Appl. Math. and Comp. Sci., 1993, vol.3, No.4, 791–807

LINDHO - USE OF THE OBJECT ORIENTED PROGRAMMING PARADIGM FOR HARDWARE DESCRIPTION AND SIMULATION

AIRES VELOSO, ANTÓNIO DE BRITO FERRARI*

This paper describes LINDHO, an object–oriented hardware description language supporting the description at the higher levels of abstraction, in particular at the architectural level. LINDHO, a much simpler and smaller language than VHDL, is based on C++, easy to be learned by electronic and computer engineers, and has expressive powers not inferior to VHDL. The language is presented informally through examples that try to convey its potential.

1. Introduction

The idea of using formal languages to support the process of machine design is contemporary to the appearance of the first programming languages (Dasgupta, 1989). The main reason for this interest was the conviction that, given the complexity of computer systems, the organization and management of such complexity, together with the need to demonstrate *a priori* the reliability and correctness of a system's design, would require the structuring of the design process itself. In this context, *Hardware Description Languages (HDLs)* made their appearence. They should provide for the formal documentation of the designs and allow the verification of their correct behavior through simulation. More recently the aspects of their integration with a large variety of other design tools, namely synthesis tools became another important consideration.

The early HDLs, designed and implemented in the 1960s, were Register–Transfer Languages (RTL), intended to describe digital systems at the register–transfer level of abstraction. Two of the earliest RTLs were CDL (Chu, 1965; 1972) and DDL (Duley and Dietmeyer, 1968). During the 1970s other hardware description languages appeared. ISPS (Instruction Set Processor Specification) (Barbacci, 1981) is notable among them. An evolution of ISP proposed by Bell and Newell (Bell and Newell, 1971) was to formally describe processor instruction sets. It described behavior at a much higher level of description than other HDLs in existence at the time. ISPS has been quite widely used in the evaluation of processor instruction sets using the technology–independent measures specified by the CFA study whose goal was to select a military Computer Family Architecture for the US Department of Defense (Fuller and Burr, 1977). ISPS descriptions of several computer architectures are provided in (Barbacci and Siewiorek, 1982). Another interesting HDL of this period, supporting more than one level of description, is HILO, one of the first HDLs to be widely available as a commercial product, and supported by an efficient and robust simulator.

* Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, 3800 Aveiro, Portugal

Most of the HDLs developed during the 1960s and 1970s had quite a peculiar syntax, and did not incorporate the advances made by general–purpose programming languages. During the 1980s new HDLs were proposed, most of them directly inspired in structured programming languages, making their use much easier for engineers with the knowledge of programming. These more modern languages provided better support for *multilevel descriptions*, where a single language may be used for specifying hardware structure and behavior at different abstraction levels. HARPA, Pascal based (Veiga, 1984), Zeus, Modula–2 based (German and Lieberherr, 1985), Verilog, C based (Thomas and Moorby, 1991) and VHDL, Ada based (IEEE, 1987) are examples of this approach. Among these different HDLs, Verilog and VHDL are by far the most widely used at present. Verilog is the preferred choice of the microelectronics industry. Developed by some of the developers of HILO, it has been a trademark of Cadence. Recently Cadence put it into the public domain and is trying to make it a IEEE standard. It is supported by a very efficient simulator and by tools for fault simulation and timing analysis. Its C–like syntax makes it easy to use by engineers.

VHDL, developed for the VHSIC (Very High Speed Integrated Circuits) program in the United States, became an IEEE standard in 1987 (IEEE 1076). The US Defense Department puts VHDL descriptions as a requirement for suppliers, a strong motivation for the industry to adopt it. It is the subject of a lot of research interest, including various efforts to extend its use to new areas not targeted by its creators. From these efforts proposals arise to use it for switch–level and analog modelling and simulation, and its use in logic synthesis is being actively investigated (Camposano and Tabet, 1989).

VHDL is closely related to Ada, another DoD product, from which it borrows its main constructs. Like Ada, it is a highly complex language. As a consequence, most implementations of VHDL support only a subset of the language, the so–called sequential VHDL.

Another important drawback of language complexity is the time and effort required to get a reasonable level of expertise in using the language. This aspect has been already a main barrier to the adoption of Ada. Regarded in the early 80s as the future universal programming language that would supersede all others in the course of time, its use remains restricted, while simpler languages based on the Object–Oriented paradigm are gaining a much wider acceptance.

We decided to explore the potential of the Object–Oriented paradigm for hardware description at the higher levels of abstraction, mainly at the architectural level. C++ was chosen as the basis for the language, LINDHO, due to its popularity in the engineering community. The reasons LINDHO has been chosen are as follows:

1. To explore the capabilities of the object oriented programming paradigm for system description and simulation in terms of:
   Reliability – using object oriented programming (information hiding, inheritance and runtime function determination), it is expected that written modules could be *easily tested and reused.*
   Support – it is expected that powerful development environments (debuggers at source code, or even integrated development environments) for C++ become available for the great majority of the machines. These tools will be used for the development of programs in LINDHO.

