Technical Report

UCAM-CL-TR-66
ISSN 1476-2986

Number 66

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A formal hardware verification methodology and its application to a network interface chip

M.J.C. Gordon, J. Herbert

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

Abstract

We describe how the functional correctness of a circuit design can be verified by machine checked formal proof. The proof system used is LCF_LSM [1], a version of Milner's LCF [2] with a different logical calculus called LSM. We give a tutorial introduction to LSM in the paper.

Our main example is the ECL chip of the Cambridge Fast Ring (CFR) [3]. Although the ECL chip is quite simple (about 360 gates) it is nevertheless real. Minor errors were discovered as we performed the formal proof, but when the corrected design was eventually fabricated it was functionally correct first time. The main steps in the verification were:

- Writing a high-level behavioural specification in the LSM notation.
- Translating the circuit design from its Modula-2 representation in the Cambridge Design Automation System [4] to LSM.
- Using the LCF_LSM theorem-proving system to mechanically generate a proof that the behaviour determined by the design is equivalent to the specified behaviour.

In order to accomplish the second of these steps, an interface between the Cambridge Design Automation System and the LCF_LSM system was constructed.

CONTENTS

1. Introduction

The main purpose of this paper is to describe how we went about verifying the functional correctness of a real circuit design. The actual proof was done entirely by John Herbert as part of his PhD research on formal methods for design automation.

Before describing the formal verification process we give an overview of our main example, the ECL chip, followed by an introduction to the specification language LSM and its rules of inference.

2. The Cambridge Fast Ring

The Cambridge Fast Ring (CFR) is a system for interconnecting digital devices. It provides a closed loop communication path on which packets circulate and to which devices can be attached. The CFR is designed to operate at around 100MHz and is implemented using several chips which can be configured in a number of ways. The main components are a high speed ECL chip, a CMOS chip and a 64k DRAM. The circuit we verified is the ECL chip.

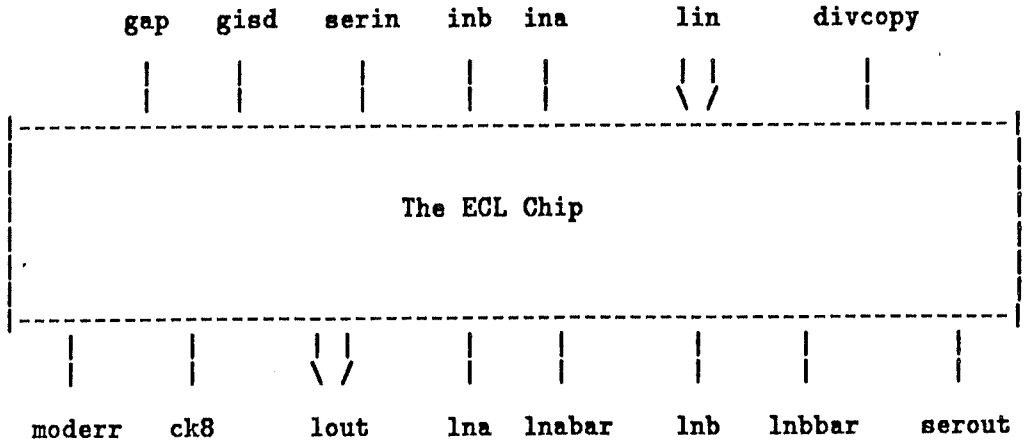
2.1. The ECL Chip

The ECL chip provides the interface between the ring and the slower access logic in the CMOS chip. It can perform modulation and demodulation if the ring links use the Cambridge modulation system (see below), or can interface to direct data inputs and outputs (*e.g.* for transmission systems using fibre optics). It transforms serial data packets on the ring to 8 bit parallel packets for the slower logic and does the reverse transformation for 8 bit wide packets from the slower logic. A Cambridge Ring contains a fixed number of slots plus a gap which consists of zeros. The gap is at least 6 bytes long and it may have an odd number of bits. The chip must detect the end of the gap and signal this to the slower logic. A clock at the byte frequency is produced by the chip. At the end of a gap this clock must be reset.

The Cambridge modulation system is based on delay modulation. In the basic scheme, data can be transmitted using two lines and boolean values (denoted by T and F) are encoded by the changes on the lines at successive clock ticks. The value T corresponds to changes on both lines. A change on one line corresponds to F. Neither line changing is an error (a modulation error). The changes on the

lines can be balanced so that each line is guaranteed to change at least once every two clock periods. The clock can be recovered from the modulated data.

The ECL chip has the following pins:



The functions of these pins are:

Inputs

`gap` is asserted when the ECL chip is required to look for the end of the gap between packets on the ring.

`gisd` (gate in serial data) selects between the modulated data inputs (`ina` and `inb`) and the serial data input (`serin`).

`serin` is a serial data input as might be used, for example, with a fibre optic link.

`ina, inb` are the inputs for data encoded in the Cambridge modulation system. Differential receivers are used to derive the ECL inputs from the ring signals.

`lin` is an 8-bit wide bus from the CMOS chip which carries the bytes to be transmitted from the station.

`divcopy` when asserted, the chip is in its normal operating mode with data received from the CMOS chip being output to the ring. When `divcopy` is not asserted the input data from the ring is copied to the ring outputs.

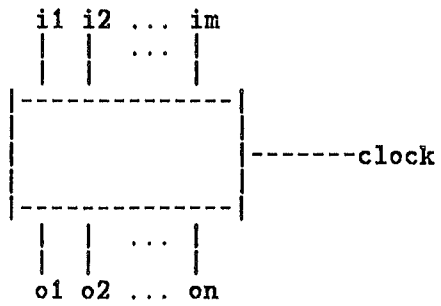
Outputs:

`moderr`	is asserted if a modulation error has been detected in the modulated data received from the ring.
`ck8`	is a clock signal to the CMOS logic at the byte frequency (1/8 of the main clock frequency) with a stretched period when the gap between packets is not an integral number of bytes.
`lout`	is an 8-bit wide data bus to the CMOS logic which presents to the slower logic the bytes received from the ring.
`lna, lnabar`	
`lnb, lnbbar`	are the modulated data output lines which are interfaced to the ring via drivers.
`serout`	is the serial data output line.

The ECL chip was designed in the the Computer Laboratory by Andy Hopper and has a complexity equivalent to about 360 gates.

3. Introduction to LSM

The kind of device that can be specified in LSM has the general form:



Such a device has input lines `i1` ,..., `im`, output lines `o1`, ... , `on` and possibly some internal registers `x1`, ..., `xp`. It is assumed to behave like a sequential machine as follows:

- At each moment in time the values on the output lines are a function of the values in the registers (the state) and the values on the input lines.
- The values in the registers stay constant until a clock pulse is received on the clock line. (Exactly how a clock pulse is realised physically is left

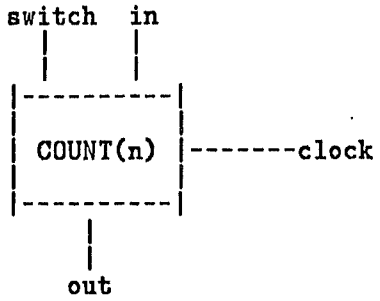
unspecified - it could, for example, actually be two voltage level changes, or just a single pulse).

3.1. Behavioural Specification

To formally specify the behaviour of such a machine in LSM one must:

- Specify the value on each output line in terms of the values of the state registers and the values on the input lines.
- Specify how the state changes when the device is clocked.

