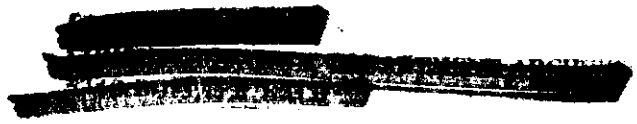
CMU-CS-85-104

Ada as a Hardware Description Language: An Initial Report

Mario R. Barbacci[1],
Steve Grout[2],
Gary Lindstrom[3],
Mike Maloney[3],
Elliott Organick[3],
Don Rudisill[2]

8 December 1984



1. Carnegie-Mellon University, 2. Martin Marietta Corporation, 3. University of Utah.

The work reported in this report was sponsored in part by Martin Marietta Corporation and in part by the Defense Advanced Research Projects Agency (DOD), DARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and DARPA Order 4395, monitored by the Office of Naval Research, under Contract MDA 903-81-C-0411.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of their sponsors.

Table of Contents

Abstract

This paper reports on our initial results in using Ada as a Hardware Description Language. Ada provides abstraction mechanisms to support the development of large software systems. Separate compilation as well as nesting of packages, tasks, and subprograms allow the construction of modular systems communicating through well defined interfaces. The complexity of modern chips (e.g. those proposed in the VHSIC program) will require the use of those features that make Ada a good language for programming-in-the-large.

The key to our approach is establishing a writing style appropriate to the objective of describing both the behavior and the structure of hardware components. We model a hardware system as an ensemble of typed objects, where each object is an instance of an abstract data type. The type definition and the associated operations are encapsulated by a corresponding package. In this paper we illustrate our approach through a series of examples, building up a hypothetical hierarchy of hardware components. We conclude by discussing ways to describe arbitrarily complex simulation models and synthesis styles.

1 Introduction

The work that lead to this report started in response to a DoD request for proposals to design a Hardware Description Language for the Very High Speed Integrated Circuit (VHSIC) program. After analyzing the requirements we found that Ada[1] [ANSI, 1983] could be a powerful, cost-effective hardware description language since it provides abstraction mechanisms to support the development of large software systems. Separate compilation as well as nesting of packages, tasks, and subprograms allow the construction of a modular system communicating through well defined interfaces.

A desire for a hardware description language should not obscure the strong commonality of approaches and techniques between designers of complex hardware and software systems. While the full power of Ada may not seem appropriate for the design and specification of small components, the design of moder chips (e.g. those proposed in the VHSIC program) will require the use of advanced complexity management techniques. We argue that these techniques are directly supported by those features of Ada which make it a good language for programming-in-the-large.

We hasten to add that what we are proposing is Ada, not an Ada-like language. We are not

[1] Ada is a trademark of the US Goverment. Ada Joint Project office

contemplating the writing of a special compiler; any off-the-shelf Ada compiler will do. We are not even proposing adapting or modifying some existing Ada support system; any validated Ada implementation will do[2].

1.1 Description of the Approach

We model a hardware system as an ensemble of typed objects, where each object is an instance of an abstract data type. The type definition and the associated operations are encapsulated by a corresponding package. Thus, each package manages a particular kind of hardare component (e.g., wire, nand gate, multiplexor). The public operations of each package include object creation, object construction (from its component parts, interconnection of those parts, and association of these parts with the outer object's interface pins), and simulation. The semantics of these operations are explained in later sections.

Central to our representation of hardware objects are the dual concepts of behavioral and structural views of a hardware description. For example, a multiplexor may be described behaviorally as an object that selects among a list of inputs. Alternatively, it can be described as a collection of interconnected nand gates (in a multiplexor configuration, of course). A behavioral description is generally a more abstract view; it hides structural details which introduce implementation decisions. The structural description includes only the information about an object's components and their interconnections. The behavior implied by a structural description is determined by the behavior of its components and by the way they are interconnected. Behavior is not intrinsically "higher" than structure since, at the lowest level, some components are taken as primitives; their structure is hidden, and their description is purely behavioral. In fact, as we shall see in some of our examples, mixing structural and behavioral descriptions at the same "level" can be used to enhance the descriptive power of the notation, making the intentions of the designers more apparent.

1.2 Elements of Style

Discussions about programming styles often degenerate into theological arguments (witness the long standing arguments about indentation and capitalization of keywords and identifiers). We do not pretend to say that the style we have used in our examples is the best or that it should adopted by all users. As a matter of fact, the examples are contrived to display the features of the language, at the expense of having perhaps too many levels in the hierarchy of components. In a production environment we would expect say, latches and flip-flops, to be implemented as primitive components

---

[2] Actually, we do not need the full language; a reasonable subset, containing the right features is sufficient. Talk about Ada subsets is, however, considered heresy in some circles and we will not raise this point again.

whose descriptions are carefully handcrafted for simulation efficiency, rather than to be implemented by building them up from inverters and nand gates. In addition, to save space, in this paper we have limited ourselves to illustrate our approach via simple examples, although we have explored other, more complex descriptions in [Maloney et al., 1985].

In constructing the examples we have followed a few guidelines to emphasize readability and modularity and to define appropriate layers of abtractions.

Readability.- The complexity of many hardware systems and their equivalent software representations requires that the code be easily understood. We do rely on comments and the flexibility Ada provides for writing extended and legible identifiers. Appropriate selection of names for variables, types, and operations are also important.

Ada permits the overloading of enumeration literals and subprogram names. We take advantage of this to reduce the names of distinct identifiers that must be learned by the user. The basic operations needed to create, construct, and simulate hardware components have the same identifier, independent of the component type. The language provides mechanisms to resolve the ambiguities.

Modularity.- We define this as the ability to connect objects that are of different types through compatible interfaces. The strong typing in Ada prevents the kind of error in which a component of the wrong type is used by mistake (e.g. passing a nand gate to a D Latch simulation procedure). To permit the connection and transfer of signals between components, we use universal interface types (e.g. pins and buses). All components define their interface in terms of these types.

Layers of abstraction.- We define this as the ability to have multiple levels of representation for the structure and behavior of hardware objects. In our approach we use packages to define libraries of abstract types, one per package. Each package is built upon types and operations defined in other packages, in a hierachical fashion. In addition, the separation of specifications and bodies for these packages, permits the use of multiple versions of bodies supporting the same specification. This has important advantages in that we can quickly "plug-in" more efficient simulation models or synthesis algorithms or design rule checkers, etc. without ever having to alter a client package, or even recompile it. The switch happens at link time.

1.3 Overcoming Language Limitations

Ada was not designed with hardware descriptions in mind, thus we have adopted some conventions to overcome two shortcomings in the language.

Lack of Timing Primitives.- Ada has no primitives for expressing time and time-based relationships. Mapping a hardware description written in Ada to actual hardware requires either that the hardware be implemented using fully asynchronous circuitry, a practice not widely advocated, or that timing specifications be introduced in the process of mapping to hardware. The direct-mapping approach is the object of current research at the University of Utah [Organick et al., 1984.]

We overcome this deficiency by building into the packages operations that perform the synthesis of the abstract data type into hardware, incorporating the appropriate synchronization and timing information. Thus, rather than counting on a "smart" compiler to decipher the designer's intentions, we use "smart" programs and libraries, where these intentions are explicitly stated.

Ada does not treat packages as first class objects. Ada packages may not directly model hardware components unless such packages are elaborated at compile time; they cannot be created dynamically as values that can be assigned to variables or passed as parameters. Ada tasks do have some of these desirable features, however they suffer from other limitations (e.g. a task specification cannot define and "export" data types, constants, or objects, only entries.)

This is an unfortunate but not unsurmountable difficulty. In our approach, we use packages to manage instances of (first class) record types which in turn model hardware components.

2 Elements of the Description Language

2.1 Representing Connections

Before we present the details of how hardware objects are represented, it is necessary to address the problem of intermodule connections.

The relationship that exists between hardware objects[3] and interconnections is many-to-one; that is, many objects can be connected through one connection. A first representation of this relation could have each object reference the "wire" to which it is connected. For example, if components A, B and C are all connected, we have the arrangement depicted in Figure 1. This representation is

[3] Actually, here we are referring to an input or output of an object. For example, the output of a nand gate.