2. Easy learning
   The designers require a means for specifying design descriptions in a language which is natural to them and does not claim a great effort in learning new syntactic and semantic rules. As many programmers are familiarized with the C language (and it is expected that C++ will become the most widely used object–oriented language), the adoption of LINDHO will not require a great investment.
3. Extendibility
   Anyone working with simulation tools, knows the limited capacities of some tools. Offering complete source code, additional possibilities could be included in a simple way. Simple extension could be implemented by classes, and contributions from other places could be easily integrated.
4. Support for generic modules
   The designer could describe the design at many levels and under many points of view, and due to the repetitive aspect of hardware, need to make generic descriptions, for example multiprocessors architectures, parameterizing the processor types.
5. Portability
   Considering that almost all systems support the ANSI–C compiler (which is enough to install the pre–compiler of C++ of AT&T, or other, such as the GNU complier), it will be easy to install LINDHO in any machine.
6. Efficiency
   Many C++ compilers generate C code in the first step, and the final code generation is transfered to the system C compiler. The actual evolution is based on the use of an intermediate language common to C, Fortran, Pascal, etc. Just as the majority of compilers have sophisticated optimizers, a well optimized code will be generated, using the improvements of compiler techniques without a direct concern about optimization.
7. Full support of behavioral and structural descriptions at any hierarchical level.
8. The description must support the project documentation.
9. The language must integrate easily into a comprehensive design environment interfacing easily with other design tools, including links to silicon implementation.

2. Basic Characteristics of Object Oriented Languages — an Illustration Based on Hardware Description Examples

"On the basis of the history of languages, it can be stated that simplicity and the ease of use is to be achieved not by a lack of structure or by limitless generality but rather by a restriction of objectives and by basing the language around a few well defined simple concepts" (Sorenson, 1985).

Object–oriented programming offers help in design simplification and implementation of CAD systems. A great advantage is its good support for the development of abstract data types. The development of abstract data types and other features of object–oriented languages help the design of any programming system and in particular help the program debugging, by structuring the program in small units. Moreover, , they

help the programmers to reuse codes, making them more productive and making it easier for the users to create new systems from existing code.

The central concept in object–oriented programming is the concept of an **object**. Booch (1986) defines an object as a model of a real world entity that joins data and defines operations over this data. This technology is based on identification of object classes on the system. Classes are described in terms of their behavior and structure. A class isolates the representation of object information in such a way that the users could have access only to the externally defined operations.

Object–oriented programming is not new, its concepts date back to the 1960s. The origins of object–oriented programming stem from the Simula 67 programming language. The *Object* term was first used in the Smalltalk environment developed at Xerox Palo Alto Research Center in the early 1970s.

More recent languages include C++, Objective C, Eiffel and Common Lisp Object System or CLOS. Under these circumstances, all object–oriented languages support three basic features (Wolf, 1991; Stroustrup, 1988; Gorlen *et al.*, 1990):

- Data Abstraction;
- Inheritance;
- Polymorphism (runtime function determination).

2.1. The Development of Abstract Data Types

The importance of data abstraction in the creation of correct and easy maintenable programs is unquestionable. Data Abstraction enables the programmer to use a data type or module without access to its complete implementation, but only to an interface. The separation between the interface and implementation has the following advantages:

- The interface creates documents of the available characteristics of the module.
- The implementers could improve the performance of module execution as far as the changes do not violate the interface.
- Users can replace a module with another, if they have compatible interfaces. For example, one designer can begin with one module and substitute it with a more efficient one.

Figure 1 ilustrates one data abstract type. The development of abstract data types is implemented in LINDHO by classes. One class is a data structure, as the *structure* in C or the *record* in Pascal, plus the methods that manipulate its internal data types. The **object** is an instance of the class. The internal information of the class cannot be directly manipulated by the user. In LINDHO, the members of the class can be either public or private. A public member can be used by other methods that do not belong to the class, while private members can be used exclusively by class members. In the example given, only ninputs() and set_inputs() could manipulate the num_inputs component.

