CONLAN-a formal construction method for hardware description languages: language derivation

R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, P. Skelly

CONLAN—A formal construction method for hardware description languages: language application

by ROBERT PILOTY
Technische Hochschule Darmstadt, FR Germany

MARIO BARBACCI
Carnegie-Mellon University

DOMINIQUE BORRIONE
Universite de Grenoble, France

DONALD DIETMEYER
University of Wisconsin-Madison

FREDRICK HILL
University of Arizona-Tucson

and

PATRICK SKELLY
Honeywell, Phoenix

1. INTRODUCTION

CONLAN is a formal semantic and syntactic base for the description of all phases in the design and documentation of digital systems. Previous papers[1,2] have presented the basic concepts and models underlying the whole CONLAN language family. On the example of the construction of Base CONLAN, the first level which can be of some operational interest to the hardware designer, it has been shown how a coherent set of special purpose application languages can be derived from a common mathematical base, Primitive Set CONLAN.

This paper emphasizes the logic system designer's point of view. By this, we mean the user of a CONLAN language rather than the definer of a member of the CONLAN language family. The reader is assumed to know the available object types and operations in Base CONLAN, which will be, for the purpose of this paper, considered as a user language.

The authors are convinced that the CONLAN concepts are general and versatile enough to be applicable to several design methodologies. In particular, the necessity of being able to decompose a system into modules is now widely recognized. The first part of this paper presents the CONLAN basic user's segmentation mechanism, called DESCRIPTION. Subsequent parts show how the same "hardware object" can be either considered to be a DESCRIPTION, an ACTIVITY or an object TYPE, depending upon the level of abstraction of its model written in CONLAN.

2. DESCRIPTION SEGMENT

Any piece of CONLAN text which models a system or a subsystem is written as a DESCRIPTION segment. A DESCRIPTION can represent as small a piece of hardware as a NAND gate, or as big a system as a computer network. The main characteristic of a DESCRIPTION segment is that it is considered as a module in itself, independently of whatever environment it may be inserted in; the best visualization of a DESCRIPTION segment is an integrated circuit.

A DESCRIPTION segment knows only its interface carriers, internal carriers and operators, and eventually subsystems into which it is further decomposed. It has no knowledge of global carriers, and all communications with its environment are done through its interface.

Several subsystems may be very similar in function and structure, except for some very specific parameter. For instance, an 8 bit and a 16 bit parallel adder perform the same function, and are of the same nature, except for the dimensions of their operands. CONLAN does not require the user to define two distinct DESCRIPTION segments in such a case, but provides a facility to indicate ATTribute parameters. A DESCRIPTION segment definition with attributes therefore describes a family of modules. From that family, instances with specific attribute values will be used in a particular context.

The reference language for all segments in this paper is taken to be Base CONLAN (bcl), as illustrated in the first example.

2.1 Structure of a DESCRIPTION segment

The minimum amount of information a DESCRIPTION segment should contain is:

1. the name of the segment
2. the list of its interface carriers, specifying their type and direction (IN for input, OUT for output, INOUT for bidirectional)
3. operation invocations showing how the signals of the OUT and INOUT carriers are related to the signals of the IN, INOUT, and internal carriers.

A very simple example is a 2 input, 1 output unit delay nand gate:

```
REFLAN bcl
DESCRIPTION nandgate1(IN x,y: btm1, OUT z: btm1)
BODY
z .= (x ¬∧ y) Δ 1
ENDnandgate1
```

where Δ is the delay operator, .= denotes terminal connection, and btm1 has been defined as the type "Boolean terminal with default value 1" [2].