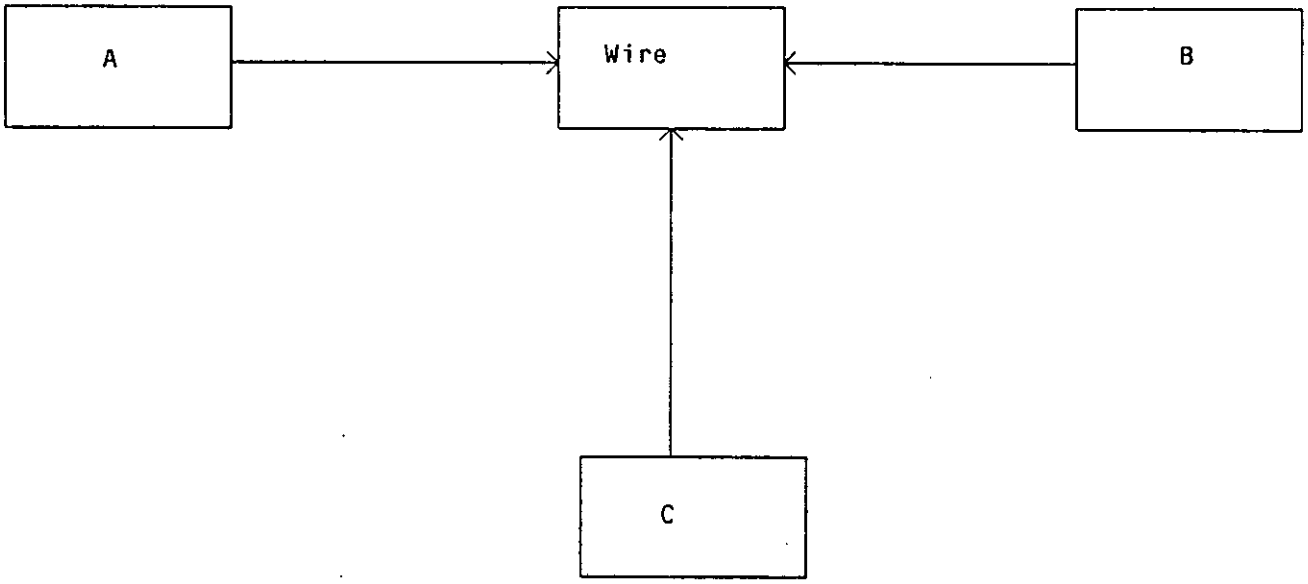


Figure 1: Three components connected by a simple wire

adequate for most situations. Here we treat the wire itself as an object. For simulation purposes, the wire object can have a value attribute that could be set and read by the components that it "joins". Information about how many components the wire is connecting can also be maintained in the wire object.

The deficiency in this representation becomes apparent when we allow components to be connected in an arbitrary order. In that situation, we must be able to connect objects that are already connected by wires. Figure 2 illustrates this point. In connecting components A and C, we would like the resulting configuration to have one wire that is referenced by components A, B, C and D. To accomplish this requires that we change C and D to reference wire_1 or A and B to reference wire_2. Note that either of these operations assumes the capability to find all references to a wire.

The desire for freedom in the order that connections are made motivates a slightly more complex representation of wire interconnections. The deficiency of the previous simple strategy is that there is no way to reference all of the objects connected by a wire. An intermediate "pin" data structure solves this problem in the following way. If a wire actually establishes connections between pins of

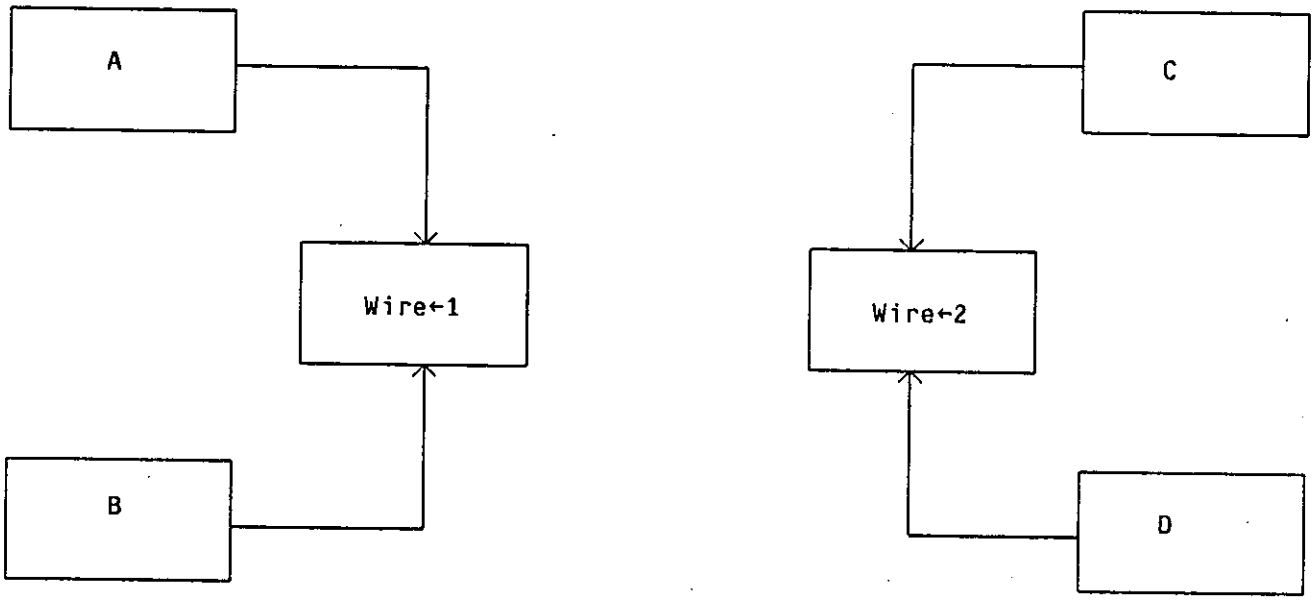


Figure 2: Two components already connected by simple wires

objects, it can reference, through a linked list of "pins", all of the "pins" that are connected to it. This is illustrated by the three pins connected as shown in Figure 3. The task of connecting pins that are already connected thus involves trivial linked-list operations to reconfigure the interconnection and merge the pin connections to one wire.

To support the abstractions or wires and pins, we have written a package, Pin_Mgr that declares wires and pins as data types and defines appropriate operations for manipulating objects of these types. The package is written so that most details about intermodule connections are placed in the body of the package and are therefore hidden from the users (the complete package listings appear in the Appendix). The public operations of this package are:

- The procedure CONNECT, which connects two pins (i.e. links them in a 'wire' list.)
- The procedure DISCONNECT, which breaks a connection (i.e. removes a pin from a 'wire' list.)
- The procedure EQUATE, which associates an internal pin of an object with an external pin (i.e. brings out an internal component-pin.)

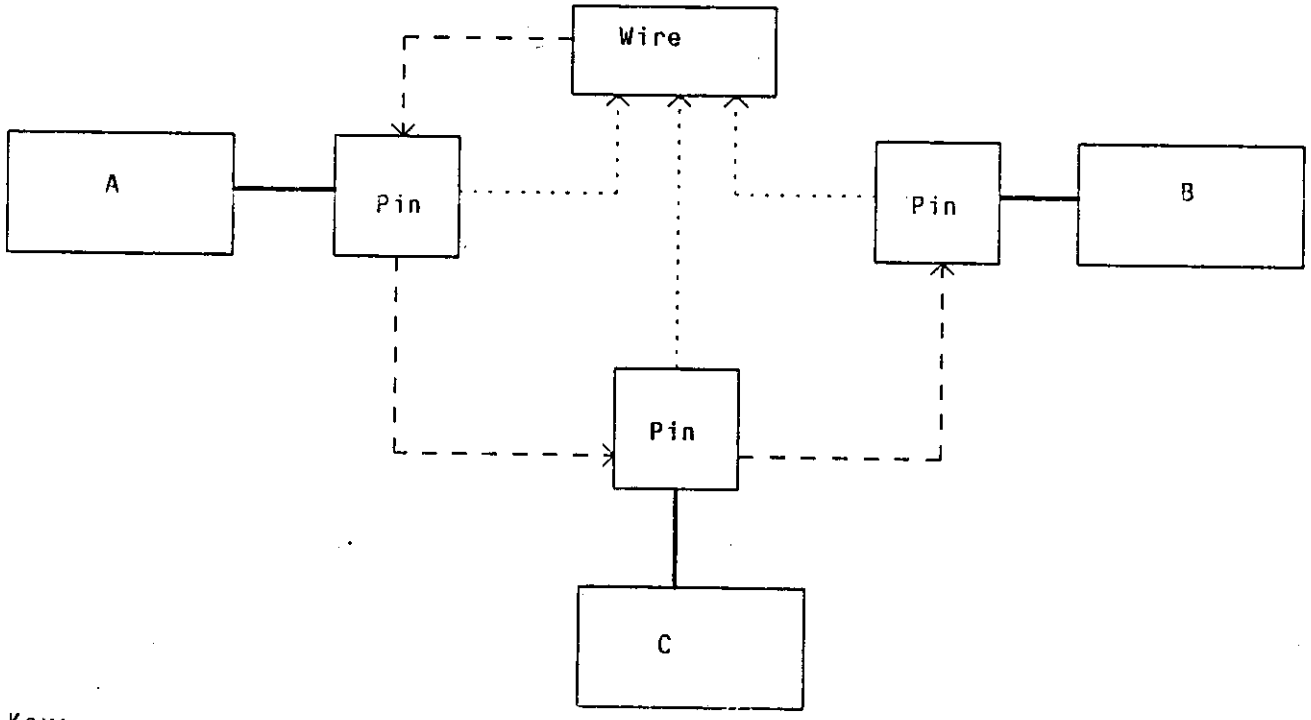


Figure 3: Three components connected by a simple wire using pins.

- The procedure UNEQUATE, which undoes an equate.
- The procedure SET_VALUE, which sets the value on a wire. This procedure and the following function can be used for simulation.
- The function VALUE_OF, returns the value of (i.e. level on) a wire.
- The function FAN_OUT, returns the number of pins connected to a wire.

The strategy, then, in building and connecting components is to provide each external input or output of an object (representing a component) with a "pin" that can be used in connecting the object with other objects. Pins and wires are not limited to modeling the idealized connections in our sample package; physical attributes such as capacitance, delays, loads, distances, locations, etc. can be easily described as "attributes" of (i.e. fields of the record types modeling) pins and wires.

