Technical Report

UCAM-CL-TR-72
ISSN 1476-2986

Number 72

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Logic programming and the specification of circuits

W.F. Clocksin

May 1985

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Logic Programming and the Specification of Circuits

W F Clocksin
Computer Laboratory
University of Cambridge
Corn Exchange Street, Cambridge CB2 3QG

Index terms: Prolog, logic programming, hardware description

Abstract

Logic programming (see Kowalski, 1979) can be used for specification and automatic reasoning about electrical circuits. Although Propositional Logic has long been used for describing the truth functions of combinational circuits, the more powerful Predicate Calculus on which logic programming is based has seen relatively little use in design automation. Previous researchers have introduced a number of techniques similar to logic programming, but many of the useful consequences of the logic programming methodology have not been exploited. This paper first reviews and compares three methods for representing circuits, which will be called here the *functional* method, the *extensional* method, and the *definitional* method. The latter method, which conveniently admits arbitrary sequential circuits, is then treated in detail. Some useful consequences of using this method for writing directly executable specifications of circuits are described. These include the use of quantified variables, verification of hypothetical states, and sequential simulation.

1. Review of Terminology

A *circuit* is composed of a set of *modules* and connections between modules. With each module is associated a set of *ports* between which the connections are defined, and which may be used for input and output. Modules can be composed hierarchically, in which modules are specified in terms of other modules. At the bottom of the hierarchy are primitive modules, the identity of which depends on the technology being used. For example, a VLSI designer might take transistors to be primitive; other digital designers might take logic gates or even individual packages to be primitive.

Circuits can be specified, simulated, and reasoned about by means of logic programming. Circuits are represented using the terms and formulae of logic, and properties of circuits can be derived by straightforward logical deductions from the given specifications. This paper will be restricted to the use of logic to represent the *structure* of circuits.

We shall use the Edinburgh Prolog syntax for terms and formulae. An introductory account is given in Clocksin and Mellish (1981). To review, terms are constant symbols, variables, and compound terms. Constant symbols are either integers, or written with an initial lower-case letter, or a string of non-alphanumeric symbols.

Variables are written with an initial upper-case letter. An n-ary compound term is written as $f(t_1, t_2, \ldots, t_n)$, where f is a constant symbol called the *functor*, and each t_i, $1 \leq i \leq n$ is a term called an *argument*. A 0-ary compound term having functor f is considered the constant symbol f. A compound term is called *ground* if all of its arguments are either constant symbols or (recursively) ground terms. It is also permitted to nominate functors as prefix, infix, or postfix operators, so they can be written for example as, $t_1 \quad f \quad t_2$.

The *list*, a commonly used compound term, is constructed recursively in the way usual to functional languages, and a syntactic device is employed to avoid the awkwardness of representing the list of length N as compound terms nested N deep. The list of length 0 is written as `[]`. The list of length N is written as $[t_1, t_2, \ldots, t_N]$, where term t_1 is called the *head* of the list, and terms $t_2, \ldots, t_N$ are the elements of a list of length $N-1$ called the *tail*. The list having head h and tail ℓ can be written $[h \mid \ell]$.

Formulae are written as Horn clauses, which represent a subset of the formulae of Predicate Calculus. Let P and some Q_i $(1 \leq i \leq q)$ stand for compound terms. Let x_j $(1 \leq j \leq m)$ stand for variables appearing in P and possibly in the Q_i. Let y_k $(1 \leq k \leq n)$ stand for variables appearing in the Q_i and not appearing in P. Then we write the Horn clause

$$P \quad :- \quad Q_1, Q_2, \ldots, Q_q.$$

which is equivalent to the Predicate Calculus formula

$$\forall x_1, \ldots, x_m (\exists y_1, \ldots, y_n \quad Q_1 \wedge Q_2 \wedge \ldots \wedge Q_q) \supset P.$$

P is called the *head*, and the Q_i are called the *body*. If $n = 0$ then there is no body, and the resulting clause is called a *unit clause*.

