was made. If the user were truly unlucky, the program would execute smoothly but produce erroneous simulation results leading the user to think that the circuit design was incorrect.

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. As can be seen above, this makes the `process()` method look much more intuitive.

The designer may perform other actions within the `process()` method to make the simulation more realistic. (For example, the user may decide to modify the delay time or other parameters internal to the component to simulate say, an increase in resistance due to local heating).

Notice that when a new circuit component is added to the library, no change whatsoever has to be made to the simulation code. If the component is a low-level component then its behaviour is supplied by simply defining the virtual `process()` method for its class. This makes the library of circuit components much easier to maintain. If the component is composed of subcomponents, then all the designer needs to do is to define how the subcomponents are connected using the enclosing component's constructor.

Chapter 5

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 modelling *and* testing of hardware designs, the language creates a uniform environment for hardware description and simulation.

The concept of distributed event queues in the form of wires as opposed to a global event queue seems more natural and less prone to error. The distributed event queue also appears to rectify most of the problems associated with the unintuitive global event queue. The use of distributed queues in conjunction with the concept of local time for hardware modules is a feasible method for simulating inherently parallel devices using a sequential language. It should be noted that while this dissertation dealt exclusively with the description and simulation of hardware, the concept of distributed queues and local time may also be used to simulate hierarchical software modules in parallel as well.

The code written to date appears to simulate relatively small combinational and sequential components correctly; however, more extensive testing is required. Further work required on the current implementation includes providing support for the dequeuing of signals from wires when they have been used by all the relevant components. Support should also be provided for signals which are generated spontaneously during a simulation such as the internal clocks or resetable timers and one-shots. In addition, the creation of a `Bus` class would make circuit descriptions more powerful and less cumbersome.

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. As was shown earlier, 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.

Appendix A

Sample Simulations

This appendix will trace through some sample simulations to demonstrate how the algorithm works. In the following diagrams, the symbol '*' represents the initial time. To prevent confusion, 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. For simplicity, we assume inputs to be available at integer times $t = 0, 1, 2, \ldots$

A.1 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 A.1. Recall that the `Wire` constructor places an `X` as the initial signal of all wires.

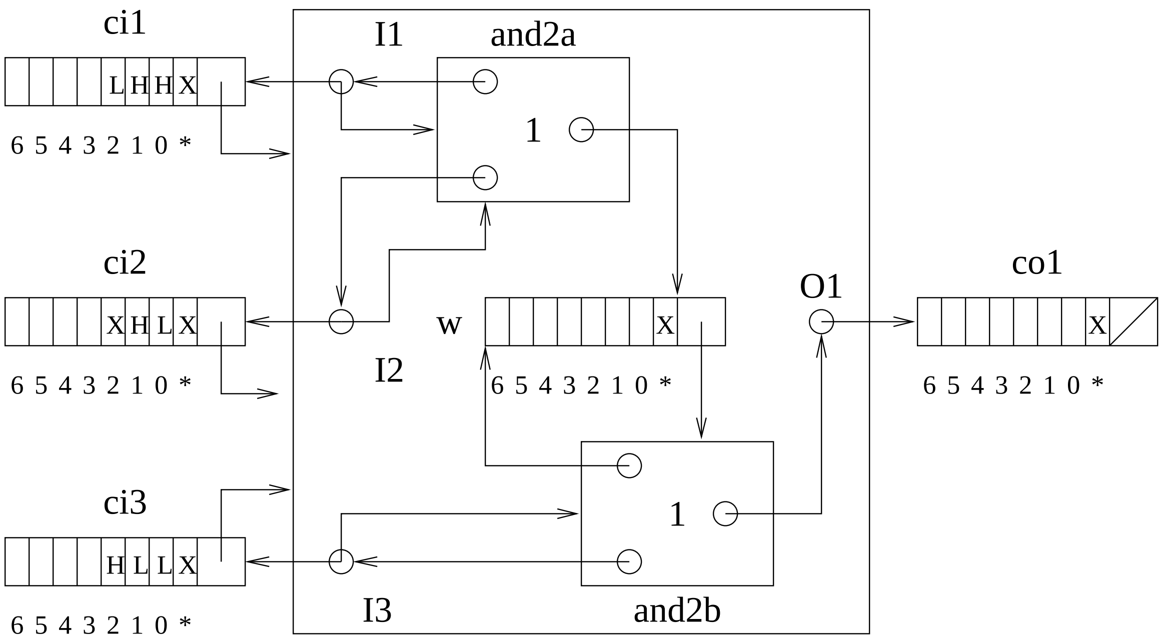

Figure A.1: Three-Input AND Gate Before Simulation

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 A.2.