2.2. Inheritance

Classes could be defined by **inheritance**, a mechanism introduced by Simula. One class could be defined in terms of others classes, reusing totally or partially the previous description, providing a greater code reuse and consequently smaller libraries. The derived

```
class gate
{
/* internal variables to implementation */
     int num_inputs;
   ... ... ...
public:    /* functions that implement the interface */
  int ninputs() { return num_inputs; }
  void set_ninputs (int newval)
       { num_inputs = newval; }
  float delay()   { return 1.5 E -9* num_inputs; }
  virtual boolean value (boolean a, boolean b) { }
    ... ... ...
};
```

Fig. 1. One abstract data type in LINDHO.

class inherits all base properties, the internal data types and methods, and includes new proprieties or improves the existing ones. Inheritance could be described by a graph, as shown in Figure 2. A class describing a particular part, such as LS00, is derived from one class by its family circuits (standard or LS) and from another class by function (gate, ALU, etc). Classes that implement a function, derive from the TTL class. The TTL class gives information common to all parts: description of packing, temperature limitations, etc. Descriptions are smaller than they would be in a conventional language, due to the code sharing provided by inheritance. The inheritance graph presented in Figure 2 is an example of multiple inheritance. A class can be derived from more than one base class. Some languages only provide simple inheritance, where one class could be derived from only one other class. Simple inheritance requires larger descriptions. Multiple inheritance is important in any object–oriented application because it provides greater code sharing than simple inheritance.

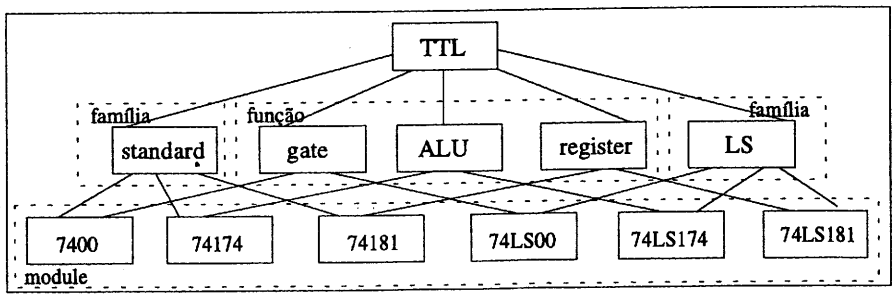

Fig. 2. Graph of multiple inheritance.

Figure 3 shows two new classes for NAND and NOR gates, built from the gate class. The gate class is the base class and nand and nor classes, the derived classes. Nand and nor define gate "**public**", meaning that any public gate member is also a public member of nand and nor. Inheritance could be made more flexible by *friend* relations. If a method external to the class is declared as *friend* inside the class, this function has also acess to the internal (private) class components.

```
class nand : public gate
{
public:
   boolean value (boolean a, boolean b) { return !(a  &&  b); }
};

class nor : public gate
{
public:
   boolean value ( boolean a, boolean b) { return !(a || b); }
};
```

Fig. 3. Definition of class by inheritance.

2.3. Polymorphism

Runtime function determination increases the power of abstract data types and inheritance. Object–oriented languages allow programs to operate over objects without knowing the precise object type, during compilation. For example, we could consider a simulator building lists of all gates, whose outputs need to be calculated at a certain instant. The simulator will run the list of gates and calculate the new value of all outputs from the inputs. Building a separate list for each gate type would not be a good solution. It is simpler to build a single list and mark each gate with its type. A simpler way to implement these possibilities in languages without these mechanisms consists in the inclusion of the type variable in class gate to determine the subclass of each object. The value() method looks the variable to determine what to do.

In Figure 1, in class gate, the value() method is qualified as "virtual", meaning that this method could be redefined in derived classes. In object–oriented languages, the execution system is responsible to determine which version of value() method is to be used in each case. The run time function determination provides the realization of many operations in a collection of related objects in a simpler way, and without knowing its implementation.

3. Syntactic and Semantic Characteristics of LINDHO

LINDHO could be used for the diferent levels of hardware description up to the architectural level, based on the object–oriented programming paradigm. The description of a system based on a simplified version of the DLX architecture (Hennessey, 1990), being implemented at the Department of Electronics, Universidade de Aveiro, is used to illustrate several characteristics of LINDHO.

The Module Concept

Besides the data types present in C++, LINDHO includes the concept of a **module**, its abstraction of physical components of hardware. The module is an adaptation of the class concept to the hardware characteristics. The module could represent any hardware entity with any complexity, a gate, a sequential circuit or a computational system. The module instantiation represents a hardware component. It is constituted by ports, links and other components. The ports are directional, providing the communication with the exterior, through which data flow in/out of the template.

Links are data carriers inside and between modules and provide the communication between the different module components. One module can be used as a component of another module, providing hierarchical descriptions without restrictions on hierarchy (top–down or bottom–up).

The module is constituted by two parts: the interface and the implementation (Fig. 4). The module interface specifies the module external view, the communication channels with the exterior and the characteristics and operation conditions. In Figure 5, the computer interface is represented by ports that provide for the communication with the exterior (terminals, printers, networks, etc.); the port (run) represents the reset input. At any level, the implementation can be described in two generic ways: by behavior and by structure. A module is constituted by methods and internal data types which describe the component organization and/or operation. LINDHO allows both purely behavioral or purely structural descriptions, as well as mixed behavioral and structural descriptions.