As an example consider a counter:



Here the input lines are `switch` and `in`, the only output line is `out` and the only state variable is `n`. The name of the device is `COUNT`; we write `COUNT(n)` (or `COUNT n`) to show that the behaviour (to be described) depends on `n`. Suppose the behaviour of `COUNT` is informally specified by:

- The value on the output line `out` is always the value in the register represented by the state variable `n`. We can express this with the output equation:

  ```
  out = n
  ```

- When the counter is clocked, if `F` is being input on line `switch`, then the value of the state variable `n` becomes `n+1`, otherwise it becomes the value input on line `in`. We express this by saying that when the clock 'ticks' the counter's behaviour changes from `COUNT(n)` to `COUNT(switch->in|n+1)`, where the expression `switch->in|n+1` is a conditional and has value `in` if `switch` is `T` and value `n+1` if `switch` is `F`.

In LSM the behaviour of the counter is specified by:

```
COUNT(n) == dev{switch,in,out}.{out=n};COUNT(switch->in|n+1)
```

This definition has the form `e1 == e2` where `e1` is the expression `COUNT(n)` and `e2` is a kind of expression called a *behaviour specification* which we describe in detail

below (readers unfamiliar with LCF should think of == as equivalent to =). Notice that the clocking is implicit in our notation (*i.e.* we don't explicitly mention the clock line). From now on we will not draw clock lines in diagrams, though they will still be needed in actual hardware implementations. Our model of behaviour abstracts away from the physical details of how state-changes are effected, and treats devices as abstract sequential machines. In the ECL chip certain state changes are clocked by a separate clock which is derived from the main clock. We discuss how this is handled in LSM later.

A typical behaviour expression has the form:

```
dev{x1,...,xm}.{l1=e1,...,ln=en};e
```

This denotes a sequential machine whose input and output lines are `x1`, ... ,`xm`. If `li` is not listed among `x1`, ..., `xm` then it is an internal (or virtual) line, such lines will be motivated later in the context of the ECL chip specification. The expression `ei` gives the value output on line `li`. The new state after clocking is specified by the expression `e`. Normally `e` will have the form `D(e1',...,er')` where `D` is a device name (*e.g.* `COUNT`) and `e1'`, ... ,`er'` are expressions giving the new values of the state variables of `D`. If `li` occurs in `e` or in one of the `ei` then its value there is determined by the equations `{l1=t1,...,ln=tn}`.

An example of a behaviour specification is:

```
dev{i,o}.{o=n};REG(i)
```

This specifies a device that outputs the value of variable `n` on line `o` and then, when clocked, becomes a device with behaviour `REG(i)` - *i.e.* becomes the device `REG` in a state holding the value input in line `i`. Suppose we specify `REG` by:

```
REG(n) == dev{i,o}.{o=n};REG(i)
```

then this defines `REG` to be a device which always outputs its state, and stores the current value input, thus it delays by one clock cycle. Definitions of this form - *i.e.* of the form:

```
D(a1,...,ap) == dev{x1,...,xm}.{l1=e1,...,ln=en};D(e1',...,ep')
```

are called *behaviour equations*. They are used to directly specify sequential machines. We will shortly show how to give a structural specification of a machine in LSM also.

The counter informally described above can be specified by the behaviour equation:

```
COUNT(n) == dev{switch,in,out}.{out=n};COUNT(switch->in|n+1)
```

The right hand side of this equation is the behaviour specification:

```
dev{switch,in,out}.{out=n};COUNT(switch->in|n+1)
```

Note that this expression has no internal lines. An example of a behaviour specification with internal lines is:

```
dev{switch,in,out}.
   {l1=(switch->in|l2),out=n,l2=out+1};
   COUNT_IMP(l1)
```

Here l1, and l2 are internal lines.

If the device COUNT_IMP has a behaviour satisfying:

```
COUNT_IMP (n) ==  dev{switch,in,out}.
                     {l1=(switch->in|l2),out=n,l2=out+1};
                     COUNT_IMP(l1)
```

then the rule of unfolding (described later) enables us to 'solve' the equations for the lines, and derive:

```
COUNT_IMP(n) ==  dev{switch,in,out}.
                    {l1=(switch->in|n+1),out=n,l2=n+1};
                    COUNT_IMP(switch->in|n+1)
```

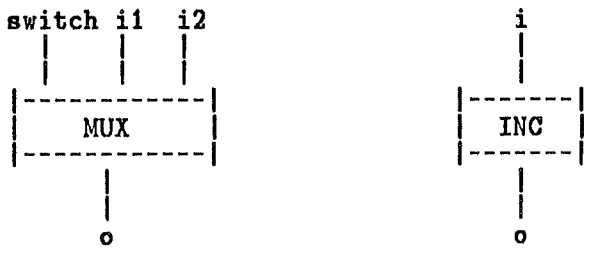
Note that in this formula l1 and l2 are no longer used anywhere. The pruning rule (described later) will enable the equations for these variables to be removed to get:

```
COUNT_IMP(n) == dev{switch,in,out}.
                   {out=n};
                   COUNT_IMP(switch->in|n+1)
```

Note that this equation for COUNT_IMP is similar to the equation specifying COUNT. Because behaviour equations have unique solutions we can infer from this that:

```
COUNT(n) == COUNT_IMP(n)
```

If the state of a device remains constant over time it is called *combinational*. Here are two examples:



The value on the output line o of the multiplexor MUX equals the value on line i1 if the value on line switch is T, otherwise it is the value on line i2. Thus the value on the output line is given by the output equation: o=(switch->i1|i2). The behaviour of MUX is thus specified by the behaviour equation:

```
MUX == dev{switch,i1,i2}.{o=(switch->i1|i2)};MUX
```

Note the lack of state variables. The behaviour of INC is specified by the behaviour equation:

```
INC == dev{i,o}.{o=i+1};INC
```

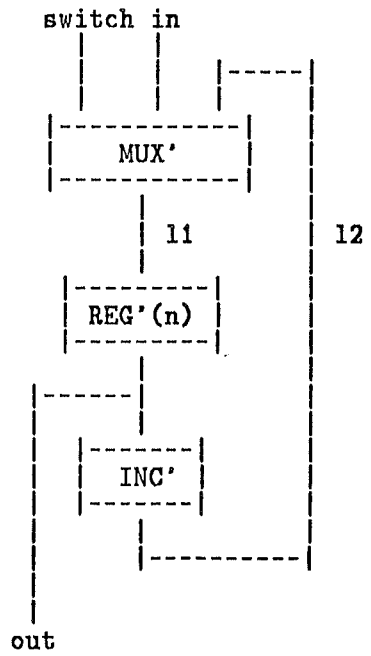
Thus INC is a combinational device that always outputs (on line o) one plus the value input (on line i).

3.2. Structural Specification

Behaviour specifications are used to directly specify what devices are supposed to do. LSM can also be used to describe the structure of digital systems as the interconnection of separate devices.

Renaming

Consider the system below:



This device is built from components similar to MUX, REG(n) and INC as described above, except that the line names have been changed. The multiplexer MUX' is

like MUX except that it has lines in, 12, 11 instead of i1, i2, o respectively. The need to rename lines motivates the following kind of expression:

```
e rn[l1=l1';...;ln=ln']
```

This expression denotes a behaviour similar to that denoted by e except that each line li is systematically renamed to li'. Suppose that MUX is as specified above, *i.e.* it satisfies:

```
MUX == dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX
```

then if we specify MUX' by:

```
MUX' == MUX rn[i1=in;i2=12;o=11]
```

then it will follow, using the renaming rule described later, that:

```
MUX' == dev{switch,in,12,11}.{11=(switch->in|12)};MUX'
```

Note that line switch has not been renamed.