Clauses are used as a program to which queries (or *goals*) can be posed. The strategy used by Prolog for the execution of goals is briefly summarised as the following recursive definition. Given an ordered collection of clauses called the *database* and a conjunction of goals $G_1, \ldots, G_n$, we can *execute* the goals by means of a simple backtracking strategy together with a pattern-matching algorithm known as *unification*. To execute goal G_i, scan the database from the beginning to find *the first* clause whose head matches the goal according to the unification algorithm. If such a clause is found, the position of the matching clause in the database is marked, and each goal in the body of the clause is executed. If all goals in the body can be executed (or if there is no body), then goal G_i is said to succeed, and goal G_{i+1} is executed. If a matching clause is not found, G_i is said to fail, and backtracking occurs: any variables in G_{i-1} instantiated by its previous success are now unbound, and goal G_{i-1} is executed by rescanning the database from the position after the placemarker associated with G_{i-1}. The result of this simple strategy is a depth-first left-to-right exhaustive search of the database that attempts to satisfy the conjunction of goals.

Two terms are matched according to the unification algorithm if there is a most general substitution for the variables in the term such that the terms may be made equal. In logic programming, unification is a general-purpose feature used for passing input and output parameters, and for incremental construction of data structures. Unification is used below for propagating values through a circuit.

2. Summary of Functional and Extensional Methods

The first two methods are summarised here only for the purpose of comparison. They are shown to have sufficient disadvantages that we do not consider them further.

2.1. The Functional Method

The functional method can easily represent combinational acyclic circuits in which a single output signal is the *function* of several input signals. Modules are represented as ground compound terms, where a particular input port is associated with a particular argument of the term. Constant symbols are used to denote primitive modules which have no input (for example named input signals and power connections). The function symbol together with its arguments names the output of the module. The only connection relationship between modules is purely functional: the syntactic form of a given term determines the connections between modules. Thus, this technique does not make use of formulae. Consider the example in Figure 1(a), which shows a simple combinational circuit and the ground term representing it.

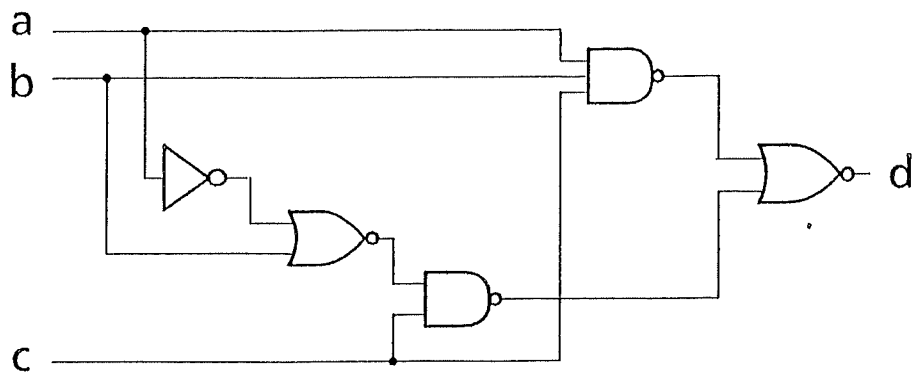
Insert Figure 1 Near Here

In this example, input signals are named by the constant symbols `a`, `b`, and `c`. Modules are named by the 2-ary function symbols `nand` and `nor` and the 1-ary function symbol `not`.

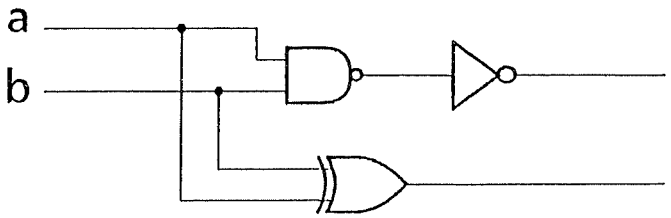
This representation permits processing of the circuit by using recursive descent to transform a given term into another (Clocksin and Mellish, 1981), and it is used in a number of elementary treatments (Sammut and Sammut, 1983; Wos, *et al.*, 1984). There are two main disadvantages to this technique. First, only acyclic circuits can be described by ground terms, and this precludes the specification of realistic sequential circuits. Second, a separate expression must be used to represent each output of a circuit, as shown in Figure 1(b). By introducing equations and metafunctional devices such as the μ-operator of Sheeran (1983), it is possible to represent sequential circuits with cyclic connections and multiple outputs, whilst remaining within a functional programming (Henderson, 1980) context. Such a treatment brings this method closer to the definitional method described later, however, the usual use of the functional technique is restricted to ground terms, and does not include formulae. The two restrictions militate aginst using the functional technique for all but the most trivial of circuits found in practice.