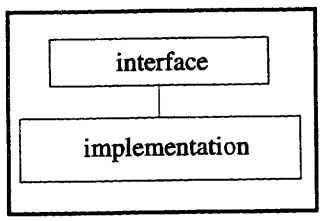

Fig. 4. Module diagram.

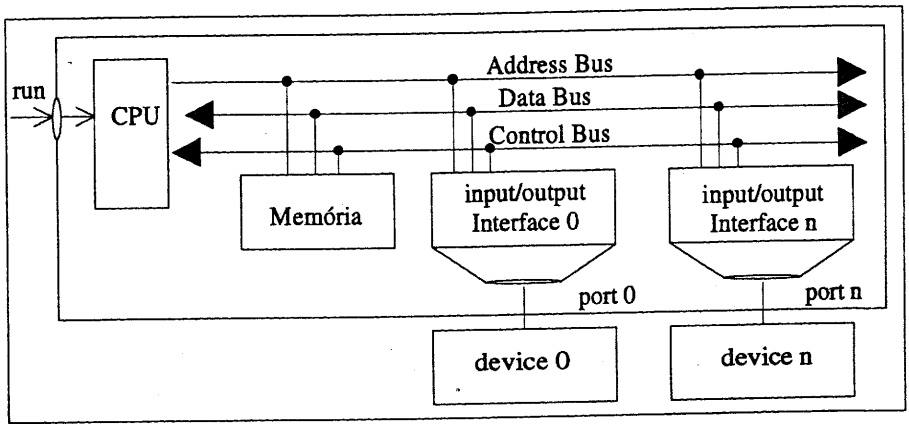

Fig. 5. Computer schematic diagram.

Structural Descriptions

Hardware components are constituted by *ports*, *links* and other **components** of lower hierarchy. Ports constitute the **module interface**. Through them the module communicates with the outside world. **Ports** are directional and may be classified as *in*, *out* or *inout*, according to whether reading, writing or both types of access are provided. In the

absence of one of these qualifiers, the port is classified as a read port. There are special ports: *tri*, *var* and *oc*. These qualifiers apply to hardware signals (connections of bit type), but they can be extended to any type accepted for links, except for the qualifier oc, for which there are some restrictions.

Tri ports are outputs with high impedance capability. In links connected to tri ports, only one of them could be active, otherwise a simulation error is generated. *Var* outputs look as *out* outputs. A link could have only one *out* output, but may coexist with many var outputs, and under this condition the present link value in each moment is the value of the last output which has made a write access.A *var* qualifier may be used as an option to the *tri* qualifier in abstract descriptions sparing the programmer the task of releasing **tri** outputs.

Oc outputs in LINDHO have affinity with the corresponding open–collector logic in hardware. The base type of the output cannot be structured. They must be scalar. At any time the present link value, where several **oc** ports are connected, matches the smallest value from the outputs connected to the link. Hence, it can be stated that the inputs can be connected to a link, the **out** outputs can be exclusively connected to inputs, the **tri**, **var** and **oc** outputs can be connected to inputs and outputs of the same type. The break of these rules generates simulation errors.

Ports are characterized also by their length, i.e. the number of data flows they support; and their base type, i.e. the range of values which may flow through them. Links are data carriers, and support the communication between modules. Links can be of scalar (integer, char, etc.) or structured types : (Links of bit arrays, Link of records, etc.).

```
1   module DLX
    {
          /* cpu registers */
        word  PC, IAR, MAR, MDR, TEMP, IR, RegFile[32];
5       fields  IR { [31 .. 26] opcode, [25 .. 21] rs1, [20 .. 16] rd };
    public:
7       DLX(bit reset, word ADDR_BUS, word DATA_DUS, bit RW, ....);
        void execute();
          /* methods associated to instruction execution */
        void lb();
        void lbu();
12      void lhu();
        void lw();
        void add();
        void sub();
        ... ... ...
  };
```

Fig. 6. Abstract data types representing part of the DLX processor (Hennessy and Patterson, 1990; Mariott and Ferrari, 1992).