2.2 Representing Buses

A bus can be described as an array of pins, using conventions similar to those of a pin. A package supporting this abstraction is listed in the Appendix. The visible operations of the Bus_Mgr package are:

- The procedure CONNECT, which connects two internal buses (provided the buses have the same width.)
- The procedure UNCONNECT, which breaks a connection.
- The procedure EQUATE, which associates an internal bus of an object with an external bus of the same width.
- The procedure UNEQUATE, which undoes an Equate.
- The procedure SET_VALUE, which sets the value on a set of wires, again as long as the width is the same.
- The function VALUE_OF, which returns the value of a set of wires.

The strategy, then, in building and connecting components is to provide each external input or output of an object with buses or pins that can be used in connecting the object with other objects.

2.3 Representing Hardware Objects

We can identify three possible approaches to the problem of representating hardware objects as typed data objects.

The first approach is to declare hardware objects as totally "private" (in the Ada sense). All operations on objects are defined by a set of procedures and functions that involve such objects, but nothing about the objects' structure is visible outside these procedures. Here problems arise when attempting to interconnect such objects, since we have no knowledge about an object's interface.

The other extreme is to declare hardware objects as totally "public" (again, in the Ada sense). However, this method exposes information about an object's structure that is irrelevant for connecting the object with another object.

The third approach is a combination of the previous two: we represent hardware structures using data types that contain both public and private parts. The public part of an object contains its interface information only, while the private part contains implementation details. The example in Figure 4, shows the (public) specifications for D flip flops.

```
type d_ff_components is private;

type d_flip_flop_record is
    record
    -- inputs
    dbar   : pin;    --Input data signal d (inverted).
    clkbar : pin;    --Input clock signal (inverted) to run slave latch.
    clear  : pin;    --Input signal to clear the flip-flop.
    clock  : pin;    --Clock signal to run master latch.
    -- output
    qbar   : pin;    --Output data signal q (inverted).
    -- private
    components : d_ff_components;
    end record;
```

Figure 4: Structure of the D Flip-Flop

In the example, the COMPONENTS field of the D Flip-Flop record is a component of a private type. Although it is visible as a record component outside its enclosing package, its structure is private. Alternatively, the publically visible fields of an object could be defined as the discriminants of an Ada private record type, with the hidden parts declared in the full type declaration, in the private part of the package specification. With this approach, not even the COMPONENTS field is visible outside the package. The drawback is that, in Ada, the values of discriminant fields cannot be modified except by a full record assignment.

2.4 Operations on Objects

Now we will describe how to use a hardware data object, that is to instantiate it, to establish its functionality, and to simulate it. To begin this process one must first create the object. The function CREATE allocates storage to hold values for a component's interface.

Once a data object has been instantiated we may perform other operations on it, such as CONSTRUCT and SIMULATE. The CONSTRUCT procedure establishes a structural description of the object by creating and connecting the object's subcomponents.

Once the function CREATE and optionally the procedure CONSTRUCT have been invoked, objects are ready to be simulated. For each object type that we define, we provide a SIMULATE procedure. This procedure operates upon information that is placed on the input pins to produce the results on the output pins. Objects at the lowest design level (primitive objects) are simulated by executing their behavioral description.

An object can either be simulated directly, by executing a SIMULATE procedure that implements its behavior or indirectly, by executing a SIMULATE procedure that invokes the procedures that simulate the subcomponents. Details of the algorithms used to simulate or construct objects are hidden. The subprogram specifications only describe the types of the parameters and the results.

Using this approach, mixed-level simulation is easily implemented (A similar approach to mixed level simulation using Simula 67 is described in [Lindstrom, 1983]). The order of simulation of components should begin with the input pins and follow the flow of new data throughout the network of components. Once all of the components have been simulated the appropriate output values will be placed on the output pins.

3 Shift Register Example

This section presents a complete example of the specification of a composite hardware object (i.e. an object that has subcomponents.)

Our specification of a shift register (National Semiconductor MM 74C168 [National, 1981]) is built bottom-up. We begin by creating certain low-level components, namely 2- and 3-input nand gates and inverters, described by packages named Two_Input_Nand_Gate_Mgr, Three_Input_Nand_Gate_Mgr, and Inverter_Mgr, respectively. These packages define primitive objects. Primitives have inputs and outputs and only a behavioral description; they are not represented by inter-connected subcomponents.

The package supporting the abstraction of an inverter declares two data types and two operations. The data types describe an inverter as a record with two fields, the input and output pins respectively. Since inverters (as well as other gates) are easier to handle as Ada access (i.e. pointer) types, the operations defined on inverters do not take an inverter record directly but rather they manipulate pointers to inverter records.

The CREATE operation is used to instantiate an inverter. Since this is a primitive component, there is no need to create and connect internal components, as we shall see in later examples. The SIMULATE operation computes the value at the output pin, depending on the value at the input pin. Notice that we are ignoring internal delays; these are idealized inverters.

The package supporting the abstraction of a nand gate is described as a generic package. This permits the definition of nand gates of arbitrary number of inputs by simply instantiating the generic package, with the right parameter (the number of input pins), without having to rewrite the type and

operation declarations. The input pins are modeled by a dynamic array of pins, whose dimension is specified when the generic package is instantiated.

Using these component definitions, we can establish a structural description for a D Latch. (See Figure 5.) The package D_Latch_Mgr package establishes the structural description of D latches by creating and interconnecting an inverter and four nand gates whenever the CONSTRUCT procedure (in the D Latch package) is invoked.

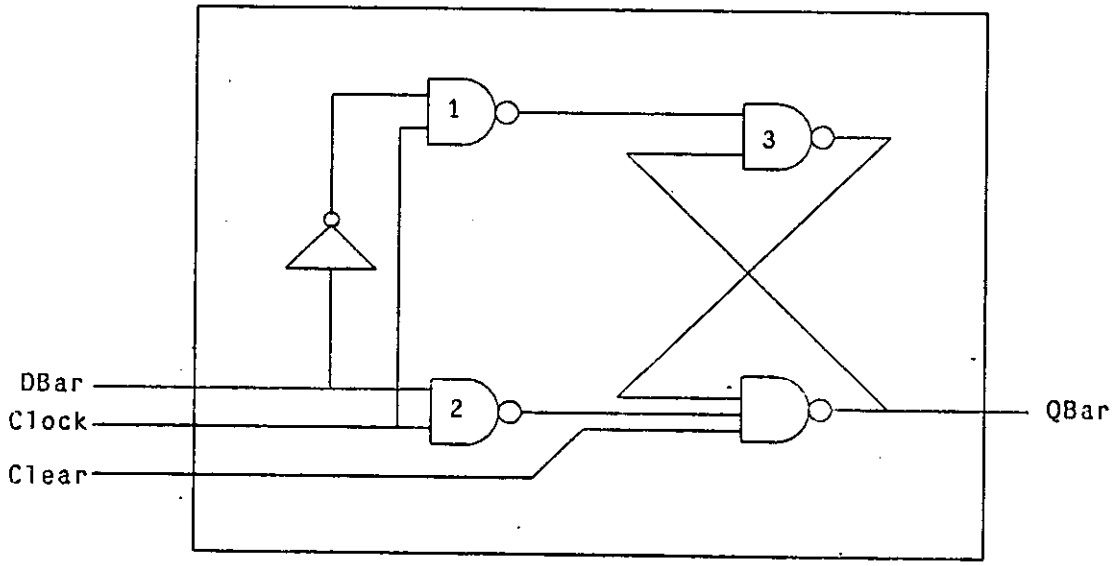


Figure 5: Inside view of the D Latch.

In the definition of the D Latch record type, we are hiding from the users of the abstraction the nature of the implementation of the latch. That is, only the input and output pins are directly available. The fact that there are components is revealed by the definition of the COMPONENTS field; however, since this field is declared to be of a private type (D_LATCH_COMPONENTS), no user of the package can make assumptions about its structure.

In addition to the CREATE and SIMULATE operations, the D Latch package also provides a CONSTRUCT operation. This operation must be invoked after a D Latch has been created and before it can be simulated. It builds the latch by instantiating the internal components, connecting them in the right configuration, and equating some internal component pins to the input and output pins of the latch.

The construction of components continues for the D flip-flop, the next level up in this component hierarchy. The package D_Flip_Flop_Mgr defines data objects that are composed of D latches (See Figure 6.)