One could wish to make the value of the delay an attribute of the DESCRIPTION, so as to write once and for all the model of a 2 input − 1 output nand gate with any propagation delay. In this case, the interface list is augmented with an ATTribute section containing the delay parameter d, which must be fixed at compile time.

```
DESCRIPTION nandgate(ATT d: pint, IN x, y: btm1,
OUT z: btm1)BODY
z .= (x ¬∧ y) Δ d
ENDnandgate
```

More complex descriptions can be written, in terms of an interconnection of instances of nand gates. This is shown on the following model of the 3 input, 1 output multiplexer displayed in Figure 1.

```
DESCRIPTION multiplex (IN a, b, c: btm1, OUT d:
btm1)BODY
  DESCRIPTION nandgate1 (IN x, y: btm1, OUT z:
  btm1)
  BODY z .= (x ¬∧ y) Δ 1 ENDnandgate1
DECLARE i, j, k: btm1 ENDDECLARE
USE g1, g2, g3, g4: nandgate1 ENDUSE

g1.x. = a,   g1.y. = a,      i .= g1.z,
g2.x. = b,   g2.y. = a,      j .= g2.z,
g3.x. = i,   g3.y. = c,      k .= g3.z,
g4.x. = j,   g4.y. = k,      d .= g4.z
ENDmultiplex
```

Four distinct, explicitly named gates are specified, in the USE statement, as instances of nandgate1. Local terminals i, j, k are declared. The interface terminals of the individual gates are referenced through dot notation, and individually connected to interface and local carriers of multiplex in the operation invocation part of that description.

We now have illustrated the general constituents of a DESCRIPTION segment, which is divided into 5 parts, of which only the first one must not be empty.

1. Description header, consisting of
   DESCRIPTION
   description name
   (list of attributes and interface carriers)
   BODY
2. Assertions part, consisting of
   ASSERT list of predicates ENDASSERT
   The predicates may be static conditions on attribute parameters, to ensure that the model is meaningful. They may also express dynamic constraints (functional and time dependencies) on interface and/or internal signals. All predicates must be true at every point in time during the simulation of an instance of a DESCRIPTION segment.
3. Definition part, defining operation and description segments to be used for portraying the behavior and structure of the description being defined. These definitions may be locally written; or they may be brought from an existing library of segments, in which case they are said to be EXTERNAL, and only their name and parameter/interface are written.

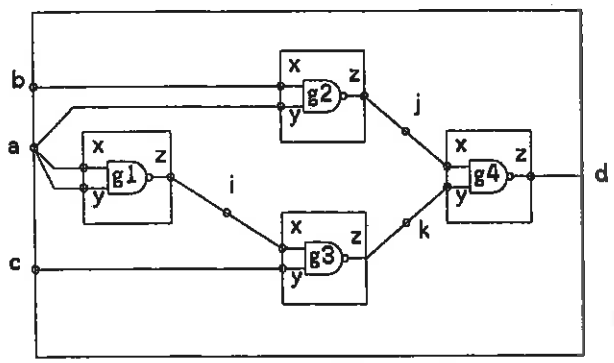

Figure 1(a)—Interconnection as operation invocation.

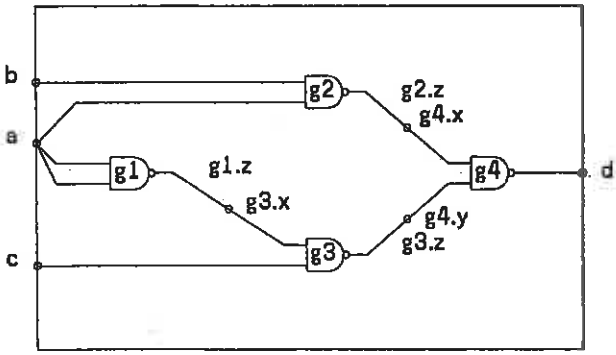

Figure 1(b)—Permanent interconnection.

4. Internal objects, where local carriers are named and typed in a DECLARE statement. If the description is constructed from smaller descriptions (which must then be defined in part 3), instances of those descriptions are named in a USE statement.
5. Operation invocation part, describing the input/output behavior of the hardware unit. It terminates with the usual END keyword, possibly followed by the description name.

3. INSTANTIATION AND INTERCOMMUNICATION

Descriptions may be instantiated and interconnected in an enclosing description in either of two different ways.