The register REG' in the diagram above is defined by renaming the lines of the generic register REG by:

```
REG'(n) == REG(n) rn[i=11;o=out]
```

Using the renaming rule we can prove that if REG is defined as above, *i.e.* by:

```
REG(n) == dev{i,o}.{o=n};REG(i)
```

then:

```
REG'(n) == dev{11,out}.{out=n};REG'(11}
```

similarly we can define:

```
INC' == INC rn[i=out;o=12]
```

and then deduce:

```
INC' == dev{out,12}.{12=out+1};INC'
```

Note that instead of defining MUX', REG' and INC' by renaming lines of MUX, REG and INC, we could have defined them directly by the behaviour equations:

```
MUX'    == dev{switch,in,12,11}.{11=(switch->in|12)};MUX'
REG'(n) == dev{11,out}.{out=n};REG'(11}
INC'    == dev{out,12}.{12=out+1};INC'
```

Joining

To represent a schematic diagram, the first step is to define its components (either directly, or by renaming) so that lines which are to be connected have the same name. For example, in the diagram above we have arranged that output line of MUX' is the same as the input line to REG' (namely 11). The next step is

to write down an expression which denotes the result of connecting together the component devices. LSM has a special kind of expression for this purpose:

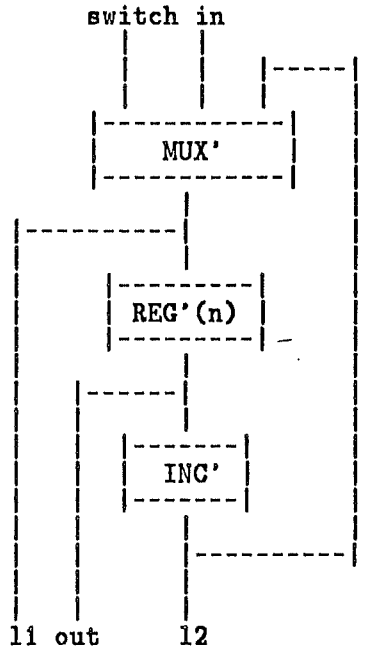
```
[| e1 | e2 | ... | en |]
```

This denotes the device that results from connecting together the devices specified by the ei's by joining lines with the same name. The lines of the resulting device are the union of the lines of each of the the component devices ei.

For example, if MUX', REG' and INC' are as above, then:

```
[| MUX' | REG'(n) | INC' |]
```

denotes the device with structure:



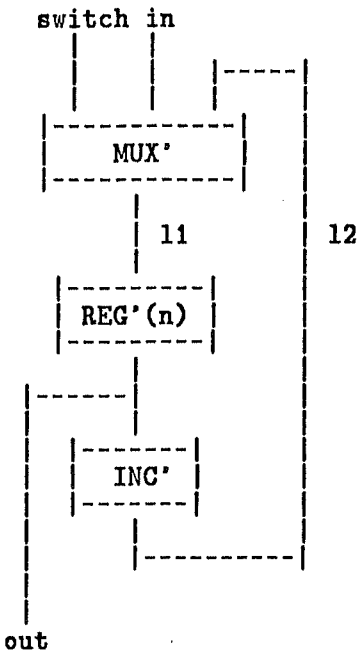
Hiding

In the diagram above the lines l1 and l2 are output lines. To represent the diagram in which these lines are internal we need another kind of expression.

```
e hide{l1,...,ln}
```

If e represents a system specified by a diagram with lines l1,...,ln, then the expression above represents the system specified by the same diagram except that lines l1,...,ln are internalised.

For example, the diagram:



can be represented by the expression:

```
[| MUX' | REG'(n) | INC' |] hide{11,12}
```

One can also explain the effect of hiding in terms of behaviour equations. Suppose COUNT_IMP1 is like COUNT_IMP (as described above) except that lines 11 and 12 are no longer internal, *i.e.*:

```
COUNT_IMP1 (n) ==  dev{switch,in,out,11,12}.
                      {11=(switch->in|12),out=n,12=out+1};
                      COUNT_IMP1(11)
```

If we define:

```
COUNT_IMP2(n) == COUNT_IMP1(n) hide{11,12}
```

then it can be deduced that:

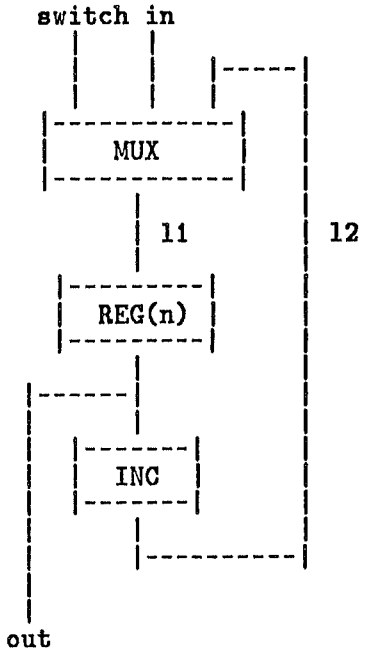
```
COUNT_IMP2 (n) ==  dev{switch,in,out}.
                      {11=(switch->in|12), out=n, 12=out+1};
                      COUNT_IMP2(11)
```

Notice that in this behaviour equation lines 11 and 12 are internal, whereas in the equation for COUNT_IMP1 they are output lines.

It is not necessary to introduce the constants MUX', REG' and INC'. One can simply write:

```
[| MUX    rn[i1=in;i2=12;o=11]
 | REG(n) rn[i=11;o=out]
 | INC    rn[i=out;o=12] |] hide{11,12}
```

When diagraming such expressions we will draw in explicit names to indicate how the lines of the devices have been renamed. For example, the expression above will be drawn as:



Note that MUX, REG and INC (as defined by behaviour equations above) have different line names to those in this diagram. For example, MUX was defined to have lines i1, i2 and o instead of in, 12 and 11. This illustrates our convention of using the names in diagrams to indicate renaming.

3.3. Terms and Types

LSM expressions contain *terms* such as 1, n, n+1 and (t->in|12). There are three kinds of such terms:

1. Constants (*e.g.* 1 and T).
2. Variables (*e.g.* n and 12).
3. Function applications of the form e1(e2) (or e1 e2) where e1 is a function and e2 its argument.

The terms n+1 and (t->in|12) are abbreviations for the function applications (+n)1 and ((COND t)in)12 respectively. Functions are terms; for example, + and

COND are constants.

Each LSM term has a *type*. A type is either *atomic* or *compound*. Examples of atomic types are num (the type of the constants 0, 1, 2 *etc.*) and bool (the type of the constants T and F). Compound types are built as follows:

- If ty1 and ty2 are types then so is ty1→ty2. It is the type of functions with arguments of type ty1 and results of type ty2.
- If ty1 and ty2 are types then so is ty1×ty2. It is the type of pairs with first component of type ty1 and second component of type ty2.

The constant + has type (num×num)→num and for *any* type *ty* the constant COND has type bool→((*ty*×*ty*)→*ty*).

We indicate that term t has type ty by writing t:ty. For example, 1:num, T:bool, +:(num×num)→num.

Expressions denoting sequential machines are terms of type dev. For example, MUX:dev, INC:dev, REG:num→dev, COUNT:num→dev *etc.*.

We adopt the convention that variables are written in lower case.