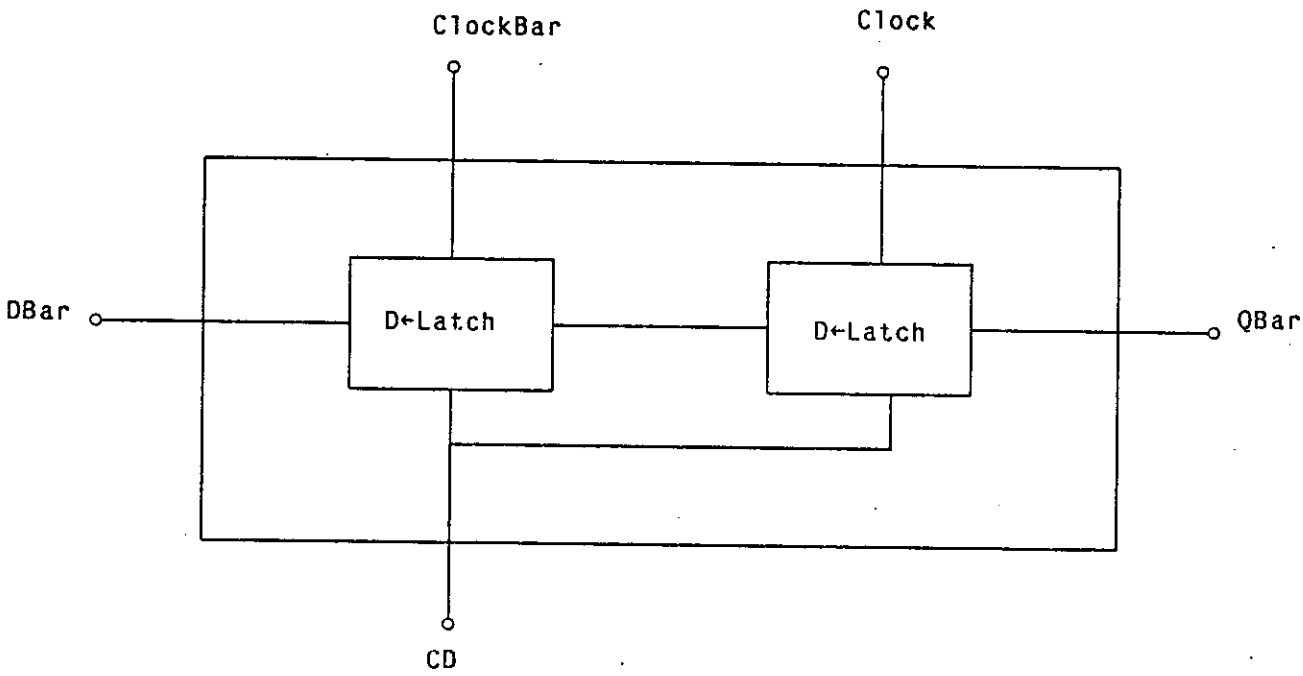


Figure 6: Inside view of the D Flip-Flop.

Notice the parsimonious nature of our approach. Every new component type is supported by a package which exports a record type and an associated access type; this permits the manipulation of the object by the support subprograms. In addition, the package exports procedures to create, construct, and simulate the component. By hiding the internal structure of a component and the implementation of the operations, the designer is free to correct or enhance the abstraction, without having to worry about amending packages that import the abstraction (provided of course, that the changes do not affect the visible part of the abstraction.) Any hardware system built out of components described in this fashion can in turn be used as a primitive component in later designs, provided these simple rules of style are observed.

As with the D latch and the D flip-flop, the serial shift register (Figure 7) is constructed by connecting components such as the inverter, nand2, and D flip-flop together in the correct (graph) structure. Once constructed, the shift register can be simulated by invoking the procedures that simulate its components.

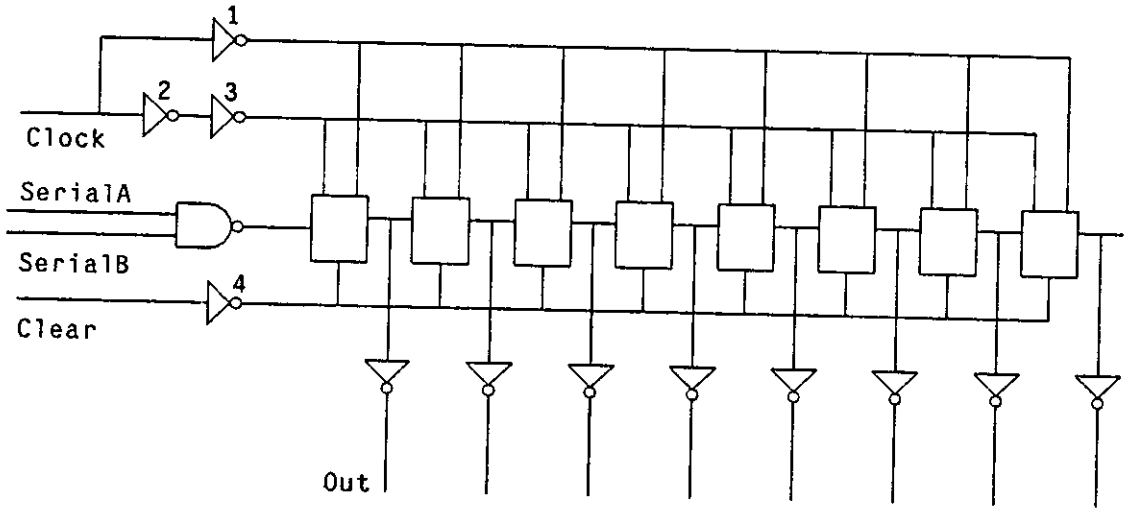


Figure 7: Inside View of the MM 74C168 Shift Register.

This hierarchical design is represented in the compilation dependency graph for the corresponding Ada packages, as shown in Figure 8.

4 Timing Models: An Example

In the preceding examples we have exhibited the power of the language to describe the structure and interconnections of hardware components. In this section we describe how it is possible to implement, within the same framework, arbitrarily complex timing and synchronization models, as well as synthesis algorithms.

By way of example, we have chosen to describe how the element of delay can be added to our library. We will represent time after the CONLAN model of computation [Piloty et al. 1983]. CONLAN uses the notion of a history of values to model digital hardware. Computation *step signals* correspond to transient values, due to the propagation of state changes in the system. The duration of a step is negligible and possible intermediate values in the carriers are invisible to the hardware designer -- only the final value of a step signal has significance. *Time signals* are sequences of step signals along time: one step signal per unit of time. Time signals can be inspected for past values (the last value of the step signal associated with a given point in the past) as well as current values (the last value of the current step signal).

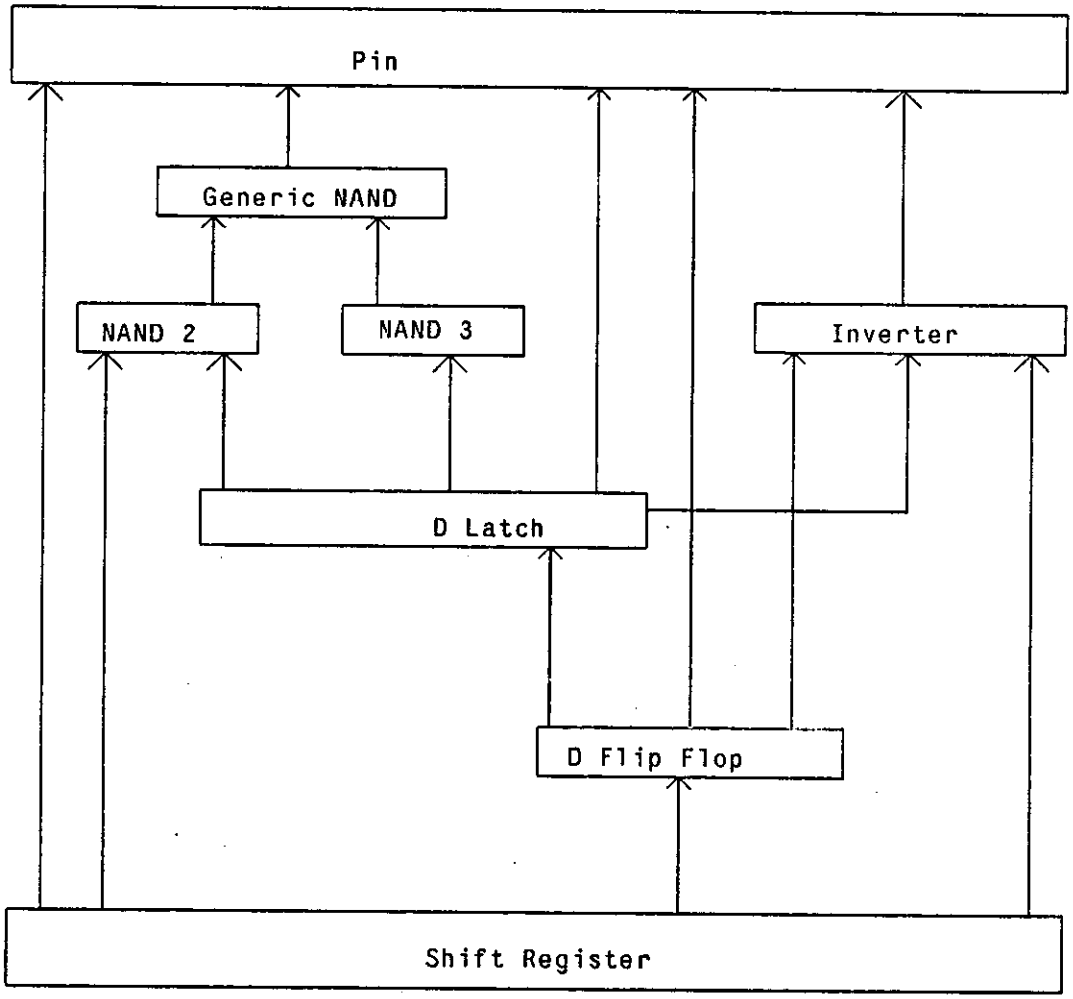


Figure 8: Compilation Dependency Graph for the Shift Register

Using this two-layered model of time, we modify our simple pin package as shown in the Appendix (under "Extended Pin Specification" and "Extended Pin Body".) Notice that in addition to the additional history carried by a wire, we have added an extra parameter (DELAY_VALUE) to the VALUE_OF function. The function now returns a previous value of a signal. We can now use this function to model a more realistic inverter gate, by rewriting the SIMULATE function, as shown in Figure 9.