3.1 Intercommunication in the operation invocation part

The first method was shown on the multiplex description above. Instance names are brought to existence in the USE statement, and actual attributes (if any) are given at that point.

This implicitly declares all the interface carriers of the instances, which will be referred to, in the enclosing description, by compounding the instance name with the formal interface carrier names.

Intercommunications between enclosed instances of descriptions, and communications between instances and the enclosing description, are stated in the operation invocation part of the enclosing description.

If all interface carriers of enclosed instances are terminals, and all these terminals are interconnected, then an explicit wiring list is being written, as in multiplex. In that case, intermediate carriers i,j,k are not necessary. The nandgate description, previously defined and assumed in the library, is used to show the EXTERNAL and instantiations with actual attributes in the equivalent multiplex2 description below:

```
DESCRIPTION multiplex2 (IN a, b, c: btm1, OUT d:
btm1) BODY
EXTERNAL DESCRIPTION nandgate
ENDEXTERNAL
USE g1, g2, g3, g4: nandgate (ATT 1) ENDUSE
g1.x .= a,     g1.y. = a,     d .= g4.z,
g2.x .= b,     g2.y. = a,
g3.x .= g1.z,  g3.y. = c,
g4.x .= g2.z,  g4.y. = g3.z
ENDmultiplex2
```

At some more abstract level of a design decomposition of a description might be more conceptual than physical. Then, interface carriers may be of types other than terminal, like registers and variables for instance. And communications to enclosed instances of descriptions would be done through other operations: register transfers, variable assignments and the like. In a more abstract model, communications could even be under the scope of conditions, with some hardware implied, rather than actually defined. For instance, in the structure of Figure 2, one could wish to consider registers a and b to be interface carriers of the alu, and to portray the behavior rather than the structure of the local memory and control part.

A very simplified model is shown below:

```
DESCRIPTION structure BODY
  DESCRIPTION alunit (IN a[0:7], b[0:7]: brtv(0),
  ctl[0:3]: btm0, OUT r[0:7], cnd[0:1]: btm0),
  ...
  ENDalunit
...
DECLARE local_memory [0:7, 0:15]: brtv(0),
  ad: tml(int,0),
  ca, cb, cm: btm0,
...
ENDDECLARE

USE alu: alunit ENDUSE
...
IF ca THEN alu.a <- local_memory [,ad] ENDIF,
IF cb THEN alu.b <- local_memory [,ad] ENDIF,
IF cm THEN local_memory [,ad] <- alu.r ENDIF,
...
ENDstructure
```

where <- denotes rtvariable transfer, btm0 has been defined as the type "Boolean terminal with default value 0," and brtv has been defined as the type "Boolean real time variable" [2]. The rtvariable transfer is accomplished in one unit of real time.

3.2 Permanent intercommunication

The second method by which an instance of a description can communicate with its environment is by considering that its interface carriers permanently drive or are driven from carriers of the enclosing description or interface carriers of other instances. This is indicated in the USE statement: the instance name is followed by the list of the environment carriers which the implicitly declared interface carriers drive or are driven from.

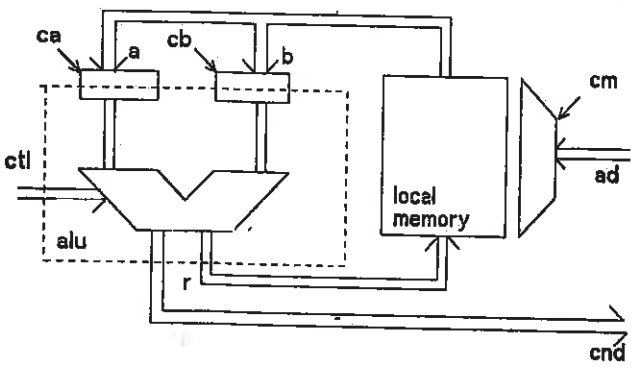

Figure 2

```
DESCRIPTION multiplex3 (IN a, b, c: btm1, OUT d:
btm1) BODY
EXTERNAL DESCRIPTION nandgate
ENDEXTERNAL
USE  g1 (IN a, a, OUT g3.x),
     g2 (IN b, a, OUT g4.x),
     g3 (IN g1.z, c, OUT g4.y),
     g4 (IN g2.z, g3.z, OUT d): nandgate (ATT 1)
ENDUSE
ENDmultiplex3
```

The above example is equivalent to multiplex2, previously studied.