3.4. Summary of LSM

If x1, x2, ... , l1, l2, ... ,l1', l2', .. are variables and t1, t2, ... are terms and e, e1, e2, ... are expressions (*i.e.* terms of type dev) then LSM's notations for specifying devices are:

- Behaviour specification:

  dev{x1,...,xm}.{l1=t1,...,ln=tn};e

- Renaming:

  e rn[l1=l1';...;ln=ln']

- Joining:

  [| e1 | e2 | ... | en |]

- Hiding:

  e hide{l1,...,ln}

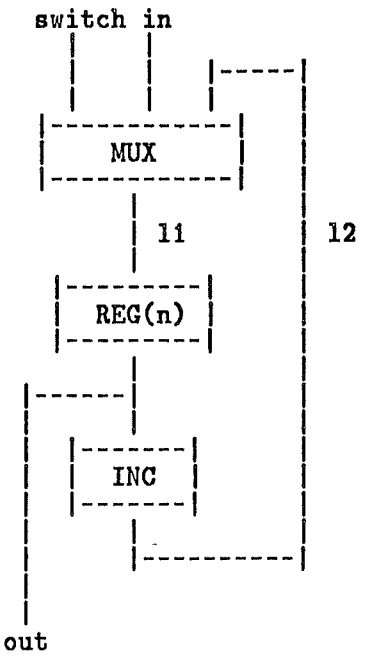
4. Verification by Formal Proof

We will explain the various proof rules of LSM by showing how they can be used to verify the COUNT example discussed above. We hope that the reader can deduce the general form of these rules from this.

We want to prove that a correct implementation of COUNT(n), where:

```
COUNT(n) == dev{switch,in,out}.{out=n};COUNT(switch->in|n+1)
```

is:



This diagram can be represented in LSM by introducing a constant COUNT_IMP defined by:

```
COUNT_IMP(n) ==  [| MUX    rn[i1=in;i2=12;o=11]
                 | REG(n) rn[i=11;o=out]
                 | INC    rn[i=out;o=12] |]
                hide{11,12}
```

Where the primitives used in this implementation are specified by:

```
MUX    == dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX
REG(n) == dev{i,o}.{o=n};REG(i)
INC    == dev{i,o}.{o=i+1};INC
```

We wish to verify that:

```
COUNT(n) == COUNT_IMP(n)
```

There are seven steps in proving this.

4.1. Unfolding Definitions

We start by expanding the definitions of the primitives in COUNT_IMP by replacing each instance of a left hand side of a behaviour equation by the corresponding right hand side. The behaviour equations defining the primitives used in COUNT_IMP are:

```
MUX    == dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX
REG(n) == dev{i,o}.{o=n};REG(i)
INC    == dev{i,o}.{o=i+1};INC
```

The definition of COUNT_IMP is:

```
COUNT_IMP(n) ==  [| MUX    rn[i1=in;i2=12;o=11]
                  | REG(n) rn[i=11;o=out]
                  | INC    rn[i=out;o=12] |]
                 hide{11,12}
```

Unfolding the definitions of the primitives in this yields the equation:

```
COUNT_IMP n ==
 [| (dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX) rn[i1=in;i2=12;o=11]
  | (dev{i,o}.{o=n};REG i) rn[i=11;o=out]
  | (dev{i,o}.{o=i+1};INC) rn[i=out;o=12] |]
 hide{11,12}
```

4.2. Line Renaming

The next step is to perform line renaming. For example, we replace:

```
(dev{switch,i1,i2,o}.{o=(switch->i1|i2)};MUX) rn[i1=in;i2=12;o=11]
```

by:

```
dev{switch,in,12,11}.{11=(switch->in|12)};(MUX rn[i1=in;i2=12;o=11])
```

The result of renaming is:

```
COUNT_IMP n ==
 [| dev{switch,in,12,11}.{11=(switch->in|12)};(MUX rn[i1=in;i2=12;o=11])
  | dev{11,out}.{out=n};(REG 11) rn[i=11;o=out]
  | dev{out,12}.{12=out+1};INC rn[i=out;o=12] |]
 hide{11,12}
```

4.3. Combining Equations

Now we come to the main step. We take the union of the output equations from each of the components to get a single set of equations for all the lines in the composite device. We also gather together the 'next-state' expressions of the components to get the following single expression for the whole:

```
COUNT_IMP n ==
 dev
  {switch,in,out}.
  {l1=(switch->in|l2),out=n,l2=out+1};
  [| MUX rn[i1=in;i2=l2;o=l1]
   | (REG l1) rn[i=l1;o=out]
   | INC rn[i=out;o=l2] |]
  hide{l1,l2}
```

Note that the set of input and output lines of the behaviour deduced for the combined device consists of the union of the lines of the component devices minus the hidden lines.

4.4. Folding

The 'next-state' part of the behaviour specification in this equation matches the right hand side of the definition of COUNT_IMP. The behaviour specification can thus be simplified by substituting in the appropriate instance of the left hand side of the definition of COUNT_IMP. The result of this is:

```
COUNT_IMP n ==
 dev
  {switch,in,out}.
  {l1=(switch->in|l2),out=n,l2=out+1};
  COUNT_IMP l1
```

4.5. Unwinding Equations

The set of equations for the values on the lines can now be solved. This consists in replacing:

```
{l1=(switch->in|l2),out=n,l2=out+1}
```

by:

```
{l1=(switch->in|n+1),out=n,l2=n+1}
```

to get:

```
COUNT_IMP n ==
 dev
  {switch,in,out}.
  {l1=(switch->in|n+1),out=n,l2=n+1};
  COUNT_IMP(switch->in|n+1)
```

4.6. Pruning Equations for Internal Lines

Next we notice that the equations for lines l1 and l2 are not used anywhere. Since these lines are internal we can delete the equations for them to get:

```
COUNT_IMP n == dev{switch,in,out}.{out=n};COUNT_IMP(switch->in|n+1)
```

4.7. Uniqueness of Solutions to Behaviour Equations

Notice that the last equation we derived shows that COUNT_IMP satisfies the same equation that was used to define the specification COUNT. From this we can deduce that:

```
COUNT(n) = COUNT_IMP(n)
```

because behaviour equations have unique solutions [5].

Although the COUNT example just done is trivial, it illustrates all the steps needed to verify the ECL chip.

5. Specification of the ECL Chip in LSM

Before giving a behaviour equation which specifies the ECL chip we must describe some of the types and constants that are needed.

The ECL chip specification makes use of a family of atomic types bool$n$ where n is a positive number. These are the types of n-tuples of booleans. A type trigger is also used; there are just two constants of this type, ON and OFF. The values ON and OFF correspond to the presence and absence respectively of an abstract synchronised clocking event. We could have used bool, T and F instead of trigger, ON and OFF but we wanted to make it explicit that we were abstracting away from the particular clocking scheme used. We discuss our approach to clocking in more detail later.

The following constants will be used:

- NOT:bool→bool - the complement function.
- AND:(bool×bool)→bool - the conjunction function. It can be infixed (*e.g.* one can write t1 AND t2 instead of (AND t1)t2).
- OR:(bool×bool)→bool - the disjunction function. It can be infixed.
- EQV:(bool×bool)→bool - the logical equality function. It can be infixed.
- =:(ty×ty)→bool (where ty is any type) is the general equality function. It can be infixed.
- CLOCKEVENT:trigger→bool - this maps ON to T and OFF to F.
- MK_BOOLn:(bool×...×bool)→bool$n$ - converts an n-tuple of booleans to something of type bool$n$.
- BmELn:bool$m$→bool - extracts the nth bit of an m-tuple.
- ELn:bool8→bool - abbreviation for B8ELn.

- SHIFTUP*n*: (bool*n*×bool)→bool*n* - shifts into the least significant position (the most significant bit of the *n*-tuple is lost).
- RINGCOUNT*n*: bool*n*→bool*n* - creates an *n*-tuple consisting of its argument moved one bit to the right, filled with the inverse of the previous rightmost bit.
- IMP_INV - is the complement function and is used to map from some specification states to implementation states (*e.g.* IMP_INV **stpreva = stholda**).