Notice that the VALUE_OF function in the new pin package specifies a default for the DELAY_VALUE

```
procedure Simulate(v : in  inverter_gate)
--
-- Function:
--  This function simulates the logic of an inverter by inverting
--  the value of the input pin and placing the value on the output pin.
--  The function assumes a 10-unit gate delay.
is
begin
case Value_of(v.input,10) is
     when low =>                    -- If input is low the output is high.
               Set_value(v.output,high);
               return;
     when high =>                   -- If input is high the output is low.
               Set_value(v.output,low);
               return;
     when others =>                    -- If input is undefined, no change
               null;
     end case;
  end Simulate;
```

Figure 9: Inverter with Internal Delay

parameter. If no delay is provided by the caller, the last value assigned to the pin is returned (i.e. no delay is assumed.)

In this example, we have made a radical change in the package supporting the abstraction of a pin, yet the only externally visible change is the addition of one extra parameter to the VALUE_OF function. In addition, by providing a default value corresponding to the previous, no-delay version, all we have to do is recompile, without changes, any existing library packages. That is, older, idealized (i.e. no-delay) components still work; new, more realistic components can now be described, and both kinds of components can be mixed in a design.

To conclude this section, we point out that in our approach we are not limited to using pins and internal components as the fields of a record modeling some hardware component. We can just as easily declare fields whose values correspond to physical dimensions, power requirements, locations, etc. Since the process of constructing components is done by calls to operations defined in the library packages (CREATE and CONSTRUCT), it is rather easy to keep track of all instances of these components and to check that no design rules are being violated. The data structures needed for the bookkeeping provide an internal representation of the design; translating it into masks, wire-lists, or other manufacturing information gives us the path towards powerful and flexible design automation systems[4].

[4] As powerful and flexible as the code we are willing to write, and we have all the power of Ada to do this.... Don't be surprised, the emperor in the fairy tale was naked!

5 Conclusions

In a conventional CAD environment, the separation between the user and the toolmaker is very sharp. Tools (translators, simulators, synthesis programs etc.) are written in different languages, by separate groups of individuals, who may in turn be distinct from those in charge of maintaining the ensemble. Users are bound by the implementers' decisions (and mistakes) and are not usually in a position to do anything about them, short of waiting for upgrades or fixing the problems themselves. One of the most serious deficiencies with this conventional approach is that often an implementation will bind knowledge about a particular technology, synthesis style, or simulation models into the implementation in such a way that it is often not possible to change it to reflect new technologies or better design methods.

The technique we are describing is certainly no panacea; errors can still be made. However, we are eliminating the middle men and exposing to the users (i.e. designers) the full implementation, technology dependent decisions, timing models, and synthesis styles. Since the implementation language is the same language that is used in the day-to-day activities of the designers, they can understand the source of the problem, can propose solutions, and finally, can implement the solution themselves, without further ado. Good system management practices will probably impose some mechanisms to prevent chaos from arising; in particular, it is likely that only expert designers will be allowed to implement such changes. Ada provides powerful features to support the development, maintenance, and graceful evolution of large software systems; these same features will be invaluable in CAD systems of the 80's and beyond.

The advantages of using the same language for both the design of hardware and software are evident. The flexibility in delaying the binding of (hardware) implementation decisions discussed in this paper is easily extensible to a more basic decision, namely, whether a portion of a system is to be built in hardware or in software. The use of a single language together with a convention on style, permits a designer to write an abstract interface to a computing engine while retaining the freedom to implement this engine in either hardware or software. The flexibility continues throughout the life-cycle of the engine; the decision can be reversed at a later time, if the trade-offs change, without affecting in any way the users of the abstraction.

In addition to the obvious superiority of Ada as a programming language over existing special purpose hardware description languages, there are other reasons why Ada is an attractive hardware design tool.

Ada is a standard language that enjoys the support of the largest software user in the world, the U.S. Department of Defense. Thus, it is inevitable that rich programming environments will be built around Ada and that the community of users will be several orders of magnitude larger than that of any existing or proposed HDL. Not only are modern software development technologies easier to apply by the use of Ada but large user communities provide a continuous supply of tools, methods, training materials, etc. All of these contribute to a reduction of the life cycle costs of a project.

6 Acknowledgements

We would like to acknowledge the helpful input from David Barton, Peter Tinker, and Jay Woodward, graduate students in the Department of Computer Science, University of Utah.

7 References

[ANSI, 1983] Reference Manual for the Ada Programming Language, American National Standards Institute, Inc., New York, ANSI/MIL-STD-1815A-1983.

[Lindstrom, 1983] G. Lindstrom and P. Tinker, "VHDL Simulator Design", Technical report, Department of Computer Science, University of Utah, 1983.

[National, 1981] CMOS Databook, National Semiconductor Corp., 1981, p 1-68.

[Organick et al., 1984] E. Organick, T. Carter, M. Maloney, A. Davis, A. Hayes, D. Klass, G. Lindstrom, B. Nelson, and K. Smith, "Transforming an Ada Program Unit to Silicon and Verifying its Behavior in an Ada Environment: A First Experiment", IEEE *Software*, Vol. 1, No. 1, January 1984, pp. 31-49.

[Piloty et al., 1983] R.Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, and P. Skelly, CONLAN Report. Lecture Notes in Computer Sciences No. 151, Springer-Verlag, 1983.

[Maloney et al., 1985] M. Maloney, M. Barbacci, E. Organick, J. Woodward, and D. Barton, "Examples of Ada as a Hardware Description Language". Technical Report, Department of Computer Science, University of Utah, 1985. Also Technical Report, Department of Computer Science, Carnegie-Mellon University, 1985.

I. Package Listings

Pin Specification

```
package Pin_Mgr is
  --
  -- Function:
  --    This package defines the value that a pin can have and the
  --    procedures to connect, disconnect, equate, unequate, set values,
  --    and find values on pins.

  type wire_level_value is (low, high, undefined);

  type pin is private;

  null_pin : constant pin; -- deferred constant

  procedure Connect(pin_1, pin_2: in out pin);
  --
  -- Function:
  --    This procedure will connect two pins.

  procedure Disconnect(pin_1, pin_2: in out pin);
  --
  -- Function:
  --    This procedure will disconnect a pin from whatever it is
  --    connected to.

  procedure Equate(pin_1, pin_2: in out pin);
  --
  -- Function:
  --    This procedure will connect an external pin to the internal
  --    pin of a component.

  procedure Unequate(pin_1, pin_2: in out pin);
  --
  -- Function:
  --    This procedure will disconnect an external pin from whatever it is
  --    internal pin of a component.

  procedure Set_value(p : pin; v : wire_level_value);
  --
  -- Function:
  --    This procedure will give the specified pin the given value.

  function  Value_of(p : pin) return wire_level_value;
  --
  -- Function:
  --    This function returns the pin_level_value of a pin.


  function  Fan_out(p : pin) return natural;
  --
  -- Function:
  --    This function returns the total number of
  --    pins that are currently connected by the wire that
  --    this pin is connected to.

private
  -- 1. pins are connected by adding them to the list of pins of
  --       a created wire.
  -- 2. the global attributes of the connection are stored
  --       in the wire record.

  -- When pins are connected, a pin record and a wire record are created.
  -- The wire record points to the first pin record in the list and that
  -- pin record points to the next until the last pin contains a null
  -- indicating the end of the list.  Each pin record contains a pointer
  -- to the wire record so the head of the list can be found from any pin.

  type pin_record;

  type pin is access pin_record;

  type wire_record;

  type wire is access wire_record;

  null_pin : constant pin := null;

  null_wire : constant wire := null;

  type pin_record is
    record
      connecting_wire  : wire  := null_wire;
      next_pin         : pin   := null_pin;     -- next pin in list
    end record;

  type wire_record is
    record
      value            : wire_level_value := undefined;
      list_of_pins     : pin              := null_pin;
      number_of_pins   : natural          := 0;
    end record;
end Pin_Mgr;
```