4. ABSTRACTION LEVELS IN CONLAN MODELS

DESCRIPTION segments provide the user with a structuring means to describe how a digital system is constructed from a set of components (network description). ACTIVITY and FUNCTION segments on hand, give him the way to express what signal state changes occur in the carriers of the digital system at every given point in time (behavior description). CONLAN does not go down to the electronic component level of a description; the most detailed description the user can write is therefore a logic gate network. The hardware designer however is not necessarily interested in gate-level details. He might not even know, at some stage of a design, neither how many components he will be actually using, nor which specific ones, in some part of his system.

CONLAN provides maximum flexibility for stepwise refinement of a design. Instances of purely behavioral descriptions can be interconnected with instances of detailed network descriptions. Moreover, inside the same DESCRIPTION segment, some parts may be expressed in terms of clearly identified hardware elements, while other parts might be abstract.

Our purpose in this section is to show how the same physical object may be more or less precisely specified, depending upon the CONLAN construct for which it stands.

4.1 Duality between TYPE and DESCRIPTION

Internal objects which are named in a description, and which therefore can be inspected for state changes, are of two categories: (a) carriers, which are elements of types and, (b) instances of description.

Although the user will probably not often define new types, he may select (or ask for) a language where some types are primitive, or decide to construct an object as a description in terms of more elementary objects.

Flip-flops as description segments

For instance, Base CONLAN has neither primitive flip-flop nor register types. But these can be constructed as DESCRIPTION segments. Assuming unit delay gates, the network description of a rising edge triggered D-flip-flop would read:

```
DESCRIPTION dff1 (IN c,d: btm1 OUT q,nq: btm0)
BODY
  DESCRIPTION nand2 (IN x,y: btm1 OUT z: btm0)
  BODY
  z .= x Δ 1 ¬∧ y Δ 1
  ENDnand2
  DESCRIPTION nand3 (IN w,x,y: btm1 OUT z: btm0)
  BODY
  z .= ¬(w Δ 1 ∧ x Δ 1 ∧ y Δ 1)
  ENDnand3
USE g1, g2, g4, g5, g6: nand2, g3: nand3 ENDUSE
g1.x .= g4.z,   g1.y .= g2.z,
g2.x .= g1.z,   g2.y .= c
g3.w .= g2.z,   g3.x .= c,     g3.y .= g4.z,
g4.x .= g3.z,   g4.y .= d,
g5.x .= g2.z,   g5.y .= g6.z,
g6.x .= g5.z,   g6.y .= g3.z,
q .= g5.z,      nq .= g6.z
ENDdff1
```

The wiring of the NAND gates is clear enough. Since all gate delays and initial values are identical, real time oscillation can be expected initially.

While the following description may be thought to provide the same internal organization as the description above, it essentially provides a signal computation model of the D-flip-flop.

```
DESCRIPTION dff2 (IN c,d: btm1 OUT q,nq: btm0)
BODY
DECLARE e, f, g, h: btm1 ENDDECLARE
e .= h Δ 1 −∧ f Δ 1,
f .= e Δ 1 −∧ c Δ 1,
g .= ¬(f Δ 1 ∧ c Δ 1 ∧ h Δ 1),
h .= g Δ 1 ¬∧ d Δ 1,
q .= f Δ 1 ¬∧ nq Δ 1,
nq .= q Δ 1 ¬∧ g Δ 1
ENDdff2
```