The complete behavioural specification of the ECL chip is given in Appendix 1. This specification uses several internal lines. These can be thought of as *virtual lines*, since there may be no actual physical lines in the implementation representing them. Such virtual lines are useful for structuring the specification. For example, the line **exringdata** is used to name the value of the data bit received from the ring. Other parts of the expression can use **exringdata** instead of the expression **(gisd->NOT stserdata|(stchangea AND stchangeb))**. There is no actual line carrying this value in the implementation.

Some state changes in the specification depend on conditional expressions of the form (CLOCKEVENT *xxx*) where *xxx* is some term of type **trigger**. These are state changes which depends on a clock other than the 'main clock'. An LSM term of type dev represents a sequential machine which has an implicit clock. It is common in synchronous circuits to derive from the main clock other clocks which are then used to effect certain state changes. The basic clock implicit in LSM defines the fundamental 'tick' of the logical state machine. Whether a state using a derived clock is re-evaluated or remains unchanged at a particular 'tick' of the fundamental clock is determined by its derived clock being ON or OFF. Thus the values ON and OFF indicate the presence or absence of a clocking event synchronous with the fundamental clock 'tick'.

This technique of dealing with several clocks seems to work well for devices that:

- are synchronous,
- define a single basic clock to which all other clocks can be explicitly related.

By introducing the special type **trigger** we have abstracted away from the actual clocking scheme used. Thus the clocking events may correspond to positive or negative edges or pulses, and the implementation could use edge-triggered or pulsed sequential devices.

6. Implementation of the ECL Chip

The ECL chip is implemented using the following components:

INV	:dev	single input inverter
NOR2	:dev	two input NOR gate
NOR3	:dev	three input NOR gate
DTYPE1Q	:bool→dev	D flipflop one input q output
DTYPE2Q	:bool→dev	D flipflop two inputs q output
DTYPE2QBAR	:bool→dev	D flipflop two inputs qbar output
DTYPE1QQBAR	:bool→dev	D flipflop one input q, qbar outputs
DTYPE2QQBAR	:bool→dev	D flipflop two inputs q, qbar outputs

We take these devices to be primitive and specify them by:

```
INV ==
 dev{in,out}.{out = NOT in};INV

NOR2 ==
 dev{in0,in1,out}.{out = NOT(in0 OR in1)};NOR2

NOR3 ==
 dev{in0,in1,in2,out}.{out = NOT((in0 OR in1) OR in2)};NOR3

DTYPE1Q b ==
 dev{data,q}.{q = b};DTYPE1Q(data)

DTYPE2Q b ==
 dev{data1,data2,q}.{q = b};DTYPE2Q(data1 OR data2)

DTYPE2QBAR b ==
 dev{data1,data2,qbar}.{qbar = NOT b};DTYPE2QBAR(data1 OR data2)

DTYPE1QQBAR b ==
 dev{data,q,qbar}.{q = b, qbar = NOT b};DTYPE1QQBAR(data)

DTYPE2QQBAR b ==
 dev{data1,data2,q,qbar}.{q = b, qbar = NOT b};DTYPE2QQBAR(data1 OR data2)
```

Circuit diagrams for parts of the ECL chip are given in the Appendix 2. The first of these diagrams corresponds to the following LSM structural specification:

```
IMP_ECL(stpreva,stchangea,stmoderr,stprevb,stchangeb,
        stserdata,stdata,stdshift4,stgap,stgapend,
        sth,strc,stdl,stdr,stleft,stlna,stlnb,stphase,stserout) ==
 [| DEMOD(stpreva,stchangea,stmoderr,stprevb,stchangeb,stserdata,stdata)
  | SHIFT4 stdshift4)
  | DETGAP(stgap,stgapend,sth)
  | RINGCOUNT strc
  | SHIFTREGS(stdl,stdr,stleft)
  | MODUL(stlna,stlnb,stphase,stserout) |]
 hide{di0,di1,data,d21,d2r,gapendbar}
```

The reference description of the ECL chip implementation that we worked from was expressed, not in LSM, but in the notation used by the Cambridge Design Automation System, a stylised subset of Modula-2. To interface to this we wrote a translator from the subset of Modula-2 to LSM. For efficiency reasons the translator directly creates the equation that would result from unfolding, unwinding, pruning *etc.* The LSM specification of the primitive components is encoded in the Cambridge DA system and the translator creates the LSM which describes the behaviour of the implementation.

Using the circuit description of the Cambridge DA system means that the circuit being verified is identical to the one being operated upon by the simulation and layout tools.

7. Verifying the ECL Chip

The functional correctness of the ECL chip is expressed by the equation:

```
SPEC_ECL(states) == IMP_ECL(states)
```

where `states` ranges over state vectors (which are 19-tuples).

The structure of the proof matches the structure of the circuit as described in Appendix 2. Thus each component is first specified and verified and the resulting lemmas are then used to show that the top-level specification of the whole chip is correctly implemented. The verification of each component is just like the `COUNT` example, except that the unfolding, unwinding *etc.* is done automatically by the Modula-2 to LSM translator.

We briefly describe the specification and verification of a simple component of the ECL chip - the demodulator (called `DEMOD`).

The specification of the demodulator is:

```
SPEC_DEMOD(stpreva,stchangea,stmoderr,stprevb,stchangeb,stserdata,stdataout)
==
dev{ina,inb,serin,gisd,moderr,di0,di1,dataout}.
   {moderr = stmoderr,
    di0 = (gisd -> F | stchangea AND stchangeb),
    di1 = (gisd -> stserdata | F),
    dataout = stdataout};
   SPEC_DEMOD(ina,
              NOT (ina EQV stpreva),
              (NOT stchangea) AND (NOT stchangeb),
              inb,
              NOT (inb EQV stprevb),
              serin,
              (gisd -> NOT stserdata | stchangea AND stchangeb))
```

The Modula-2 description of the demodulator component is:

(cf. circuit diagram for DEMOD in Appendix 2)

```
BEGIN
    BeginNode("Demodulator", "DEMOD");

    (*  Create and initialise some data structures   *)

    (*  Main code to construct device                *)

    inabar := Not("inabar", ina);
    dType12(inabar, ck, holda, holdabar);
    la1 := Nor2("la1", ina, holda);
    la2 := Nor2("la2", inabar, holdabar);
    dType22(la1, la2, ck, changea, changeabar);
    errout := Nor2("errout", changea, changeb);

    inbbar := Not("inbbar", inb);
    dType12(inbbar, ck, holdb, holdbbar);
    lb1 := Nor2("lb1", inb, holdb);
    lb2 := Nor2("lb2", inbbar, holdbbar);
    dType22(lb1, lb2, ck, changeb, changebbar);

    serinbar := Not("serinbar", serin);
    dType11 (serinbar, ck, sholddata);
    gisdbar := Not("gisdbar", gisd);

    di0 := Nor3("di0", changeabar, changebbar, gisd);
    di1 := Nor2("di1", sholddata, gisdbar);

    dType21(di0, di1, ck, data);

    OutputPin(1, moderr);
    OutputPin(2, di0);
    OutputPin(3, di1);
    OutputPin(4, data);
    InputPin(5, ina);
    InputPin(6, inb);
    InputPin(7, serin);
    InputPin(8, gisd);

    EndNode("DEMOD");

END Demod.
```

This is translated automatically to:

```
IMP_DEMOD(stholda,stchangea,stmoderr,stholdb,stchangeb,stsholddata,stdata) ==
dev{ina,inb,serin,gisd,moderr,di0,di1,data}.
   {moderr = stmoderr,
    di0 = (NOT(((NOT stchangea) OR (NOT stchangeb)) OR gisd)),
    di1 = (NOT(stsholddata OR (NOT gisd))),
    data = stdata};
   IMP_DEMOD(NOT ina,
             (NOT(ina OR stholda)) OR (NOT((NOT ina) OR (NOT stholda))),
             NOT(stchangea OR stchangeb),
             NOT inb,
             (NOT(inb OR stholdb)) OR (NOT((NOT inb) OR (NOT stholdb))),
             NOT serin,
             (NOT(((NOT stchangea) OR (NOT stchangeb)) OR gisd)) OR
             (NOT(stsholddata OR (NOT gisd))))
```

Some of the abstract states of the specification do not have identical corresponding states in the implementation. A function is defined which maps from the specification states to the implementation states. The function IMP_INV maps `stpreva`, `stprevb` and `stserdata` to `stholda`, `stholdb` and `stsholddata` respectively.