Pin Body

```
package body Pin_Mgr  is
  --
  -- Function:
  --   Pins allow value access to all components, these values are stored in
  --   wire records. This package determines how to connect pins with the
  --   procedures: connect,disconnect(for internal components) and equate,
  --   unequate(for internal to external connection).The function Connected
  --   returns a boolean value stating if the pin is connected or not. The
  --   function Value_of returns the value of the pin.Function Fan returns
  --   the number of pins connected to a specific wire. Procedure Set_value
  --   sets the value of a wire record.

  function Connected(p : pin) return boolean
    --
    -- Function:
    --   Returns a boolean value true if pin is connected, and false if the
    --   pin is not connected.
  is
  begin
    if p = null_pin or else p.connecting_wire = null_wire then
      return false;
    else
      return true;
    end if;
  end connected;

  procedure Connect(pin_1, pin_2 : in out pin)
  is
    --
    -- Function:
    -- pins are connected by linking them into a list
    -- and creating a wire common to them both.
    first_pin, last_pin, loop_pin : pin;
  begin
    -- first make sure that pin_records exist for each
    -- pin
    if pin_1 = null_pin then -- create a pin_record for pin_1
      pin_1 := new pin_record;
    end if;
    if pin_2 = null_pin then -- create a pin_record for pin_2
      pin_2 := new pin_record;
    end if;

    if not Connected(pin_1) and not Connected(pin_2) then
      -- Case 1 : neither pin is connected
      --   If neither pin is connected a new wire record is created and
      --   both pins point to it. The pin_1 becomes the head pin and is
      --   pointed to by the wire record. The pin_2 becomes the next wire
      --   for the pin_1. Finally the number of pins pointing to the wire
      --   record is incremented by 2.
      pin_1.connecting_wire := new wire_record;
      pin_2.connecting_wire := pin_1.connecting_wire;
      pin_1.next_pin        := pin_2;
      pin_1.connecting_wire.list_of_pins := pin_1;
      pin_1.connecting_wire.number_of_pins :=
        pin_1.connecting_wire.number_of_pins + 2;

    elsif Connected(pin_1) and not Connected(pin_2) then
      -- Case 2 : pin_1 is connected and pin_2 isn't
      -- If pin_1 is connected to a wire and pin_2 is not. Pin_2 wire record
      -- becomes pin_1 wire record.pin_2 is now the head of the pin list
      -- which is pointed to by wire record. Pin_1 becomes the next pin for
      -- pin_2 and the number of pins for the wire is incremented by 1.
      pin_2.connecting_wire := pin_1.connecting_wire;
      pin_2.next_pin := pin_1.connecting_wire.list_of_pins;
      pin_1.connecting_wire.list_of_pins := pin_2;
      pin_1.connecting_wire.number_of_pins :=
        pin_1.connecting_wire.number_of_pins + 1;

    elsif Connected(pin_2) and not Connected(pin_1) then
      -- Case 3 : pin_2 is connected and pin_1 isn't
      -- If pin_2 is connected to a wire and pin_1 is not. Pin_1 wire record
      -- becomes pin_2 wire record,pin_1 is now the head of the pin list
      -- which is pointed to by wire record. Pin_2 becomes the next pin for
      -- pin_1 and the number of pins for the wire is incremented by 1.
      pin_1.connecting_wire := pin_2.connecting_wire;
      pin_1.next_pin := pin_2.connecting_wire.list_of_pins;
      pin_2.connecting_wire.list_of_pins := pin_1;
      pin_1.connecting_wire.number_of_pins :=
        pin_1.connecting_wire.number_of_pins + 1;

    else
      -- Case 4 : both are connected
      -- The two wire records will be merged and the head will be the
      -- head of the pin_2 record list.
      if pin_1.connecting_wire = pin_2.connecting wire then
        return;                    -- already connected to each other
      end if;

      pin_1.connecting_wire.number_of_pins :=
              pin_1.connecting_wire.number_of_pins +
              pin_2.connecting_wire.number_of_pins;
      loop_pin := pin_2.connecting_wire.list_of_pins;          -- can't be null
      first_pin := loop_pin;
      while loop_pin /= null_pin
      loop
        loop_pin.connecting_wire := pin_1.connecting_wire;
        last_pin := loop_pin;
        loop_pin := loop_pin.next_pin;
      end loop;
      -- Insert the (pin_1) list at the tail of the list_of_pins of the wire
      last_pin.next_pin := pin_1.connecting_wire.list_of_pins;
      pin_1.connecting_wire.list_of_pins := first_pin;
    end if;
  end Connect;

  procedure Disconnect(pin_1, pin_2 : in out pin)
  is
    --
    -- Function:
    -- The functionality is not yet defined.
  begin
    null;
```

Pin Body

```
  procedure Equate(pin_1, pin_2 : in out pin)
  is
    --
    -- Function:
    -- The connections are from internal pins to external pins.
  begin
    connect(pin_1,pin_2);
  end;

  procedure Unequate(pin_1, pin_2 : in out pin)
  is
    --
    -- Function:
    --  The connections from internal pins to external pins is disconnected.
  begin
    Disconnect(pin_1,pin_2);
  end;

  procedure Set_value(p : pin; v : wire_level_value)
  is
    --
    -- Function:
    --  Assigns a value to a wire record.
  begin
    if Connected(p) then
      p.connecting_wire.value := v;
    else
      null; -- raise some exception
    end if;
  end;

  function Value_of(p : pin) return wire_level_value
  is
    --
    -- Function:
    --  Determines the value of a pin and returns that value.
  begin
    return p.connecting_wire.value;
  end;

  function Fan_out(p : pin) return natural
  is
    --
    -- Function:
    --  Determines the number of pins pointing to a wire record and returns
    --  that value.
  begin
    return p.connecting_wire.number_of_pins;
  end;
end Pin_Mgr;
```

Bus Specification

```
with Pin_Mgr;
use  Pin_Mgr;

package Bus_Mgr is
  --
  -- Function:
  --   Buses are a defined as arrays of pins. and bus values as arrays
  --   of levels. This package provides procedures for connecting and
  --   disconnecting buses. The function Value_of returns the bus value of
  --   a bus. The procedure Set_value sets the value of a bus.

  type bus_value is array (natural range <>) of wire_level_value;

  type bus is array (natural range <>) of pin;

  procedure Connect(bus_1, bus_2 : in out bus);
  --
  -- Function:
  --    Creates a bus connecting a pair of (conforming) pin arrays
  --    belonging to subcomponents of a given component.

  procedure Disconnect(bus_1, bus_2 : in out bus);
  --
  -- Function:
  --    Undoes the Connect operation.

  procedure Equate(bus_1, bus_2 : in out bus);
  --
  -- Function:
  --    Creates a bus connecting a pair of (conforming) pin arrays,
  --    one array belongs to a component and the other to a subcomponent.

  procedure Unequate(bus_1, bus_2 : in out bus);
  --
  -- Function:
  --    Undoes an Equate.

  procedure Set_value(b : bus; v : bus_value);
  --
  -- Function:
  --    Assigns a bus value to a specified bus.

  function  Value_of(b : bus) return bus_value;
  --
  -- Function:
  --    Returns the bus value of a specified bus.

  bus_mismatch : exception;
  -- Raised when operations are attempted on nonconforming buses.
end Bus_Mgr;
```

Bus Body

```
with Pin_Mgr;
use  Pin_Mgr;

package body Bus_Mgr is
   --
   -- Function:
   --    This package provides implementations of procedures for connecting
   --    and disconnecting buses. The function Value_of returns the bus value
   --    of a bus. The procedure Set_value sets the value of a bus.

  function Nonconforming_buses(bus_1, bus_2: bus) return Boolean
  is
  begin
  if bus_1'length /= bus_2'length then
       return true;
    else
       return false;
    end if;
  end Nonconforming_buses;

  function Nonconforming_bus_and_value(b: bus; v: bus_value) return Boolean
  is
  begin
  if b'length /= v'length then
       return true;
    else
       return false;
    end if;
  end Nonconforming_bus_and_value;

  procedure Connect(bus_1, bus_2 : in out bus)
  is
    --
    -- Function:
    --      Creates a bus connecting a pair of (conforming) pin arrays
    --      belonging to subcomponents of a given component.
  begin
    if Nonconforming_buses(bus_1, bus_2) then
      raise bus_mismatch;
    end if;

    -- Connect the buses by connecting their respective pins.
    for i in bus_1'range
    loop
      Connect(bus_1(i), bus_2(i));
    end loop;
  end Connect;

  procedure Disconnect(bus_1, bus_2 : in out bus)
  is
    --
    -- Function:
    -- The functionality is not yet defined.
  begin
    null;
  end;

  procedure Equate(bus_1,bus_2 : in out bus)
    --
    -- Function:
    --      Creates a bus connecting a pair of (conforming) pin arrays,
    --      one array belongs to a component and the other to a subcomponent.
  is
  begin
    Connect(bus_1, bus_2);
  end;

  procedure Unequate(bus_1, bus_2 : in out bus)
    --
    -- Function:
    --   Undoes an Equate.
  is
  begin
    Disconnect(bus_1, bus_2);
  end Unequate;

  procedure Set_value(b : bus; v : bus_value)
    -- Function:
    --      Assigns a bus value to a specified bus.
  is
  begin
    if Nonconforming_bus_and_value(b, v) then
      raise bus_mismatch;
    end if;
    for i in b'range
      loop
        Set_value (b(i), v(i));
      end loop;
  end Set_value;

  function Value_of(b : bus) return bus_value
    --
    -- Function:
    --    Returns the value of a bus b
  is
    temp_val :bus_value (b'range);
  begin
    for i in b'range
    loop
      temp_val(i) := Value_of(b(i));
    end loop;
    return temp_val;
  end;
end Bus_Mgr;
```

Inverter Specification

```
with Pin_Mgr;    use  Pin_Mgr;

package Inverter_Mgr is
   --
   -- Function:
   --    This package specifies the external pin connections of an inverter
   --    and a function to create a specified inverter. The procedures to
   --    construct and simulate inverter execution are also described.

   type inverter_record is
      record
         input  : pin;
         output : pin;
      end record;

   type inverter_gate is access inverter_record;

   function Create return inverter_gate;
      --
      -- Function:
      --    Space is allocated for the inverter and all pins are set to a
      --    disconnected state.

   procedure Simulate(v: in inverter_gate);
      --
      -- Function:
      --    The procedure Simulate will read input values and generate the
      --    proper output signal for the inverter_gate.
end Inverter_Mgr;
```