At some more abstract level, we want to think of a flip-flop as a 2 state device. Then, correct initial values are assumed inside the module. Oscillations due to default in initialization or wrong setting of data and clock inputs, can no longer be displayed on such a simplified model. However, it is possible to ASSERT properties on input signals to ensure that the description is reacting correctly when the environment produces acceptable waveforms. The following description provides a behavioral 3 units of time delay model of the d-flip-flop:

```
DESCRIPTION dff3 (IN c,d: btm1, OUT q: brtv(0), nq:
brtv(1)) BODY
ASSERT ¬ c ∨ d = d Δ 1) ENDASSERT
                "/we assert stability of d when c is 1/"
IF ¬ c Δ 4 ∧ c Δ 3 THEN q <− d Δ 3, nq <− ¬ d
Δ 3 ENDIF
ENDdff3
```

The IF condition detects the rising edge of the c input. The transfers introduce a 3 unit propagation delay, to represent 3 layers of unit delay nand gates. The two values of q provide the two states we desired. nq has been kept only for interface equivalence with previous descriptions. Other varieties of flip-flops may be modeled similarly to the above 3 models. Just one last behavioral example is provided, for a master- slave flip-flop. In the ASSERT statement, we insist that j and k do not change value when the clock is high. A 4 unit propagation delay stands for the 4 layers of nand gates.

```
DESCRIPTION jkff3(IN c,j,k: btm0,OUT q: brtv(0),
nq: brtv(1)) BODY
ASSERT ¬ c ∨ (j = j △ 1 ∧ k = k △ 1)
ENDASSERT
IF ¬ c △ 5 ∧ c △ 4 THEN
  IF j △ 4 ≠ k △ 4 THEN q <− j △ 4, nq <− k △ 4
  ELIF j △ 4 THEN q <− nq, nq <− q ENDIF
  ENDIF
ENDjkff3
```

Instances of such descriptions may now be used to represent the existence of flip-flops and registers in a circuit as shown in Figure 3, where a register is depicted as an array of flip-flops.

```
DECLARE fa, fb: btm0 ENDDECLARE
USE a[0:15], b[0:15], r[0:15]: dff2 ENDUSE
OVER i FROM 0 TO 15 REPEAT
  r[i].c .= fa ∨ fb,
  r[i].d .= a[i].q ∧ fa ∨ b[i].q ∧ fb
ENDOVER
```

Here, the OVER statement has been used to compactly write repetitive connections. A macro-generation into 16 pairs of connections is expected from the software.

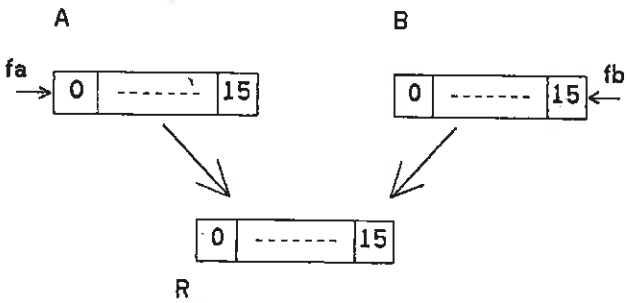

Figure 3(a)—Global structure.

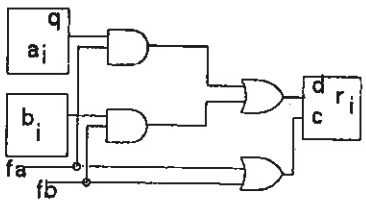

Figure 3(b)—Detail of routing.

Flip-flops as carrier types

In register transfer level hardware modeling, emphasis is no longer on data paths between memory elements, but rather on state changes. Taking this point of view, the hardware designer sees flip-flops and registers as carriers rather than descriptions, and portrays state transitions by explicit operations rather than by signal changes on interfaces. The following type segment illustrates the definition of a jk-flip-flop as it would appear in a language definition [1]. The three activities provide for the three combinations of jk input values which change the flip-flop state. Unit delay is assumed for convenience.

```
TYPE jkff BODY brtv(0)
CARRY content@, delay brtv(0) ENDCARRY
ACTIVITY set (c: btm0, x: jkff) BODY
                                        "/ j = 1, k = 0 /"
IF ¬c △ 1 ∧ c THEN old@(x) <− 1 ENDIF
ENDset

ACTIVITY reset (c: btm0, x: jkff) BODY
                                        "/ j = 0, k = 1 /"
IF ¬c △ 1 ∧ c THEN old@(x) <− 0 ENDIF
ENDreset

ACTIVITY toggle (c: btm0, x: jkff) BODY
                                        "/ j = k = 1 /"
IF ¬c △ 1 ∧ c THEN old@(x) <− ¬ x ENDIF
ENDtoggle
ENDjkff
```