Figure 6 shows the DLX representation in LINDHO. Among the methods presented in the public part, there is a special method (line 7 of Fig. 6), whose name matches the module name. This method is automatically invoked when an object of DLX type

is instantiated, and it automatically builds and initiates the object. Figure 7 shows the system description presented in Figure 5. The system is formed by one DLX component, one memory module and several interfaces which support the communication with the outside world. This components are connected by data, control and address buses, represented in the model by links. Components are the lowest level module instances in the structural hierarchy and provide the basic building blocks.

```
1    typedef bit[32]    word;

3    typedef struct
     {
      in bit RXD, DTR;
      out bit TXD, CTS;
     } RS232C;

9    system:: system(in bin run,  RS232C  port0, ...)
     {
11   link of word AddressBus;
     link of word DataBus;
     link of bit rw, ...;

15   DLX cpu(run, AddressBus, DataBus, rw, ...);
     Memoria mem(ABus[1 . . 21], dataBus, rw, ...);
     Interf_serie0 Interface0(AddressBus[19 . . 21], DataBus, rw, port0, ...);
     ... ... ...
     }
```

Fig. 7. Structural description of the computer presented in Figure 5.

The cpu declaration in line 15, Figure 7, realizes two functions : automatically invokes the DLX module constructor (line 7 of Fig. 6), and gives as parameters the efectively connected links. Its execution creates the cpu and simultaneously states the connections with the system. The links connected to the module must be compatible with the base type of the ports declared in the interface. It is important to emphasize that the information sent as parameters to the constructor is used for defining the connections of the instantiated module with the system. In class type object instantiation, the arguments sent to the constructor as parameters are used for the object's automatic initiation. As each module has only one interface with the exterior, each module provides one exclusive constructor, while one class could have several constructors, each one providing the different options of building and initiation of possible instances. For class and module instances, the associate constructors realize two functions: object memory allocation and attribution of initial values. Nevertheless, the object instantiation of derived classes invokes the associated constructor, and this one automaticaly and recursively invokes the base class constructor(s). For derivate module instances, the associated constructor never invokes the base module constructors. This is so because besides the former two functions, the module constructors have a third function, to establish the instance connection with the system. The module constructor must allocate and initiatie all object memory areas, including those connected to the support of the base module inherited information.

Behavioral Descriptions

The module transfer function and its timing characteristics constitute the module behavior. Given a precise behavioral description and its input data, it is possible to foresee the output and relative delay. Due to the great parallelism that could exist inside a module, LINDHO behavioral descriptions are based on the data flow model.

Statements "SEQ" and "PAR"

Complex operations result from the association of less complex operations. The complex operation components may be activated in sequence or in parallel. In LINDHO, the parallelism is described by the *PAR* statement. By default and in the absence of the *PAR* statement the execution is meant to be sequential, but when necessary the *SEQ* statement can be used (see exemple in Fig. 9). The use of SEQ statement, is in some cases redundant and provided for greater clarity in descriptions. The sequential blocks and the concurrent ones, framed by *PAR*, involve the start and end time notions. For concurrent blocs, the start time is the same for all instructions and the end time matches the execution of all its components. For sequential blocs, the start time matches the execution of the first instruction and the end time the execution of the last instruction.

Timing Description Mechanisms

- **After**

Timing characteristics are described in the model, by associating to each data flow the time for its execution. In LINDHO, the statement "*after*" provides this action.

The statement "*after*" accepts as argument temporal expressions, providing for the simulation of complex delay models. The syntax of LINDHO expressions including the "*after*" statement is :

[**transport**] expression *after* time_expression

where expression means the executed operations in the associated data flow, and time_expression is a timing expression, whose result, calculated during the execution or defined during the compilation (in case of constants), will determine the delay time introduced in the data flow. The reserved word "*transport*" is optional, and its use specifies that the delay associated with the statement is to be constructed as transport delay. Transport delay is characteristic of devices (such as transmission lines) that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration. If the reserved word is not presented, inertial delay is assumed. Inertial delay is characteristic of switching circuits: a pulse whose duration is shorter than the switching time of the circuit will not be transmitted.

- **Wait**

The *wait* statement provides for the description of data flow synchronization. It has the following syntax:

wait expression_time *while* expression

where **expression_time** is an expression that defines the maximum hanging time of the associated data flow, and **expression** is a Boolean expression, which conditions the hanging.

Event Control

In LINDHO instruction execution can be synchronized with the change of a variable or expression value or the occurrence of a certain event, using statements with the following syntax:

event_controlled_statement :
    event_control statement
event_control :
    @ identifier
    @ (expression)

Figure 8 presents the memory write cycle of DLX processor. A descritption in LINDHO, showing the use of *PAR* and *SEQ* statements is presented in Figure 9.