Inverter Body

```
with Pin_Mgr;    use  Pin_Mgr;

package body Inverter_mgr is
   --
   -- Function:
   --    This package body contains procedure and function details to
   --    implement a nand gate, namely Create and Simulate.

   function Create return Inverter_Gate
      --
      -- Function:
      --    Creates a new instance of a inverter.
   is
   begin
      return new Inverter_record;
   end Create;

   procedure Simulate(v: in inverter_gate)
      --
      -- Function:
      --    This function simulates the logic of an inverter by inverting
      --    the value of the input pin and placing the value on the output pin.
   is
   begin
      case Value_of(v.input) is
         when low =>                    -- If input is low the output is high.
                  Set_value(v.output,high);
                  return;
         when high =>                   -- If input is high the output is low.
                  Set_value(v.output,low);
                  return;
         when others =>                 -- If input is undefined, no change
                  null;
         end case;
   end Simulate;
end Inverter_Mgr;
```

Generic Nand Specification

```
with Pin_Mgr;    use  Pin_Mgr;

generic

  N : in integer;                          --Number of inputs for nand gate.

package N_Input_Nand_Mgr is
  --
  -- Function:
  --    Package creates a nand gate with an input array of length N.

  type nand_input_array is array(1..N) of pin;

  type nand_record is
    record
      input  : nand_input_array;       --Array of input pins for nand gate.
      output : pin;
    end record;

  type nand_gate is access nand_record;

  function Create return nand_gate;
    --
    --Function:
    --   Creates a new instance of the nand gate.

  procedure Simulate(gate: in nand_gate);
    --
    -- Function:
    --    Simulates the nand_gate by reading input values and determining
    --    appropriate output.
end N_Input_Nand_Mgr;
```

Generic Nand Body

```
with Pin_Mgr;    use  Pin_Mgr;

package body N_Input_Nand_Mgr is
  --
  -- Function:
  --   This package body contains procedure and function details to
  --   implement a nand gate, namely Create and Simulate.

  function Create return nand_gate
    --
    -- Function:
    -- Creates a new instance of a nand_gate. Space is allocated for a
    -- nand_record and all pins are initialized to NULL.
  is
  begin
    return new nand_record;
  end Create;

  procedure Simulate(gate : in nand_gate)
    --
    -- Function:
    -- This function simulates the logic of a nand_gate from the input
    -- pins and sets the value of the output pin.
  is
  begin
    for p in 1..N
    loop
      if Value_of(gate.input(p)) = low then --If all inputs are low
        Set_value(gate.output,high);        --output pin is high.
        return;
      end if;
    end loop;
    Set_Value(gate.output,low);                  --Otherwise output pin is
    return;                                          --low.
  end Simulate;
end N_Input_Nand_Mgr;
```

Two-Input Nand Gate

```
with N_Input_Nand_Mgr;

package Two_Input_Nand_Mgr is
    new N_Input_Nand_Mgr(n => 2);

  --
  -- Instantiation of generic N_Input_Nand_Mgr package creating a 2
  -- input nand gate.
  --
```

Three-Input Nand Gate

```
with N_Input_Nand_Mgr;

package Three_Input_Nand_Mgr is
    new N_Input_Nand_Mgr(n => 3);

  --
  -- Instantiation of generic N_Input_Nand_Mgr package creating a 3
  -- input nand gate.
  --
```

D Latch Specification

```
with Pin_Mgr, Inverter_Mgr, Two_Input_Nand_Mgr, Three_Input_Nand_Mgr;

use  Pin_Mgr, Inverter_Mgr, Two_Input_Nand_Mgr, Three_Input_Nand_Mgr;

package D_Latch_Mgr is
  --
  -- Function:
  --   This package specifies the external pin connections of a d_latch and
  --   a function to create an instance of a d_latch. The procedures to
  --   construct and simulate its execution are also described. Note that the
  --   d_latch specified in this package is somewhat specialized in pin-out for
  --   use in constructing National Semiconductor's 74164 components.

  type d_latch_components is private;

  type d_latch_record is
    record
      --input pins
      clock :      pin;
      dbar  :      pin;
      clear :      pin;
      --output pins
      qbar  :      pin;
      --private part
      components : d_latch_components;
    end record;

  type d_latch is access d_latch_record;

  function Create return d_latch;
    --
    -- Function:
    --    Creates a d_latch instance.

  procedure Construct(dlt : in d_latch);
    --
    -- Function:
    --    The specified d_latch is constructed by first creating then
    --    connecting all internal components.

  procedure Simulate(dlt : in d_latch);
    --
    -- Function:
    --    The specified d_latch components are simulated in the exact order
    --    as a real chip would operate.

private
  type nand2_array     is array(1..3) of two_input_nand_mgr.nand_gate;

  type d_latch_components is
    record
      nand2_gates : nand2_array;
      nand3       : three_input_nand_mgr.nand_gate;
      inverter    : inverter_gate;
    end record;
end D_Latch_Mgr;
```

D Latch Body

```
with Pin_Mgr, Inverter_Mgr, Two_Input_Nand_Mgr, Three_Input_Nand_Mgr;
use  Pin_Mgr, Inverter_Mgr, Two_Input_Nand_Mgr, Three_Input_Nand_Mgr;

package body D_Latch_Mgr is
  --
  -- Function:
  --    This package describes the functions and procedures to produce
  --    a specified d_latch, to construct it from its components, and
  --    to simulate an actual electrically-static case of reading input
  --    values and returning output values.

  function Create return d_latch
    --
    -- Function:
    --    This function creates an instancse of a d_latch.
  is
  begin
    return new d_latch_record;
  end Create;

  procedure Construct(dlt:in d_latch)
    --
    -- Function:
    --    This procedure builds the components of a d_latch and
    --    interconnects them.
  is
  begin
    for i in 1..3
    loop
      dlt.components.nand2_gates(i) := Create;
    end loop;

    dlt.components.inverter          := Create;
    dlt.components.nand3             := Create;    -- This creates a
                                                   -- three input nand_gate.
    -- Equates Section
    ------------------
    Equate(dlt.dbar ,     dlt.components.inverter.input);
    Equate(dlt.dbar ,     dlt.components.nand2_gates(2).input(2));
    Equate(dlt.clock ,    dlt.components.nand2_gates(1).input(2));
    Equate(dlt.clock ,    dlt.components.nand2_gates(2).input(1));
    Equate(dlt.clear ,    dlt.components.nand3.input(3));
    Equate(dlt.components.nand3.output ,   dlt.qbar);
    Equate(dlt.components.nand3.input(2) , dlt.qbar);

    -- Connects Section
    -------------------
    Connect(dlt.components.inverter.output ,
            dlt.components.nand2_gates(1).input(1));
    Connect(dlt.components.nand2_gates(1).output ,
            dlt.components.nand2_gates(3).input(1));
    Connect(dlt.components.nand2_gates(2).output ,
            dlt.components.nand3.input(2));
    Connect(dlt.components.nand2_gates(3).output ,
            dlt.components.nand3.input(1));
  end Construct;

  procedure Simulate(dlt:in d_latch)
    --
    -- Function:
    --    Each component of the d_latch is simulated by reading input values
    --    and generating its output in the order that the circuit would
    --    normally be executed.
  is
  begin
    Simulate(dlt.components.inverter);
    Simulate(dlt.components.nand2_gates(1));
    Simulate(dlt.components.nand2_gates(2));

    --Feedback loop exists between nand gates two and three.
    Simulate(dlt.components.nand2_gates(3));
    Simulate(dlt.components.nand3);

    --When a feedback loop exists then the first gate involved
    --is executed again.
    Simulate(dlt.components.nand2_gates(3));
  end Simulate;
end D_Latch_Mgr;
```

D Flip-Flop Specification

```
with  Pin_Mgr, D_Latch_Mgr;
use   Pin_Mgr, D_Latch_Mgr;

package D_Flip_Flop_Mgr is
  --
  -- Function:
  --   This package specifies the input and output pins to a D_Flip_Flop
  --   provides operations to create and simulate instances of this component.

      type d_ff_components is private;

      type d_flip_flop_record is
        record
          -- inputs
          dbar    : pin;      -- Input data signal d (inverted).
          clkbar  : pin;      -- Input clock signal (inverted).
          clear   : pin;      -- Input signal to clear the flip_flop.
          clock   : pin;      -- Clock signal to run master latch.
          -- output
          qbar    : pin;      --Output data signal q (inverted).
          -- private
          componen : d_ff_components;
        end record;

      type d_flip_flop is access d_flip_flop_record;

  function Create return d_flip_flop;
    --
    -- Function:
    --   Creates a d_flip_flop.

  procedure Construct(dff: d_flip_flop);
    --
    -- Function:
    --   Creates and interconnects components of a d_flip_flop.

  procedure Simulate(dff: d_flip_flop);
    --
    -- Function:
    --   The specified d_flip_flop components are simulated in order,
    --   according to the pattern of signal flow from inputs to outputs.

private
  type d_latch_array is array (1..2) of d_latch;

  type d_ff_components is
    record
      d_latches : d_latch_array;
    end record;
end D_Flip_Flop_Mgr;
```