At this point, a one to one correspondence is still present between the carrier type jkff and the hardware component. A conditional transfer such as Figure 3.b could be embedded in a description by:

```
DECLARE a, b, r: jkff c, fa, fb: btm0 ENDDECLARE
c .= fa ¬ fb,
IF fa THEN IF a = 1 THEN set(c,r) ELSE reset(c,r)
ENDIF ENDIF,
IF fb THEN IF b = 1 THEN set(c,r) ELSE reset(c,r)
ENDIF ENDIF
```

More abstraction is obtained in the following ff type definition segment, where only one activity load, with an additional parameter, replaces set, reset, and toggle. The ff carrier type represents both, and allows more compact descriptions, with a loss in precision with respect to the actual component it is supposed to model:

```
TYPE ff BODY brtv(0)
CARRY content@, delay ENDCARRY
ACTIVITY load (c: btm0, d: bool, x: ff) BODY
IF ¬c △ 1 ∧ c THEN old@(x) <− d ENDIF
ENDload
ENDff
```

If TYPE ff were to be given as a primitive to the user of

a CONLAN language, a FORMAT statement would probably be attached to ACTIVITY load, in order to define a more convenient infix notation, such as

$$?c?\ x <= d$$

But such possibilities of abbreviation are not our point in this paper, and we shall retain the standard prefix notation.

If y has been declared ff, the statement:

IF ci THEN load (c, ¬y, y)

can be implemented by either of the circuits in Figure 4.

The module 6 counter of Figure 5, whether implemented with one type of flip-flop or another, is described by:

```
DECLARE r1, r2, r3: ff, c1: btm0 ENDDECLARE
load(c1, −r3, r1),
load(c1, r1, r2),
load(c1, r2, r3)
```

Using this abstract carrier type ff, the conditional transfer of Figure 3 can be simply written as:

```
DECLARE fa,fb,c: btm0, a[0:15],b[0:15],r[0:15]: ff
ENDDECLARE
c .= fa ∨ fb,
OVER i FROM 0 to 15 REPEAT
  IF fa THEN load (c, a[i], r[i]) ENDIF,
  IF fb THEN load (c, b[i], r[i]) ENDIF
ENDOVER
```

4.2 Duality between operation and DESCRIPTION

In a digital system, some components are viewed primarily as operations which transform input signals, seen as operands, to produce new values. Typical examples are combinatorial circuits, but sequential operative modules such as microprocessors are not excluded.

CONLAN supports two ways of representing such components. If the hardware designer is mainly interested in describing behavior, he will define and invoke FUNCTION and ACTIVITY segments. If structure must be displayed, he will define and instantiate DESCRIPTION segments. The choice is not dictated by the level of detail he wishes to display: all of these segments can be used for various levels of abstraction. Yet a fundamental difference exists as to how much hardware is implied in the designers text. If the same FUNCTION or ACTIVITY segment is invoked three times

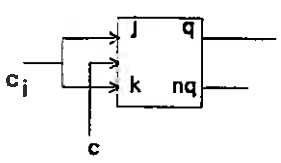

Figure 4(a)—y as JKFF.

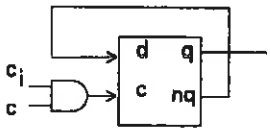

Figure 4(b)—y as dff.