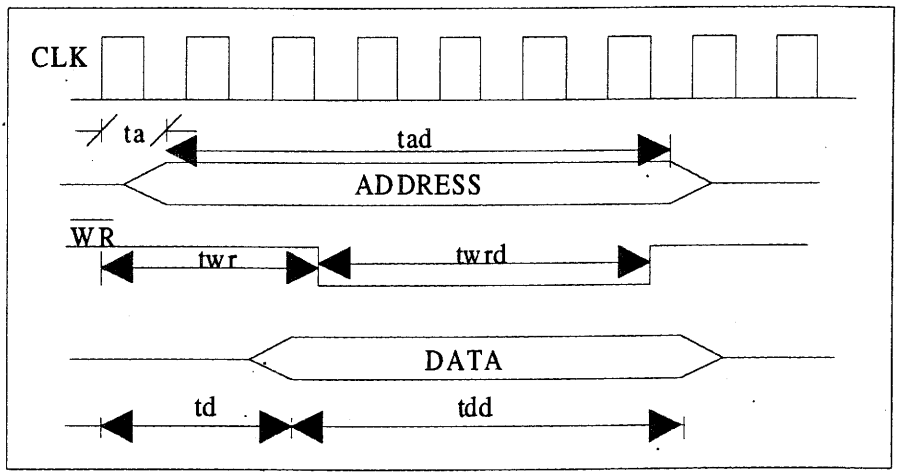

Fig. 8. Write cycle of one processor.

In the processor 'write' method, the variables AddrBus, DataBus and WR, are module internal variables and represent the information put into the address bus, the data bus and the write control line.

When the write method is invoked, it waits for the event represented by the next positive clock transition; then the three blocks framed by the *SEQ* statements are executed simultaneously, with instructions within each block executed sequentially. The parallel block ends its execution after the address bus deactivation. The number b32*Z corresponds to one 32 bit binary entity (with all bits in high impedance).

Access to the Current Simulation Time

LINDHO provides access to the current simulation time by the system variable **_time**.

Operator Redefinition

LINDHO provides for the adoption of a special meaning for any of its operators when applied to abstract data types. We could consider the mvl class (Fig. 10a), which implements 8–value logic.

```
void DLX::write(word data, word address)
{
  @(clock -> 1)
  PAR
  {
    SEQ
    {
      AddrBus=address after ta;
      AddrBus=b32*Z after tad;
    }
    SEQ
    {
      WR=0 after twr;
      WR=1 after twrd;
    }
    SEQ
    {
      DataBus=data after td;
      DataBus=b32*Z after tdd;
    }
  }
}
```

Fig. 9. DLX write cycle description in LINDHO.

```
class mvl {
  enum MVL {
   L, // logic low
   H, // logic high
   R, // rising from low to high
   F, // falling from high to low
   U, // undefined
   A, // ambiguous
   E, // error
   Z, // high impedance }
  MVL m;
public:
  mvl(char);
  mvl();
  mvl operator & (mvl);
  int operator==(char);
};
```

a)

```
mvl& resolution_mvl(mvl* source, int n)
{
  mvl& resoult=source[0];
  int nZ=0;
  for (int i=0;  i<n; i++)
  {
    if (source[i] == 'Z')
      continue;
    result=source[i];
    nZ++;
  }
  if (nZ>1)
    result = *new mvl('E'); // error
  return result;
}
```

b)

Fig. 10. Operator redefinition:
a) mvl class (8 value logic),
b) mvl resolution function.

LINDHO provides the logic product of two mvl objects, using the operator &

```
mvl a('0'), b('1'), c;
c = a.operator&(b);
c = a & b;
```

The first line declares three mvl objects. The a and b objects are built and initialized by the mvl(char) constructor, while for the c object the mvl() constructor is invoked. The last two lines are equivalent. The operator redefinition makes it easy to use abstract data types.

Additional Possibilities

Besides the pre–defined types in the language, the designer could define new abstract data types when necessary. LINDHO has basically two data types: the scalar types and the compound types. The scalar types are pre–defined and include: bit, integers, characters, floating point types, enumerates and physical types (time and length measures). The compound types include arrays, unions, classes and modules. All the C++ flexibility, namely the run time function determination, the operator redefinition, the information hiding supported by classes and now extended to modules, etc. is kept by LINDHO.

Some inherited concepts from C++ are reinforced. The possibility of definition of enumerates inside classes introduced in the more recent C++ versions, is improved. (see example in Fig. 6). Enumerates can now represent ordered values, system status or processor codes. Similary, the argument of the switch statement is not limited to scalars and extends to classes, with the operator == defined to the related constants.
Fields provides for the discrimination of the different components of a record, making possible the manipulation of any component in an automatic and independent way. Line 5 of Figure 6 shows the use of *fields* discriminating the different parts of one instruction format accepted by the DLX processor.

Control structures are inherited from C (C++), the tradicional *if then else, while, for, do while*. The subprograms are implemented by functions returning or not a value. LINDHO supports concurrent subprograms (functions) whose execution can be extended by several simulation time steps. The functions are declared by their name, parameter types, the type of return value and a sequence of expressions implementing an algorithm. Functions used for Link resolution are not invoked explicitly, but are invoked implicitly where necessary to resolve multiple values driving a signal into a single value.