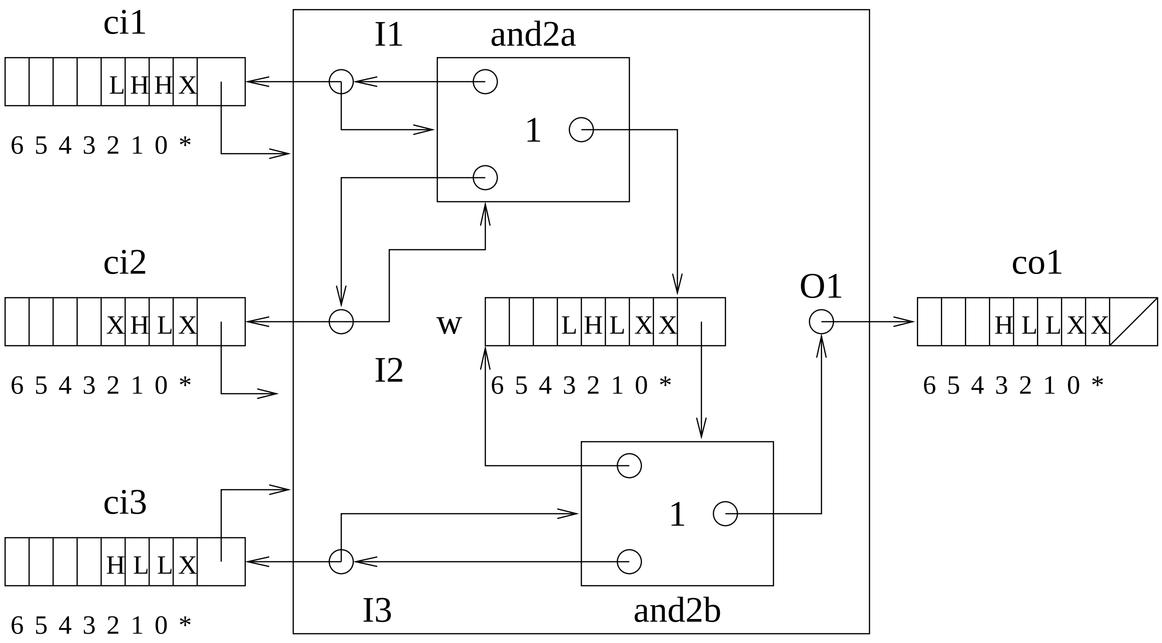

Figure A.2: Three-Input AND Gate After Simulation

A.2 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 A.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.

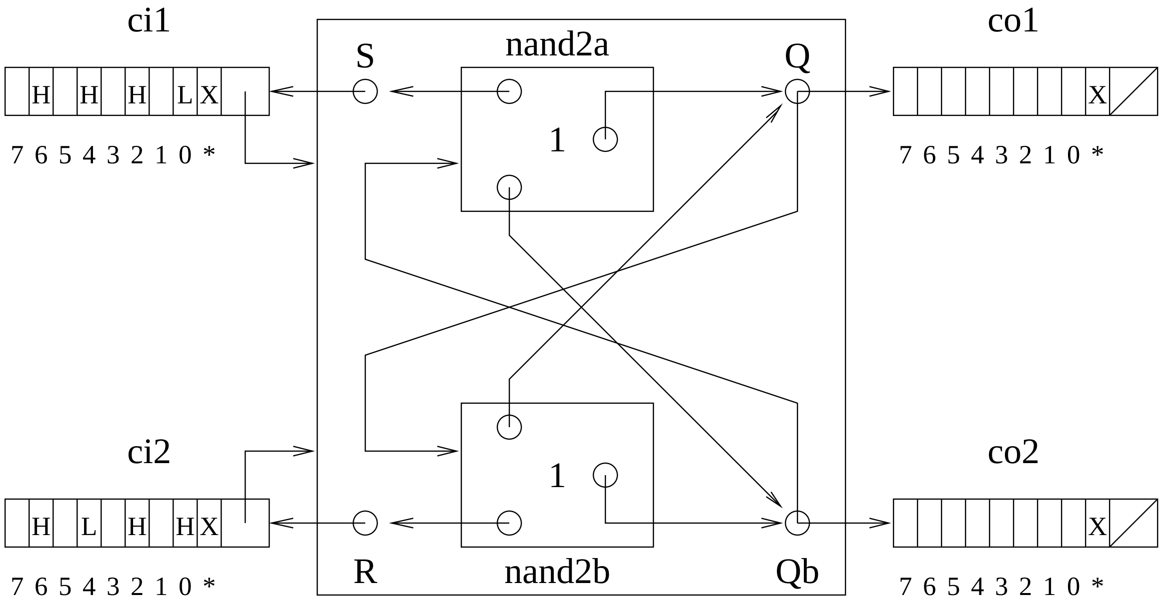

Figure A.3: RS-Latch Before Simulation

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 A.4.