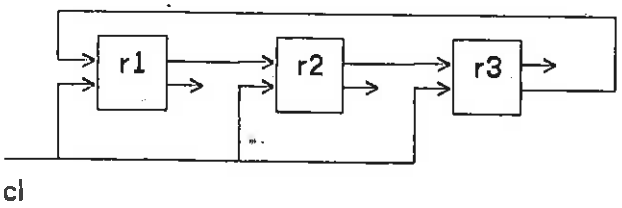

Figure 5—Module 6 counter.

in a model, there is no a priori indication of the number of hardware units: there could be one (with multiplexing or time sharing) or as many as three. On the contrary, if a DESCRIPTION segment is used, there are exactly as many distinct hardware components as are instantiated. And in fact, in a top-down approach, going from a model expressed in terms of operations to a model expressed in terms of instances of descriptions is precisely what is called synthesis.

This duality between operation and instance of description was suggested in the description of flip-flops dff1 and dff2 presented above. We shall illustrate it here with additional examples.

An adder as FUNCTION and as DESCRIPTION

Suppose we have defined the two types: (a) *btmlvector* to be a normalized (1:n) vector of Boolean terminals, default 0, and (b) *boolvector* to be a normalized vector of Booleans.

A generalized parallel adder, with interface carriers being Boolean terminals, can be written for all possible lengths of its x and y inputs, and z result, provided the size of x, y and z are equal:

```
DESCRIPTION adder(IN x,y: btmlvector, cin: btm0
  OUT z: btmlvector, cout: btm0) BODY
ASSERT size= size(y), size(z) = size(y) ENDASSERT
DECLARE c[0:size(x)]: btm0 ENDDECLARE

c[size(x)] .= cin,
OVER i FROM 1 TO size(x) REPEAT
  z[i] .= x[i] XOR y[i] XOR c[i],
  c[i−1] .= x[i] ∧ y[i] ∨ x[i] ∧ c[i] ∨ y[i] ∧ c[i]
ENDOVER,
cout .= c[0]
ENDadder
```

On the other hand, an adder may be viewed as a function on two Boolean vectors of same size, returning a Boolean vector with one more element, the leftmost bit being the carry.

```
FUNCTION add (x, y: boolvector, cin: bool):
boolvector BODY
ASSERT size(x) = size(y) ENDASSERT
DECLARE c[0:size(x)], z[1:size(x)]: btm0
ENDDECLARE
```

```
c[size(x)] .= cin,
OVER i FROM 1 TO size(x) REPEAT
  z[i] .= x[i] XOR y[i] XOR c[i],
  c[i−1] .= x[i] ∧ y[i] ∨ x[i] ∧ c[i] ∨ y[i] ∧ c[i]
ENDOVER
RETURN c[0] # z              "/ # is catenate operator /"
ENDadd
```

At the point of use of either segment, the size of their inputs must be known.

Suppose we want to perform addition on two bytes, and the model includes the following DECLARE statement:

```
DECLARE a[1:8], b[1:8], s[1:8], ca: btm0
ENDDECLARE
```

A number of possible statements will produce the same effect in terminal s and ca:

```
USE addition(a, b, 0, s, ca): adder ENDUSE
USE addition1(a[5:8], b[5:8], 0, s[5,8], addition 2.cin),
  addition2(a[1:4], b[1:4], addition 1.cout, s[1:4],ca):
  adder
ENDUSE
ca # s .= add(a, b, 0)
```

The first method clearly identifies one 8 bit adder; the second method shows two 4 bit adders; the third method does not call for a particular hardware implementation.

An alu as ACTIVITY and as DESCRIPTION

From IC data books, the logic diagrams and truth tables for each component can be modeled by a CONLAN description segment. If the hardware designer had access to a library of such descriptions, the logic of a CONLAN description could be tested, simulated, and verified by software.