2.2 The Extensional Method

The extensional method represents each module and connection as a unit clause in which constants are used to indicate the connections between modules. In the example shown in Figure 2, the 3-ary predicate `module` describes the type of a module, a list of its input port names, and a list of its output port names. The binary predicate `connect` describes connections between ports, where the named port of a given module is represented as a ground 1-ary compound term consisting of a function symbol naming the module type together with a argument naming the port.



(a) nor(nand(nor(not(a),b),c),nand3(a,b,not(c)))



(b) not(nand(a,b)
xor(a,b)

Figure 1

```
module(xor,[a,b],[c]).
module(not,[a],[b]).
module(csrff,[s,c,r],[q]).

connect(xor(a),z).
connect(xor(b),x).
connect(xor(c),csrff(s)).
connect(xor(c),not(a)).
connect(not(b),csrff(r)).
connect(clock,csrff(c)).
connect(csrff(q),z).
```

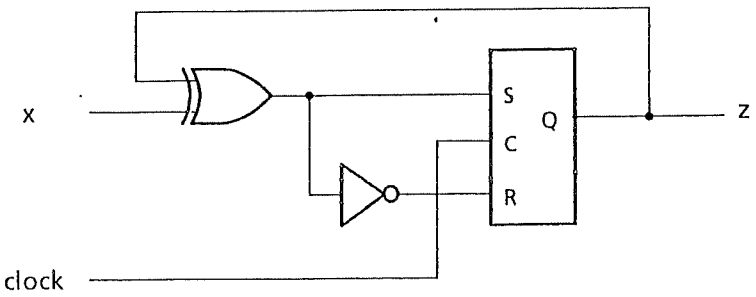


Figure 2

Insert Figure 2 Near Here

The circuit is thus described as the *extension* of the `module` and `connect` relations. Variations on this method are used in a number of systems (Barrow, 1984; Horstmann, 1983; Kollaritsch and Weste, 1984). It is usual to consider the `module` and `connect` relations as templates, and to augment their arguments with variables that stand for any instance of the module and connection. Additional arguments can be used to represent state variables, type information, and so forth.

The extensional method can accommodate arbitrary types of circuits such as multiple output cyclic circuits. However, the disadvantages of this method stem from the property that modules are not represented by a single term. The modules and connections of a circuit are represented extensionally, with no syntactic relationship between them. The result is difficulty in efficiently carrying out certain operations such as circuit transformations, where in this case it is necessary to make awkward modifications to the database. On the other hand, single terms used by the other methods are easily rewritten either by tree- or graph-rewriting. Also, there is less opportunity for modularity, as unit clauses contain no existentially quantified variables.

3. The Definitional Method

This method, which we shall treat in detail, represents a circuit as a formula in a subset of first-order logic. Modules having n ports are represented as n-ary predicate symbols. The modules in a given circuit are composed with a binary connective (here we use an infixed comma). Ports within a given circuit that share a common connection are represented by a like-named variable. It is convenient to write a module as a Horn clause in which the head of the clause represents the module to be defined, and the body of the clause is a composition of modules. The ':-' operator is re-interpreted to mean 'is defined by'. The order of modules in the body of the clause is not important. The examples from above are shown as follows:

```
combo(A,B,C,D) :-
        not(A,T1), nor(T1,B,T2), nand(C,T2,T3),
        not(C,T4), nand3(T4,A,B,T5), nor(T3,T5,D).

half_add(A,B,S,C) :- xor(A,B,S), nand(A,B,T1), not(T1,C).

seq_par(X,Clock,Z) :- xor(X,Z,T1), not(T1,T2), csrff(T1,Clock,T2,Z).
```

A specification of each primitive module is required. For example, `not(`a`,`b`)` specifies an inverter, with input port a and output port b; `nand3(`a`,`b`,`c`,`d`)` specifies a three-input nand gate for inputs a, b, c, and output d; `csrff(`a`,`b`,`c`,`d`)` specifies a clocked set-reset flip-flop with S input a, clock b, R input c, and output d.