The statement:

typedef resolution function_name$_{opt}$ decl_specifiers declarator_list

is an extension of a typical C++ typedef statement providing the association of a resolution function with a type. Figure 10b defines a Link resolution function for type mvl, whose purpose is to verify that only one signal driver is driving with a non 'Z' value, and to return that unique value as the bus's resolved value.

Parameterizable Descriptions

Supporting parametrizable descriptions, LINDHO provides the description of regular structures. Parameterizable types are not yet implemented in the C++ language, and its use is simulated with the macroprocessor. On the other hand, it has been experienced

that in some machines the UNIX macroprocessor has presented a strange behavior for very long macros[8]. LINDHO has its own macroprocessor, overcoming those limitations. The parameterization of descriptions has the advantages of better readability and smaller libraries, giving the designer the possibility of using higher level descriptions.

The parameterization provides the defintion of a family of types or functions. It works as a generic container of types and functions, whose specific type appears as a parameter and is solved during compilation. It is possible with parameterization to define an array processor without indicating the number of processors and even their type. These data will appear as parameters in the formal description of modules.

The syntax of generic description declarations is the following:

template–declaration :
template < template–argument–list > declaration

where *'declaration'* declares a function, a class or a module.
There are no restrictions in types used as parameters. They could be scalar or structured, as well as constant expressions, object, function or class/module addresses, giving great flexibility to the programmer.

Template, with the arguments before the definition, is a reserved word used to identify a parameterizable declaration and to guarantee that the parameter types are lexically introduced before their use, making the work of the parser easier.

Figure 12 shows an n–bit parity checker built with two input XOR gates. The private part of the PARITY_CHECKER module supports the basic information about the module, i.e. the px pointer points to a list of XOR pointers and nxor represents the number of XOR gates. The gate XOR is supported in a specialized library, and its constructor has the form:

XOR(in bit, in bit, out bit);

The public part has one method, the constructor (7th line), whose implementation apears after line 10.

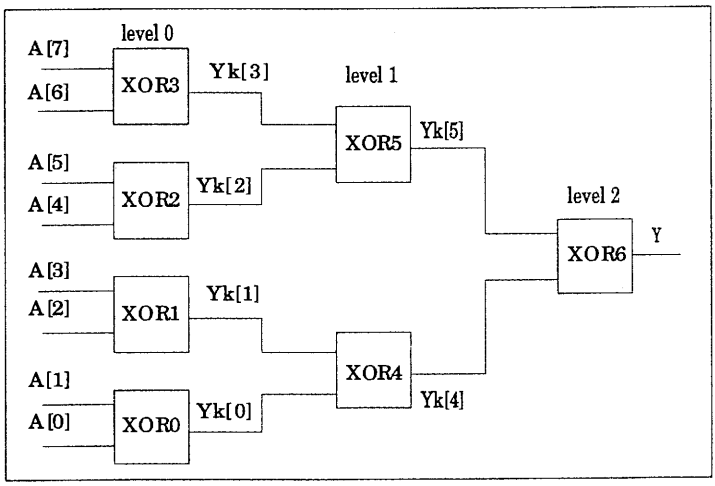

Fig. 11. 8–bit parity checker.

```
1   template <int NBIT> module PARITY_CHECKER
    {
      /* NBIT must be power of 2 */
      XOR* px;
      int nxor;
6   public:
      PARITY_CHECKER(in bit A[ ], out bit Y);
      ... ... ...
    };

11  template <int NBIT> PARITY_CHECKER:: PARITY_CHECKER(in bit A[], out bit Y)
    {
      // determination of number of gate levels;
14    int nbit=NBIT;
      nxor=0;
       for (int nlevel=0; nbit>1; nlevel++)
      {
        nbit/=2;
        nxor+=nbit;
21    }

      // space allocation for pointer to pointer to XOR gates
      XOR**ppx=new XOR*[nxor];
      px=ppx[0];

27    // instantiation of 1st level gates
      int nxor=NBIT/2;

      bit*yk=new bit[nxor]; // connected to XOR gates' outputs
31    int k=0;
      for (int i=0; i<nxori; i++)
      {
         ppx[k]=new XOR (A[2*i], A[2*i+1], yk[k]);
        k++;
36    }

      int m=0;
      // instantiation and interconnection of the other XOR gates
      for ( int l=1; l<nlevel; l++)
41    {
        nxori/=2;
        for( i=0; i<nxori; i++ )
        {
            ppx[k]=new XOR ( yk[m++], yk[m++], yk[k] );
46          k++;
        }
      }
      Y=yk[--k]; // module output
    }
```

Fig. 12. Generic description of a n-bit parity checker.