D Flip-Flop Body

```
with  Pin_Mgr, D_Latch_Mgr;
use   Pin_Mgr, D_Latch_Mgr;

package body D_Flip_Flop_Mgr is
  --
  --Function:
  --   This package contains the procedure and function details required
  --   to execute the specifications given in the package D_Flip_Flop_Mgr.

  function Create return d_flip_flop
    --
    --Function:
    -- Creates a d_flip_flop.
  is
  begin
    return new d_flip_flop_record;
  end Create;

  procedure Construct(dff:in d_flip_flop)
    --
    -- Function:
    -- Creates and interconnects components of a d_flip_flop.
  is
  begin
    -- Creates two instances of a d_latch.
      dff.components.d_latches(1)   := Create;
      dff.components.d_latches(2)   := Create;

    -- Equates Section
    ------------------
    Equate(dff.dbar , dff.components.d_latches(1).dbar);
    Equate(dff.clock, dff.components.d_latches(1).clock);
    Equate(dff.clkbar, dff.components.d_latches(2).clock);
    Equate(dff.components.d_latches(2).qbar , dff.qbar);

    -- Connects Section
    -------------------
    Connect(dff.components.d_latches(1).qbar,
            dff.components.d_latches(2).dbar);
  end Construct;

  procedure Simulate (dff:in d_flip_flop)
    --
    --Function:
    --   This procedure simulates the operation of the d_flip_flop
    --   by simulating its components, i.e., by executing the
    --   functional behavior of its components.
  is
  begin
    Simulate(dff.components.d_latches(1));
    Simulate(dff.components.d_latches(2));
  end Simulate;
end D_Flip_Flop_Mgr;
```

Shift Register Specification

```
with  Pin_Mgr, Inverter_Mgr, Two_Input_Nand_Mgr, D_Flip_Flop_Mgr;
use   Pin_Mgr, Inverter_Mgr, Two_Input_Nand_Mgr, D_Flip_Flop_Mgr;

package Eight_Bit_Parallel_Shift_Register_Mgr is
  --
  -- Function:
  --   This package specifies an 8-bit serial-in parallel-out shift register.
  --   The Create function, and the procedures Construct, and Simulate are
  --   provided for this register.

  type output_index is (a,b,c,d,e,f,g,h);

  type pin_array is array(output_index) of pin;

  type shifter_components is private;

  type shifter_record is
    record
      --input pins
      clock      : pin;
      serial_a   : pin;
      serial_b   : pin;
      clear      : pin;
      --output pins
      output     :  pin_array;
      --private part
      components :  shifter_components;
    end record;

  type shifter is access shifter_record;

  function Create return shifter;
    --
    -- Function:
    --   Creates an instance of a shifter (shift register).

  procedure Construct(sr: in out shifter) ;
    --
    -- Function:
    --   Constructs the shifter's internal components and
    --   interconnects them.

  procedure Simulate(sr: in shifter);
    -- Function:
    --   The specified shifter's components are simulated in order,
    --   according to the pattern of signal flow from inputs to outputs
    --   of the shifter.

private
  type misc_inverters_array is array(1..4) of inverter_gate;

  type output_inverter_array is array(output_index) of inverter_gate;

  type d_flip_flop_array     is array(output_index) of d_flip_flop;

  type shifter_components is
    record
      nand             : nand_gate;
      misc_inverters   : misc_inverters_array;
      output_inverters : output_inverter_array;
      d_flip_flops     : d_flip_flop_array;
    end record;
end Eight_Bit_Parallel_Shift_Register_Mgr;
```

Shift Register Body

```
with  Pin_Mgr, Inverter_Mgr, Two_Input_Nand_Mgr, D_Flip_Flop_Mgr;
use   Pin_Mgr, Inverter_Mgr, Two_Input_Nand_Mgr, D_Flip_Flop_Mgr;

package body Eight_Bit_Parallel_Shift_Register_Mgr is
  --
  -- Function:
  --   This package body specifies an 8-bit serial-in parallel-out shift
  --   register whose internal structure conforms with the National
  --   Semiconductor 74164.
  --   The Create function, and the procedures Construct, and Simulate
  --   are provided for this register.

  function Create return shifter
    --
    -- Function:
    --    Creates an instance of a shifter.
  is
  begin
    return new shifter_record;
  end Create;

  procedure Construct(sr: in out shifter)
    -- Function:
    --   Constructs the shifter's internal components and
    --   interconnects them.
  is
  begin
    -- Create all the components.
    for m in output_index
    loop
      -- Create the flip flops
      sr.components.d_flip_flops(m) := Create;
      -- Create the output inverters.
      sr.components.output_inverters(m) := Create;
    end loop;
    -- Create (4) misc_inverters.
    for i in 1..4
    loop
      sr.components.misc_inverters(i) := Create;
    end loop;
    -- Create a nand_gate.
    sr.components.nand := Create;

  -- Equates Section
  ------------------
    for m in output_index
    loop
      Equate(sr.components.output_inverters(m).output , sr.output(m));
    end loop;
    Equate(sr.clear, sr.components.misc_inverters(4).input);
    Equate(sr.serial_a, sr.components.nand.input(1));
    Equate(sr.serial_b , sr.components.nand.input(2));
    Equate (sr.clock, sr.components.misc_inverters(1).input);

  -- Connects Section
  -------------------
    for m in output_index
    loop
      Connect(sr.components.misc_inverters(1).output ,
              sr.components.d_flip_flops(m).clkbar);
      Connect(sr.components.misc_inverters(3).output ,
              sr.components.d_flip_flops(m).clock);
      Connect(sr.components.misc_inverters(4).output ,
              sr.components.d_flip_flops(m).clear);
      Connect(sr.components.d_flip_flops(m).qbar ,
              sr.components.output_inverters(m).input);
    end loop;
    -- also
    Connect(sr.components.misc_inverters(2).output ,
            sr.components.misc_inverters(3).input);
    Connect(sr.components.nand.output,
            sr.components.d_flip_flops(a).dbar);
    -- and also chain the flip flops
    for i in output_index'(a).. output_index'(g)
    loop
      Connect(sr.components.d_flip_flops(i).qbar ,
              sr.components.d_flip_flops(output_index'succ(i)).dbar);
    end loop;
  end Construct;

  procedure Simulate(sr:in shifter)
    -- Function:
    --   The specified shifter's components are simulated in order,
    --   according to the pattern of signal flow from inputs to outputs
    --   of the shifter.
  is
  begin
    Simulate(sr.components.misc_inverters(2));
    Simulate(sr.components.misc_inverters(3));
    Simulate(sr.components.misc_inverters(1));
    Simulate(sr.components.misc_inverters(4));
    Simulate(sr.components.nand);

    -- These components can be simulated in a loop since all their inputs
    -- will have been generated dynamically.
    for m in output_index
    loop
      Simulate(sr.components.d_flip_flops(m));
      Simulate(sr.components.output_inverters(m));
    end loop;
  end Simulate;
end Eight_Bit_Parallel_Shift_Register_Mgr;
```

Extended Pin Specification

```
package Pin_Mgr is
  --
  -- This is an abridged version of the pin manager package. It shows only
  -- those declarations that are new or different from the original version.
  --
  -- ..................

  function  Value_of(
    p : pin;
    delay_value: natural := 0)
    return wire_level_value;

  -- ..................

private

  -- ..................

  type time_signal_record;

  type time_signal is access time_signal_record;

  type step_signal_record;

  type step_signal is access step_signal_record;

  type step_signal_record is
    record
      step_value      : wire_level_value := undefined;
      prev_step_value : step_signal;
      end record;

  type time_signal_record is
    record
      time_value      : step_signal;
      prev_time_value : time_signal;
      end record;

  type wire_record is
    record
      value           : time_signal;
      list_of_pins    : pin      := null_pin;
      number_of_pins  : natural  := 0;
    end record;
end Pin_Mgr;
```

Extended Pin Body

```
package body Pin_Mgr  is
  --
  -- This is an abridged version of the pin manager package. It shows only
  -- those declarations that are new or different from the original version.
  --
  -- .....................

  procedure Set_value(s : in out step_signal; v : wire_level_value)
  is
  sl : step_signal;
    --
    -- Function:
    --    Adds a value to a step_signal history.
  begin
    sl := new step_signal_record;
    sl.step_value := v;
    sl.prev_step_value := s;
    s := sl;
  end Set_value;

  procedure Set_value(t : in out time_signal; v : wire_level_value)
  is
    --
    -- Function:
    --    Adds a value to a time_signal history
  begin
    if t = null then t := new time_signal_record; end if;
    Set_value(t.time_value, v);
  end Set_value;

  procedure Set_value(p : pin; v : wire_level_value)
  is
    --
    -- Function:
    --    Assigns a value to a wire record.
  begin
    if Connected(p) then
      Set_value(p.connecting_wire.value, v);
    else
      null; -- raise some exception
    end if;
  end;

  function Value_of(s : step_signal) return wire_level_value
  is
    --
    -- Function:
    --    Determines the value of a step_signal and returns that value.
  begin
  if s = null
      then return undefined;
      else return s.step_value;
      end if;
  end;

  function Value_of(t: time_signal; delay_value: natural) return wire_level_value
  is
  tl: time_signal;
    --
    -- Function:
    --    Determines the value of a time_signal and returns that value.
  begin
    tl := t;
    for i in natural
    loop
        if tl = null
            then return undefined;
            else
              if i = delay_value
                  then return Value_of(tl.time_value);
                  else tl := tl.prev_time_value;
                  end if;
            end if;
        end loop;
  end;

  function Value_of(p: pin; delay_value: natural := 0) return wire_level_value
  is
    --
    -- Function:
    --    Determines the value of a pin and returns that value.
  begin
    return Value_of(p.connecting_wire.value, delay_value);
  end;

  -- .....................

end Pin_Mgr;
```