Variations on this method are used by Batten (1983), Fujita (1983), Gordon (1983), Moszkowski (1983), Svanæs and Aas (1984), and Clocksin (1984). The method has numerous advantages, especially when the circuits are represented as here by Prolog clauses. First, the module name is explicitly part of the specification. This permits easy modular decomposition. For example, consider a three-bit subtractor, which consists of a half-subtractor and a pair of full-subtractors:

```
half_sub(I1,I2,D,BO) :- xor(I1,I2,D), not(I1,T1), and(I2,T1,BO).
```

```
full_sub(A,B,BI,D,BO) :-
        xor(A,B,T1), xor(T1,BI,D), not(T1,T2), not(A,T3),
        nand(T2,BI,T4), nand(T3,B,T5), nand(T4,T5,BO).

three_sub(A0,A1,A2,B0,B1,B2,D0,D1,D2,T2) :-
        half_sub(A0,B0,D0,T0),
        full_sub(A1,B1,T0,D1,T1),
        full_sub(A2,B2,T1,D2,T2).
```

Internal connections, which are named by variables which do not appear in the head of the clause, are effectively "hidden". Such lexical scoping is a good engineering practice which is not provided by the extension method described above. By inspection of the Predicate Calculus equivalent formula, hidden variables are existentially quantified.

The second advantage is that specifications can be directly executed by a Prolog system. We shall return to this issue below. A third advantage is the inherent bidirectionality obtained by the use of the logical variable. Bidirectionality is an important behaviour of some components such as pass transistors, and has not been explored in previous treatments. Input and output rôles of ports can be constrained by extra clauses if necessary, however, the method shown here permits a representation of bidirectionality if required. Furthermore, an uninstantiated variable is a natural representation of the "floating" or high-impedance state. These issues will be explored in the examples that follow.

4. Direct Execution of Specifications

Consider first the direct execution of the `half_sub` module. Definitions of the `xor`, `not`, and `and` primitives are given by the following unit clauses, which resemble the standard truth tables for these relations:

```
xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).

not(0,1).
not(1,0).

and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).
```

The constants `1` and `0` stand for logic high and logic low, respectively. Now the Prolog goals for executing the half subtractor given above with all combinations of inputs to obtain difference `D` and borrow `B` are as follows, with the computer's output given in italics:

```
?- half_sub(0,0,D,B).
```
D = 0, B = 0

```
?- half_sub(0,1,D,B).
```
D = 1, B = 1

```
?- half_sub(1,0,D,B).
```
D = 1, B = 0

```
?- half_sub(1,1,D,B).
```
D = 1, B = 0

The four goals provide a verification by exhaustive simulation. This is an unrealistic verification method in practice, and even generation of a smaller incomplete set of input test patterns can be unwieldy. One alternative, still confined to simulation, is to use a method of "hypothetical states". Test patterns corresponding to inputs or outputs are queried, and the Prolog system computes the possible conditions (by depth-first backtracking search of the circuit) under which the test pattern can be obtained. The clauses specifying a module are seen as constraints on the possible behaviour of the module. For example, this query asks for the possible input and difference outputs for which the borrow output is 1. The only solution, computed by the Prolog system, is given in italics:

```
?- half_sub(I,J,D,1).
```
I = 0, J = 1, D = 1

This "backwards" analysis of the circuit is possible because of the definition of circuits in terms of relations. Another example, which queries the conditions under which the first input is 1, and the second input is the same as the difference output, is as follows:

```
?- half_sub(1,J,J,B).
```
no

The Prolog system correctly reports that no such state is possible for a correctly behaving `half_sub` module.

Next, consider the representation and simulation of logic functions using complementary transistors. We will make the simplifying but reasonable assumption that the difference between gate-source capacitance and gate-drain capacitance is negligible. Where G stands for gate, S stands for source, and D stands for drain, we represent a p-type transistor as `ptrans(`G`,`S`,`D`)`, and an n-type transistor as `ntrans(`G`,`S`,`D`)`. Definitions of the transistors, a complementary inverter, and a complementary nand gate are as follows:

```
ntrans(1,Y,Y).
ntrans(G,X,Y).

ptrans(0,Y,Y).
ptrans(G,X,Y).

invert(In,Out) :- ntrans(In,Out,0), ptrans(In,Out,1).

nand(I1,I2,Out) :-
        ptrans(I1,Out,1), ptrans(I2,Out,1),
        ntrans(I1,Out,W), ntrans(I2,W,0).
```

If these specifications are directly executed as a Prolog program, the logic gates behave correctly whether or not the inputs are instantiated. However, this transistor model also permits transistors to be driven "backwards": if the source and drain are equal, then a floating n-type gate will be forced high; a floating p-type gate will be forced low. Real transistors do not behave this way, and it is possible to modify the model accordingly. The simplicity of the model shown here is useful because it is not necessary to order the simulation of modules when the clause is directly executed. Furthermore, the "backwards" behaviour is needed to implement the method of hypothetical states. For example, the model will tell us the valid conditions under which the source and drain are equal. For an n-type transistor, there are two possibilities: (1) the gate is 1 (by the first clause of the definition of `ntrans`), or (2) the gate is floating (by the second clause) purely coincidentally.