The description is completely generic, the number of bits being parameterizable. Each level of gates compares groups of pairs of input lines, so each level has half the number of inputs of the preceeding level. Lines 14 to 21 of Figure 12 represent the determination of the number of gate levels (nlevel) and the total number of gates (nxor). The variable ppx, an array of nxor pointers to XOR gates, is declared at line 24. Lines 27 to 36 present the instantiation of the first level of gates. The variable yk, an array supporting the nxor XOR gate outputs, is declared at line 30. The rest of the description presents the interconnection and the instantiation of other gates.

Figure 11 shows a typical example of a triangular XOR array of gates implementing an 8–bit parity checker.
The declaration of a 16–bit parity checker, that interfaces as input 16 lines of bit type (i) and one output line (o) of type bit, would be:

PARITY_CHECKER < 16 > parity_checker(i, o) ;

Performance Measurements and the Collection of Statistics

Support for performance measurements is a fundamental requirement for the evaluation of system architectures. LINDHO provides the gathering of any simulation results with the **stat** statement, which makes it possible to collect several types of statistics, namely average values (**mean**), range of values (**range**), the variance (**variance**) and histogramming (**histogram**).

4. Conclusions

LINDHO has a syntax similar to C++, easy to be learned by any engineer who is familiar with C. The language includes all C++ facilities augmented with the mechanisms necessary to hardware simulation and description, namely, two kinds of delays (inertial and transport) the *module* concept, the concurrent function concept, the resolution functions, new control structures, the *fields, wait, after,* @ statements. These extensions, jointly with the C++ facilities, should make it possible to develop powerful CAD environments, supported by LINDHO.

A compiler for the language has been developed written in C++. At the moment, an event–driven simulator is being developed, using LINDHO as an interface for system description. The development of a synthesis tool accepting LINDHO descriptions as input will start shortly.

References

Barbacci M.R. (1981): *Instruction set processor specifications (ISPS): The notation and its applications.* — IEEE Trans. Computers, v.C–30, No.1, pp.24–40.

Barbacci M.R. and Siewiorek D.P. (1982): *The Design and Analysis of Instruction Set Processors.* — New York: McGraw–Hill.

Bell C.G. and Newell A. (1971): *Computer Structures: Readings and Examples.* — New York: McGraw–Hill.

Booch G. (1986): *Object–oriented development.* — IEEE Trans. Software Eng., v.SE–12, No.12, pp.211–221.

Camposano R. and Tabet R.M. (1989): *Design representation for the synthesis of behavioral VHDL models.* — Proc. Conf. *Computer Hardware Description Languages*, North–Holland, pp.49–58.

Chu Y. (1965): *An Algol–like computer design language.* — Comm. ACM, v.8, pp.607–615.

Chu Y. (1972): *Computer Organization and Microprogramming.* — New York: Prentice–Hall.

Dasgupta S. (1989): *Computer Architecture – A Modern Synthesis v.2 – Advanced Topics.* — New York: John Wiley and Sons.

Duley J.R. and Dietmeyer D.L. (1968): *A digital system design language (DDL).* — IEEE Trans. Computers, v.C–17, No.9, pp.850–861.

Fuller S.H. and Burr W. (1977): *Measurement and evaluation of alternative computer architectures.* — Computer, v.10, No.10, pp.24–35.

German S.M. and Lieberherr K.J. (1985): *Zeus: A language for expressing algorithms in hardware.* — IEEE Computer, v.18, No.2, pp.55–65.

Gorlen K.E., Orlow S.M. and Plexico P.S. (1990): *Data Abstraction and Object–Oriented Programming in C++.* — New York: John Willey and Sons.

Hennessey I. and Patterson D.A. (1990): *Computer Architecture – A Quantitative Approach.* — Morgan Kaufman.

IEEE (1987): IEEE Standard VHDL Language Reference Manual, IEEE Press, pp.1076–1987.

Marriott A.P. and Ferrari A.B. (1992): *ARICH experience in teaching computer architecture and VLSI design.* — Third Eurochip VLSI, Design Training Workshop, Grenoble, France, pp.260–267.

Sorenson J.P. (1985): *The Theory and Practice of Compiler Writing.* — Singapore: McGraw–Hill.

Stroustrup B. (1988): *What is O.O.P. ?.* — IEEE Software, May 88, pp.10–20.

Thomas D.E. and Moorby P. (1991): *The Verilog Hardware Description Language.* — Boston: Kluwer Academic Press.

Veiga P.M.B. (1984): *Uma Linguagem de Projecto para Especificação e Simulação Hierarquica de Systemas Digitais.* — Dissertação de tese de Doutoramento no Instituto Superior Técnico de Lisboa.

Wolf W. (1991): *Object Oriented Programming for CAD.* — IEEE Design & Test of Computers, March 91.

Received January 28, 1993
Revised December 21, 1993