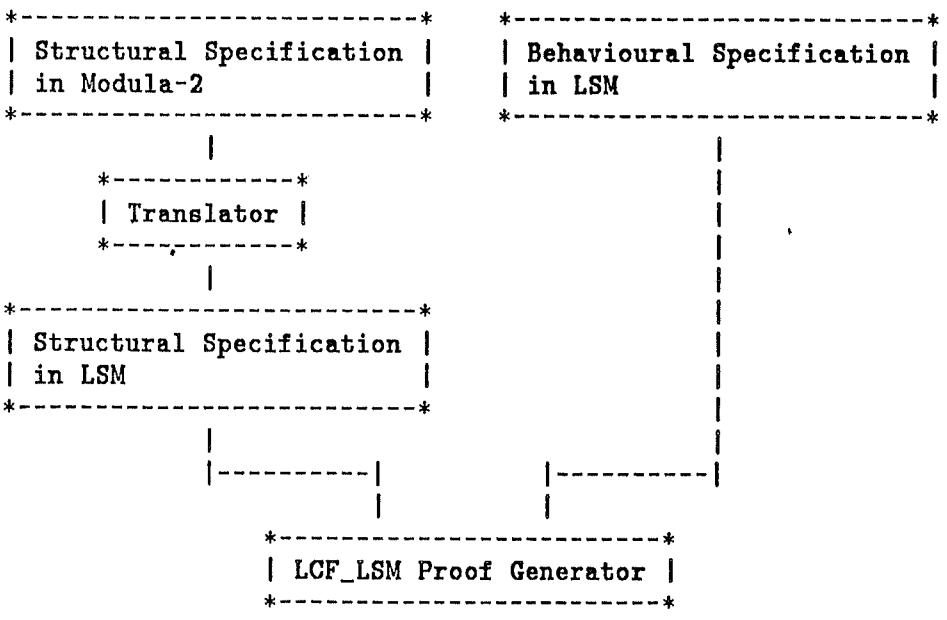
To verify the demodulator design we must show that:

```
SPEC_DEMOD(IMP_INV stpreva,stchangea,stmoderr,IMP_INV stprevb,
           stchangeb,IMP_INV stserdata,stdata)  ==
IMP_DEMOD(stpreva,stchangea,stmoderr,stprevb,stchangeb,stserdata,stdata)
```

to prove this we must prove the following lemmas:

```
(gisd -> F | stchangea AND stchangeb) =
 NOT((NOT stchangea OR NOT stchangeb) OR gisd)

(gisd -> stserdata | F) = NOT((IMP_INV stserdata) OR NOT gisd)

NOT (ina EQV stpreva) = NOT(ina OR (IMP_INV stpreva))
                         OR  NOT(NOT ina OR NOT (IMP_INV stpreva))

NOT stchangea AND NOT stchangeb = NOT(stchangea OR stchangeb)

NOT (inb EQV stprevb) = NOT(inb OR (IMP_INV stprevb))
                         OR NOT(NOT inb OR NOT (IMP_INV stprevb))

(gisd -> stserdata | stchangea AND stchangeb)) =
 NOT(((NOT stchangea) OR (NOT stchangeb)) OR gisd)
 OR NOT((IMP_INV stserdata) OR (NOT gisd))
```

These are all straightforward to do using the LCF part of LCF_LSM. The overall structure of the verification process can be summarised by:



7.1. Uncovering Design Errors

The purpose of verification in the context of practical circuit design is as much the discovery of errors as the proof of their absence. Most hardware designers converge on a solution via a process of successive approximation. It is very important to be able to locate inconsistencies between the specification and implementation. In the ECL chip verification, no major design flaws were found but some incorrect wiring of the implementation was discovered. There were also some mistakes in the initial specification. As formal verification proceeds the specification becomes tighter. Failure of verification can result in changes to the implementation or the specification.

The LCF_LSM system gives a clear indication of where errors are located when the formal proof fails. A typical failure might result in:

```
Unsolved goal: "state0 = NOT state0"     ["x OR y = T"]
```

This indicates that the system cannot prove `"state0 = NOT state0"` when `x OR y` has the value `T`. If the specification is correct then we must change the design - maybe insert an inverter or reconnect some signals in that particular part of the circuit.

8. Conclusions

As others have found, the lemmas needed to prove hardware correct are mostly trivial. We conjecture that the entire ECL chip proof could be done automatically by, for example, the VERIFY system [6]. This is in marked contrast to the situation with software verification; there the verification conditions produced by real programs are usually well beyond the current state-of-the-art in automatic theorem proving. This is one of the reasons why we believe that hardware verification by formal proof will become practical sooner than software verification by proof.

LSM, although nice and simple, is clumsy for expressing certain kinds of things. We found it necessary to introduce extra state variables to hold past values, whereas it would have been more natural to have a notation for denoting these values directly.

The use of two separate descriptive formalism, LSM and Modula-2, was messy and necessitated the writing of a translator. Although this was straightforward, it introduced one more place where insecurity could creep in. Also it forced the user to learn two languages, where one should have been enough.

Despite the various problems just mentioned, we believe that hardware verification by formal proof is here to stay. In some cases it is already the only feasible way of attaining satisfactory security. Recently John Cullyer of The Royal Signals and Radar Establishment (RSRE) has been arguing that the quality of embedded aerospace systems is declining [7]. This is happening just at the time when the integrity of these systems is becoming more essential. Future aircraft control systems will carry a much heavier burden, as the aircraft being controlled will be designed in ways that make them unflyable manually. According to Cullyer, future aerospace microprocessors will have to be formally specified and verified. Until recently this was not considered feasible, but the group at RSRE have designed and proved correct (by manual proof using LSM) a simple processor called VIPER. This establishes that the task is possible, and so contractors will no long be able to claim that it is unreasonable to require them to do formal verification. Aircraft are not the only source of life-critical closed-loop control systems, other examples include nuclear reactors, chemical plants and medical devices (such as microprocessor controlled pacemakers).

9. Current Research and Future Prospects

We have implemented a successor to LCF_LSM which overcomes the lack of expressiveness mentioned in the previous section. The new system is also based on LCF but uses classical Higher-Order Logic instead of LSM. There are no special hardware oriented terms in the new system because everything can be expressed directly in logic. For example, joining and hiding can be represented by conjunction and existential quantification respectively. This idea of using a pure logic, rather than an *ad hoc* calculus such as LSM, is due to Keith Hanna.

Only register-transfer level specifications can be expressed in LSM since everything is assumed to be a sequential machine. This precludes the verification of certain kinds of implementations whose functioning depends on asynchronous delays. For example, the normal implementation of a D-type register uses NAND gates, the state being held in feedback loops. To verify such a register one must take into account the delay in combinational devices. We have been working on this by studying several levels of behavioural description, together with mappings between them. This work fits nicely into the Higher Order Logic framework; it is based on ideas from the VERITAS project [8].