Our next example is a full adder constructed from complementary transistors. The previous specifications of p- and n-transistors are used. An interesting feature of the adder is the use of two transistors which must conduct bidirectionally depending on the input state. Direct execution of the `adder` module given in Figure 3 correctly simulates the bidirectional behaviour. It is easy to enumerate all 2^3 input states of this module, so this module can be verified by exhaustive simulation. Also, the method of hypothetical states can be used for "spot checking" of particular states of interest.

Insert Figure 3 Near Here

5. Sequential Simulation by Direct Execution

We have seen how the definitional method can be used for direct execution of specifications. We now demonstrate how direct execution can be used for simulating sequential circuits. We begin by specifying the D-type flip-flop. The term `dff(`D`,`C`,`Q`,`Q'`)` represents the D-type flip-flop with input D, clock C, output Q, and next state Q'. We have left out the $\overline{Q}$ output available on some devices for convenience. The two unit clauses specifying `dff` are:

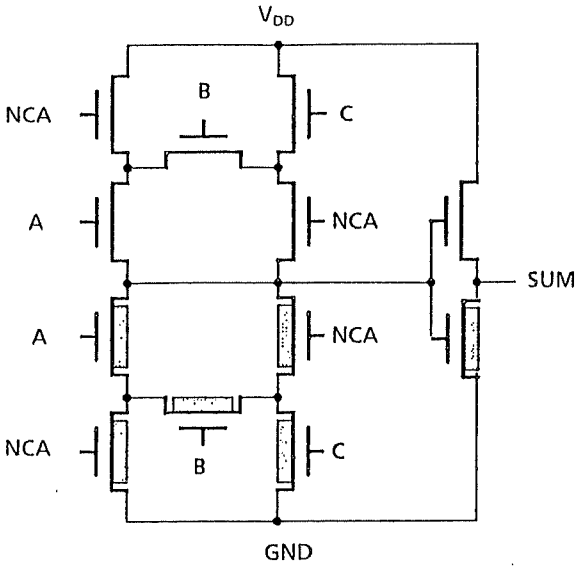
```
dff(D,0,Q,Q).
dff(D,1,Q,D).
```

The first clause specifies behaviour on a falling clock: the next state is the same as the current state. The second clause specifies behaviour on a rising clock: the next state is the same as the D input.

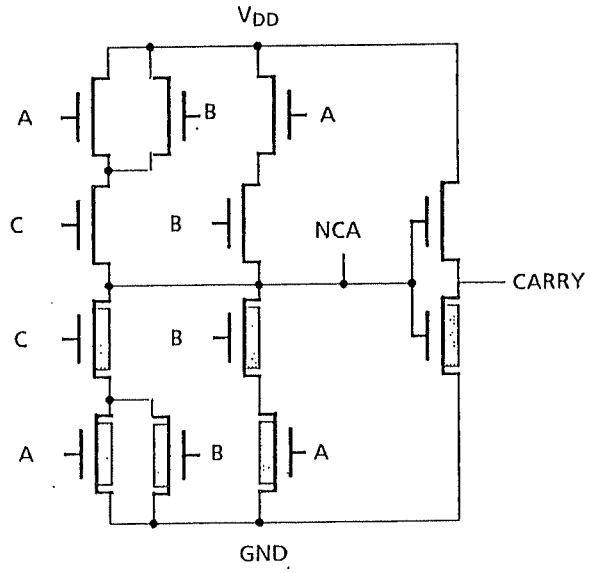
The simplest sequential circuit, a divide-by-2 pulse divider, can be specified as follows (with the `not` module defined as above):

```
div(C,Q,Z) :- not(Q,D), dff(D,C,Q,Z).
```