For instance, a 4 bit MSI alu chip is represented below. Only logic functions are spelled out, for reasons of space. A one to one correspondence exists between the actual IC pins and the DESCRIPTION segment interface carriers, except for power supply pins which are not taken into account.

```
DESCRIPTION mc10181(IN a[0:3], b[0:3], s[0:3], cin,
m: btm0, OUT f[0:3], co, pg, gg: btm0) BODY
If m = 1 THEN                "/ logic functions /"
  CASE s IS
  0000: f .= ¬ a,
  0001: f .= ¬ a ∨ ¬ b,
  0010: f .= ¬ a ∨ b,
  0011: f .= 1111,
  0100: f .= ¬ a ∧ ¬ b,
  0101: f .= ¬ b,
  0110: f .= a XOR b,
  0111: f .= a ∨ − b,
  1000: f .= ¬ a ∧ b,
  1001: f .= a EQV b,
  1010: f .= b,
  1011: f .= a ∨ b,
  1100: f .= 0000
  1101: f .= a ∧ ¬ b,
  1110: f .= a ∧ b,
  1111: f .= a
  ENDCASE
ELSE ...                     "/ arithmetic functions /"
ENDIF
ENDmc10181
```

The same circuit can as well be written as an ACTIVITY segment. To do so, it is only sufficient to change the header, the body can be identical. However, for more clarity, input parameters can be typed bool, rather then Boolean terminal, to show that this ACTIVITY segment treats them as values.

```
ACTIVITY mc10181a(a[0:3], b[0:3], s[0:3], cin, m: bool,
    f[0:3], co, pg, gg: btm0) BODY
.....                        "/same as above/"
ENDmc10181a
```

Usage of one formulation or the other will be dictated, once again, by whether or not the actual number of ICs, and their exact interconnection, is to be displayed.

Constructing a 32 bit alu from 4 bit slices would imply instantiating 8 mc10181 DESCRIPTION segments. Whereas the two conditional invocations

```
IF c1 THEN mc181a (.....) ENDIF
IF c2 THEN mc181a (.....) ENDIF
```

might imply multiplexing, if c1 and c2 are mutually exclusive.

5. CONCLUSIONS

Base CONLAN is primarily a starting point, with well defined and semantically sound primitives, for language designers, to derive a coherent and comprehensive family of digital system description languages. As a result, writing hardware descriptions in Base CONLAN, although quite possible as we have seen, may look verbose. More time and effort will be needed before user oriented, less general but simpler, special purpose languages will be developed.

Nevertheless, all concepts pertinent to hardware modeling are already here, and will be common to all languages of the CONLAN family. The ability to partition a design into a network of interconnected modules is supported by the notion of DESCRIPTION segments. Instantiation of such segments, and intercommunication between instances, directly depicts the structure of a system, and the data and control paths which connect its components.

Declaring objects of a carrier TYPE may correspond to a physical hardware unit, but may also constitute an abstraction. In that sense, it is expected that hardware com-

pilers, fed with alternative libraries of DESCRIPTION segments, will generate models specialized for a given technology. CONLAN also provides FUNCTION and ACTIVITY segments, to describe the behavior of a system, often in more concise manner than when structure is fully displayed.

Space limitations prevented us from giving other than small examples. However, we hope to have made clear to the reader that, at each stage of the design of a real project, judicious selection of CONLAN segments leads to descriptions which are both adapted to the available amount of information and writing style.

6. ACKNOWLEDGMENTS

The authors are indebted to Bell Northern Research (Ottawa), Sperry Univac (Philadelphia), Office of Naval Research, Ballistic Missile Defense Advanced Technical Center (Huntsville), IRIA (Paris), Busdesministerium fur Forschung und Technologie (Bonn), Siemens (Munich), and Fujitsu (Tokyo) for their interest and support, Professor Yaohan Chu for his early contributions, and in particular to Professor Jack Lipovski for his help and unwavering confidence in the group.

7. REFERENCES

1. Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F. and Skelly, P., "CONLAN—A Formal Construction Method for Hardware Description Languages: Basic Principles," *Proceedings National Computer Conference*, Volume 49, Anaheim, California, 1980.
2. Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F. and Skelly, P., "CONLAN—A Formal Construction Method for Hardware Description Languages: Language Derivation," *Proceedings National Computer Conference*, Volume 49, Anaheim, California, 1980.