The LCF system is a proof assistant, the user proves theorems by setting up

goals and then invoking proof strategies (called *tactics* in LCF jargon). Many trivial lemmas which are well within the current capabilities of automatic theorem proving have to be done manually. Although we are slowly developing a set of general purpose tactics, our current repertoire does not provide as much automation as is possible. LCF_LSM is satisfactory for LCF experts, because they can usually see what tactics are needed for a given goal, but it is unusable by anyone else, *e.g.* circuit designers. We plan to increase the automation by adding more powerful theorem-proving strategies. In particular, we hope to incorporate ideas from Barrow's impressive VERIFY system and the Boyer-Moore Theorem Prover. We also hope to add tactics corresponding to well known decision procedures (such as Presburger arithmetic). We still think theorem-proving experts will sometimes be needed, but we hope to reduce the need for them as much as possible.

The function of a complex system must be specified and verified along a number of dimensions. Typically one must specify:

- Detailed timing (minimum clocking rate, gate delays *etc.*).
- Register-transfer behaviour.
- Relationships between abstractions (numbers, bitstrings *etc.*) and realisations (truth values, voltages *etc.*).

For each of these aspects of behaviour there exist specialised languages: interval arithmetic for timing, LSM for register-transfer behaviour and the theory of homomorphisms for data-type relationships. Each aspect of behaviour is usually analysed using different tools operating on different representations. For example, timing analysers work from *circuit descriptions*, simulators often require *procedural models* (Modula-2 programs in the case of the Cambridge Design Automation System) and functional verifiers manipulate *sentences* in formal logics like LSM. There are important connections between the various different aspects of behaviour, for instance a 'high-level' behavioural descriptions may only be valid provided certain timing requirements are met (*e.g.* a register may only have a desired latching behaviour if the signals conform to specified setup and hold times). It is thus important to be able to state conditions which guarantee the consistency between different kinds of descriptions. This is much easier if the descriptions are expressed in a unified framework. We believe that Higher-Order Logic provides the basis for such a framework. Various design automation tools can be viewed as logical inference rules. For example, a timing analyser can be viewed as a rule for inferring sentences about timing relationships from sentences describing the structure of a circuit. Recent work by Ben Moszkowski [9] has shown how logic specification can

be the basis of efficient simulators.

We hope to use the new system we are developing as the top-level user-interface for a *verification oriented design environment.* This top-level, like LCF, will be based on the functional language ML[10]. Although logic sentences will be the primary formalism used for hardware specification, it is planned to interface other existing tools to the system. These tools will be made into ML functions which operate on logic sentences.

10. Acknowledgements

The ECL chip was designed by Andy Hopper. The LCF_LSM system is based on Cambridge LCF [11] which in turn is based on Milner's original Edinburgh LCF. The LSM logic is inspired by CCS [12] which originated the idea of representing interconnections of devices with the operations of joining, hiding and renaming. The semantics of LSM is derived from Milner's early work on processes [13]; it is also related to that of SCCS [14]. Interaction with the CIRCAL [15] group has been very stimulating.

Part of the work was supported by SERC/Alvey grant number GR/D/17304 entitled "Formal Methods for Hardware Verification".

11. References

1. M. J. C. Gordon. "LCF_LSM". University of Cambridge Computer Laboratory Technical Report No. 41. 1983.

2. M. J. C. Gordon, R. Milner and C. Wadsworth. "Edinburgh LCF: A mechanised logic of computation". Lecture Notes in Computer Science Number 78, Springer-Verlag, 1979.

3. A. Hopper. "The Cambridge Fast Ring", University of Cambridge Computer Laboratory, Internal Report. 1984.

4. P. Robinson and J. Dion. "Programming Languages for Hardware Description". In proceedings of the *20th Design Automation Conference*. 1983.

5. M. J. C. Gordon. "The Denotational Semantics of Sequential Machines". Information Processing Letters, Volume 10, Number 1, 1980

6. H. G. Barrow. "Proving the Correctness of Digital Hardware Designs". VLSI DESIGN, July 1984.

7. W. J. Cullyer. "Hardware Integrity". CAA Review, RSRE Malvern, June 1985.

8. F. K. Hanna. "Overview of the VERITAS Project". Electronics Laboratory, University of Kent at Canterbury. 1983.

9. B. Moszkowski. "Executing Temporal Logic Programs". Proceedings of the NSF/SERC Workshop on the Semantics of Concurrency, Carnegie-Mellon University, Pittsburgh, Pennsylvania, U.S.A., July 1984.

10. M. J. C. Gordon. "Representing a Logic in the LCF Metalanguage". In "Tools and Notions for Program Construction" (ed. D. Neel), Cambridge University Press, 1982

11. L. C. Paulson. "The Revised Logic PPLAMBDA: A Reference Manual". University of Cambridge Computer Laboratory Technical Report Number 36, 1983. "A Higher Order Implementation of Rewriting". Science of Computer Programming. 1983. "Tactics and Tacticals in Cambridge LCF". University of Cambridge Computer Laboratory Technical Report Number 39, 1983.

12. R. Milner. "A Calculus of Communicating Systems". Lecture Notes in Computer Science No. 92, Springer-Verlag. 1980.

13. R. Milner. "Processes: A Mathematical Model of Computing Agents". Proceedings Logic Colloquium '73, edited by Rose and Shepherdson, North Holland, 1973.

14. R. Milner. "On Relating Synchrony and Asynchrony". Department of Computer Science Internal Report CSR-75-80, University of Edinburgh, 1080.

15. G. J. Milne. "CIRCAL: A calculus for circuit description". Integration, Vol. 1, Nos. 2 & 3, October 1983.

12. APPENDIX 1: Specification of the ECL Chip

The top level specification of the ECL chip in LSM uses the state variables listed below. These are *abstract states* and do not necessarily correspond to values actually stored in distinct registers.

States

`stpreva`, `stprevb` hold the values on lines **ina** and **inb** respectively at the previous clock tick.

`stchangea`, `stchangeb` hold the value T if the value on the corresponding line (**ina** or **inb** respectively) has changed over the two previous clock ticks; otherwise they hold F.

`stserdata` holds the value on the serial data line.

`stmoderr` is T if a modulation error occurred, otherwise it is F.

`stdata` holds the value of the data received from the ring. If the line **gisd** ("gate in serial data") has been asserted then this corresponds to serial data otherwise it is the demodulated data.

`stshift4` is a 4-bit shift register which delays the data received from the ring by 4 clock periods.

`stgap` samples (at the byte frequency) the value of the **gap** line.

`stgapend` has the value T when the end of a gap between packets has been reached; otherwise it has the value F.

`sth` is a 2-bit state which is used in a ringcounter at the end of a gap to count 3 periods during which the **gap** input is disabled.

`strc` is the state of the ringcounter which is used to generate the byte frequency clock (ck8).

`stdl , stdr` are two 8-bit shift registers which hold the bytes received from the ring and received from the CMOS logic. The registers are used to convert parallel data to serial data and serial to parallel.

`stleft` is inverted on successive bytes and is used to switch between the use of registers **stdl** and **stdr** in the conversion of data.

`stlna , stlnb` hold the values which are to be output on the lines **lna** and **lnb** respectively.