Term `div(`C`,`Q`,`Q'`)` has a clock input C, a current state Q, and next state Q'. We can insert this module into a "test circuit" by writing a Prolog procedure that recurs over an input list of clock pulses. The current state can be jammed with an initial state (say 0), and the output states are collected into an output list. The Prolog goal `divide(P,I,Q)`, when given a pulse list `P` and initial state `I`, will construct an output pulse list `Q`. The definition of `divide` is:




```
sumpart(A,B,C,NCA,SUM) :-
    ptrans(NCA,T1,1),
    ptrans(C,1,T5),
    ptrans(B,T1,T5),
    ptrans(A,T1,T2),
    ptrans(NCA,T5,T2),
    ptrans(T2,1,SUM),
    ntrans(A,T2,T3),
    ntrans(NCA,T2,T6),
    ntrans(T2,SUM,0),
    ntrans(B,T3,T6),
    ntrans(NCA,T3,0),
    ntrans(C,T6,0).
```




```
carrypart(A,B,C,NCA,CARRY) :-
    ptrans(A,T1,1),
    ptrans(B,T1,1),
    ptrans(A,T2,1),
    ptrans(C,T1,NCA),
    ptrans(B,T2,NCA),
    ptrans(NCA,1,CARRY),
    ntrans(C,NCA,T3),
    ntrans(B,NCA,T4),
    ntrans(NCA,CARRY,0),
    ntrans(A,T3,0),
    ntrans(B,T3,0),
    ntrans(A,T4,0).
```

Figure 3

```
divide([],S,[]).
divide([P|Ps],IS,[Q|Qs]) :- div(P,IS,Q), divide(Cs,Q,Qs).
```

Sample executions of this test circuit follow (computer's response in italics):

```
?- divide([1,1,1,1,1,1],0,Q).
```
Q = [1,0,1,0,1,0]

```
?- divide([0,1,0,0,1,1,0,0],0,Q).
```
Q = [0,1,1,1,0,1,1,1]

The next example is a sequential parity checker. On each clock pulse, the output provides an odd-parity check on however many data bits have been received by the serial input since the initial state of the circuit. The sequential parity checker is specified by the term par(C,D,Q,Q') for clock input C, serial data input D, parity output Q, and next state Q', by the following definition (making use of xor as defined above):

```
par(C,D,Q,N) :- xor(D,Q,T), dff(T,C,Q,N).
```

We use the same technique of recurring over a list of clock pulses to form a test circuit checker(C,S,I,Q) for clock pulse list C, serial input list S, initial state I, and serial parity output list S:

```
checker([],S,I,[]).
checker([C|Cs],[S|Ss],IS,[NS|L]) :-
        par(C,S,IS,NS),
        checker(Cs,Ss,NS,L).
```

When the initial state is jammed to 0, an example goal together with the computer's reply is as follows:

```
?- checker([1,1,1,1,1,1],[1,0,0,1,1,0],0,Q).
```
Q = [1,1,1,0,1,1]

Note that, for the given input, odd parity is counted for the first three and last two clock pulses.

A final example is a three-bit synchronous Gray code counter depicted in Figure 4.

Insert Figure 4 Near Here

Here the two combinational subcircuits are specified as separate clauses for procedures neta and netb. The and and dff modules are defined as above, and the or module is easily defined in a similar way. A circuit state vector is represented by the compound term s(Q_a,Q_b,Q_c), in which the state for each flip-flop is stored.

```
neta(A,B,C,Q1,Q2) :-
        and(A,C,T1),
        not(C,NC),
        and(B,NC,T2),
        not(A,NA),
        and(NA,C,T3),
        or(T1,T2,Q1),
        or(T2,T3,Q2).
```