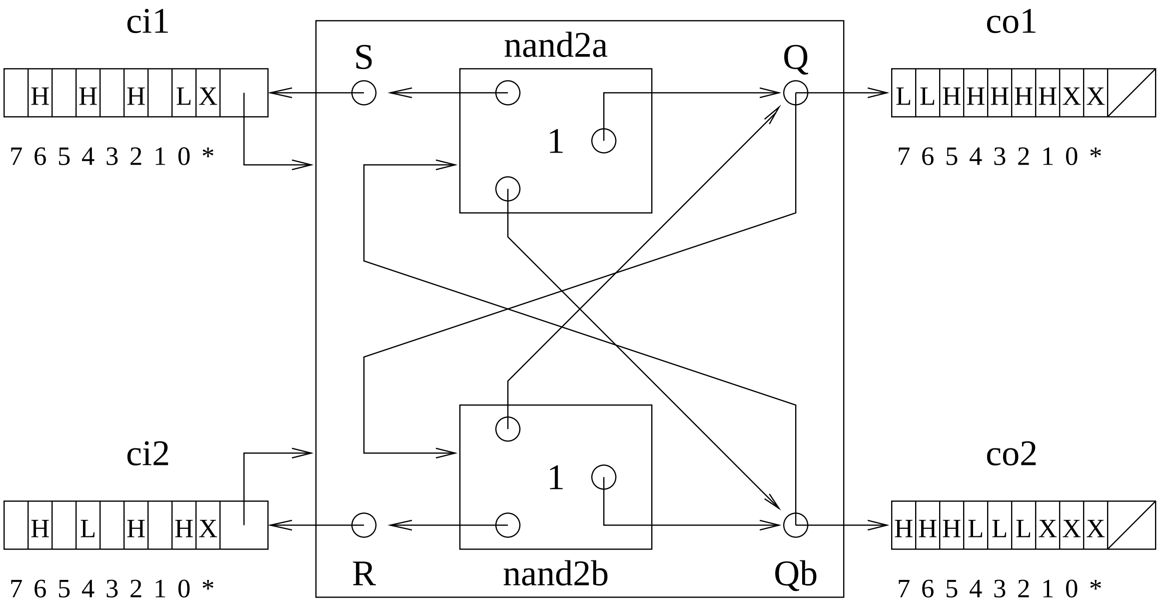

Figure A.4: RS-Latch After Simulation

The reader can easily verify that this model exhibits the characteristics of an RS-latch, including the "race condition," when the appropriate inputs are provided.

Bibliography

[1] Grady Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings Publishing Company, Inc., 1991.

[2] Harold Brown, Christopher Tong, Gordon Foyster, "Palladio : An Exploratory Environment for Circuit Design," *IEEE Computer*, pp 41–56, Vol. 16 No. 12, December 1983.

[3] Manuel A. d'Abreu, "Gate-Level Simulation," *IEEE Design & Test*, Vol. 2, No. 6, December 1985.

[4] J. Cloutier, M. Bourgault, S. Fauvel, C. Roy, E. Cerney, J. Gessei, "An Object-Oriented Mixed-Mode Hierarchical VLSI Simulator," Dept. d'information et de recherche operationnelle, Université de Montreal, 1986.

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

[6] Margaret A. Ellis, Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[7] Sumit Ghosh, "Using Ada as an HDL," *IEEE Design & Test of Computers*, pp 30–42, Vol. 5, No. 1, May 1988.

[8] Rajiv Gupta, Wesley H. Cheng, Rajesh Gupta, Ido Hardonag, Melvin A. Breuer, "An Object-Oriented VLSI CAD Framework : A Case Study in Rapid Prototyping," *IEEE Computer*, pp 28–37, Vol. 22, No. 9, May 1989.

[9] Sean M. Hogan, "YADIS–1 — An Implementation," Technical Report #8906, Department of Computer Science, Memorial University of Newfoundland, St. John's NF, Canada, A1C 5S7, 1989.

[10] Richard H. Lathrop, Robert S. Kirk, "An Extensible Object-Oriented Mixed-Mode Functional Simulation System", *Proceedings of the* 22^{nd} *Design Automation Conference*, June 1985.

[11] Stanley B. Lippman, *C++ Primer*, Addison-Wesley, 1989.

[12] Roger Lipsett, Erich Marschner and Moe Shahdad, "VHDL – The Language," *IEEE Design & Test*, pp 28–41, Vol. 3, No. 2, April 1986.

[13] M. Morris Mano, *Digital Design*, Prentice-Hall, 1984.

[14] Bryan Preas, Michael Lorenzetti (eds), *Physical Design Automation of VLSI Systems*, Benjamin/Cummings Publishing Company, Inc., 1988.

[15] C. Roy, L.P. Demers, E. Cerney, J. Gessei, "An Object-Oriented Switch Level Simulator," *Proceedings of the* 22^{nd} *Design Automation Conference*, June 1985.

[16] Steven M. Rubin, *Computer Aids for VLSI Design*, Addison-Wesley, 1987.

[17] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[18] Bjarne Stroustrup, "What is Object-Oriented Programming?" *IEEE Software*, Vol. 5, No. 3, May 1988.

[19] Ron Waxman, "Hardware Design Languages for Computer Design and Test," *IEEE Computer*, pp 90–97, Vol. 3, No. 2, April 1986.

[20] Richard S. Wiener, Lewis J. Pinson, *An Introduction to Object-Oriented Programming and C++*, Addison-Wesley, 1988.

[21] 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.

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