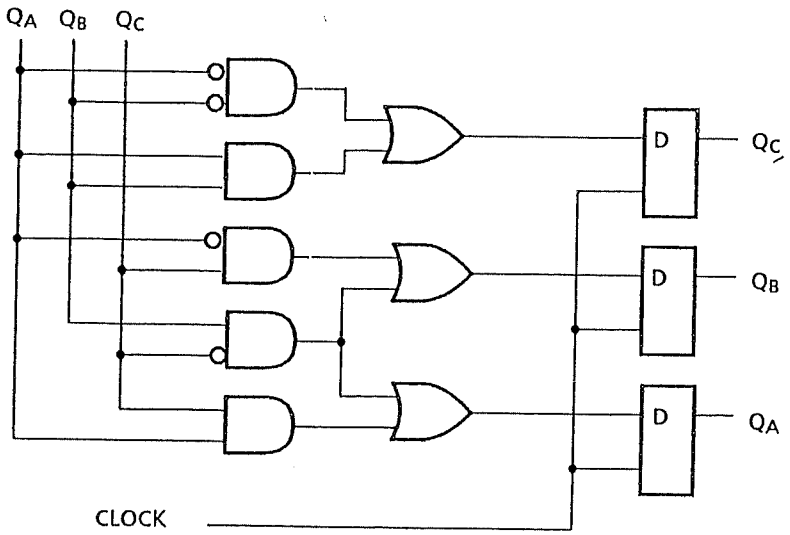


Figure 4

```
netb(A,B,Q) :-
        and(A,B,T1),
        not(A,NA), not(B,NB), and(NA,NB,T2),
        or(T1,T2,Q).

gcc(C,s(Qa,Qb,Qc),s(Za,Zb,Zc)) :-
        neta(Qa,Qb,Qc,D1,D2),
        netb(Qa,Qb,D3),
        dff(C,D1,Qa,Za),
        dff(C,D2,Qb,Zb),
        dff(C,D3,Qc,Zc).
```

A test circuit is constructed as before:

```
testgcc([],S,[]).
testgcc([C|Cs],I,[N|Ns]) :-
        gcc(C,I,N),
        testgcc(Cs,N,Ns).
```

A query to test the circuit for nine pulses together with the result is:

```
?- testgc([1,1,1,1,1,1,1,1,1],s(0,0,0),Q).
```

Q = [s(0,0,1),s(0,1,1),s(0,1,0),s(1,1,0),
s(1,1,1),s(1,0,1),s(1,0,0),s(0,0,0),s(0,0,1)]

It can be observed that the successive states represent an incrementing Gray code.

6. Conclusions

Although logic programming has been used previously in the area of design automation, many of its properties have not been exploited. We have described some consequences of using logic programming for constructing executable specifications of digital circuits, with examples written in Prolog. Of particular interest is the power of unification with logical variables. Circuit structure can be specified by representing a connection between ports as a named variable. Values are propagated through a circuit as goal execution proceeds, and backtracking occurs whenever module constraints cannot be satisfied. An uninstantiated variable can represent a high-impedance state, and bidirectionality and hypothetical states can be simulated. The main drawback is the over general relational model of devices. More constrained models can be specified, but at the sacrifice of order-independent execution of goals.

7. References

Batten, J W, 1983. Prolog: Its potential for hardware description and verification. Department of Computation, UMIST.

Barrow, H G, 1984. VERIFY: A program for proving correctness of digital hardware designs. *Artificial Intelligence* **24**, 437-491.

Clocksin, W F, 1984. A greedy gate assigner. Computer Laboratory, University of Cambridge.

Clocksin, W F, and Mellish, C S, 1981. *Programming in Prolog*, Springer-Verlag.

Fujita, M, 1983. Logic design assistance with temporal logic. Draft Thesis, University of Tokyo.

Gordon, M J C, 1983. LCF-LSM: A system for specifying and verifying hardware. *Technical Report 41*, Computer Laboratory, Univerisity of Cambridge.

Henderson, P, 1980. *Functional Programming*, Prentice-Hall.

Horstmann, P W, 1983. Expert systems and logic programming for CAD. *VLSI Design*, November 1983.

Kollaritsch, P W, and Weste, N H E, 1984. A rule-based symbolic layout expert. *VLSI Design*, August, 62-66.

Kowalski, R A, 1979. *Logic for Problem Solving*, North-Holland.

Moszkowski, B C, 1983. A temporal logic for multi-level reasoning about hardware. *Proc IFIP 6th Int Conf Comp Hard Desc Lang and App.*

Sammut, R A, and Sammut, C A, 1983. Prolog: A tutorial introduction. *The Australian Computer Journal* **15**(2), 42-51.

Sheeran, M, 1984. μFP, An Algebraic VLSI Design Language. D.Phil. Thesis, University of Oxford.

Svanæs, D, and Aas, E J, 1984. Test generation through logic programming. *Integration* **2**, 49-67.

Wos, L, Overbeek, R, Lusk, E, and Boyle, J, 1984. *Automated Reasoning*, Prentice Hall.