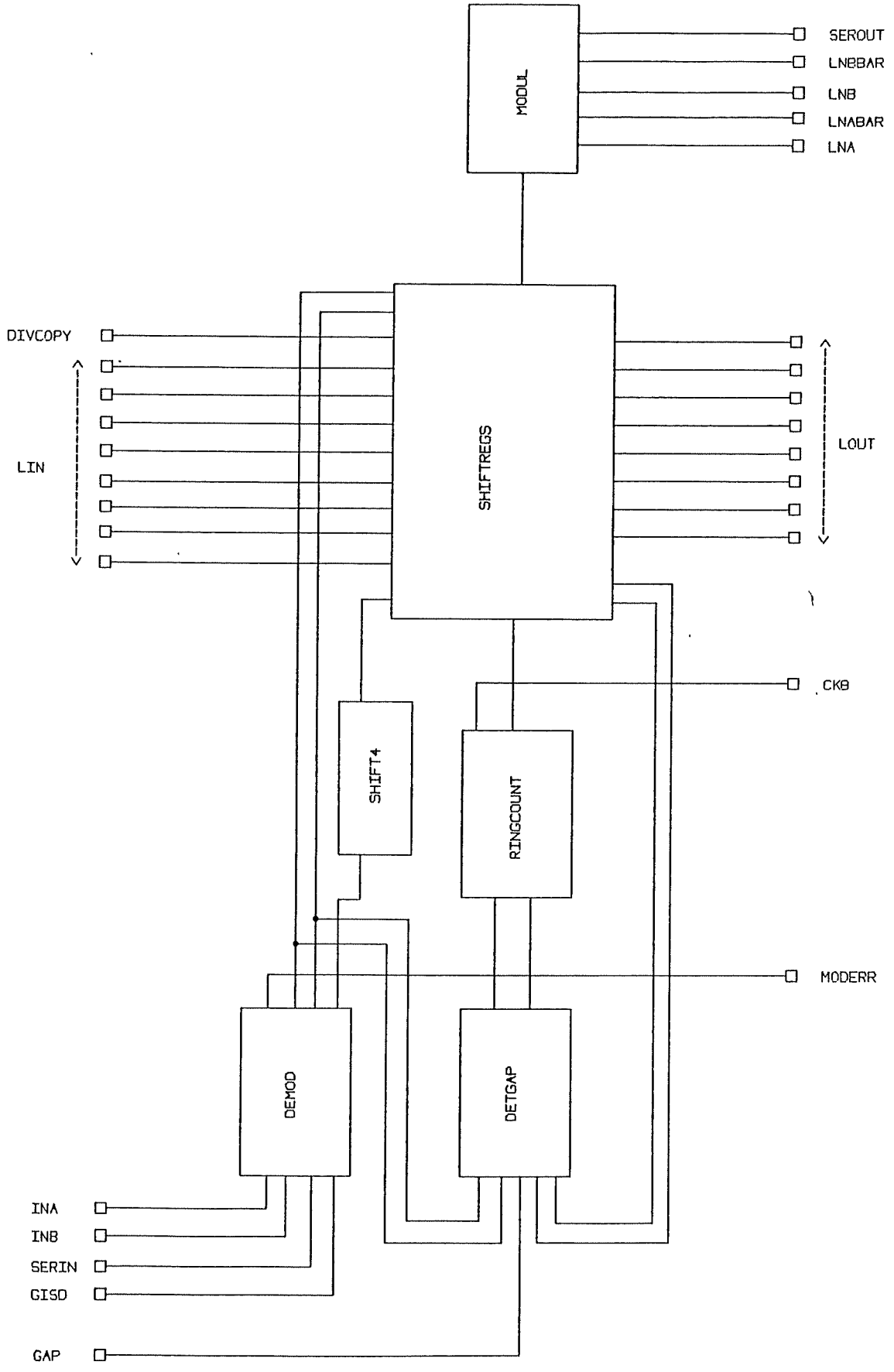
`stphase` is inverted on each clock tick and is used to balance the value changes on lines **lna** and **lnb** . (At least one line is always changed and by balancing the changes evenly we reduce the possible electrical degradation of signals.)

`stserout` holds the value to be output as serial data onto the ring. In the copy mode this is the data received from the ring; in the divide mode (the usual case) it is data in the shift registers **stdl** and **stdr** which has been received from the CMOS chip.

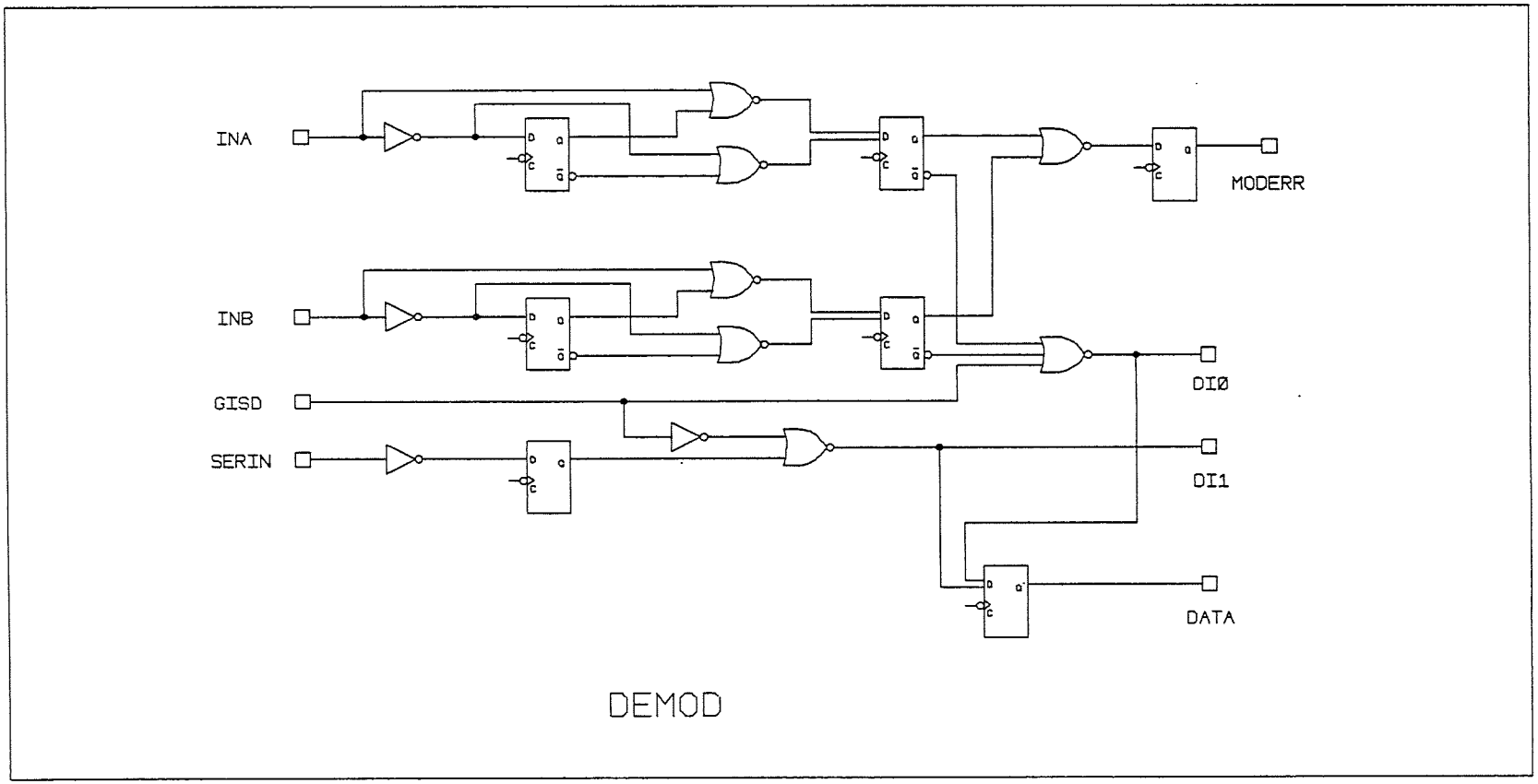
The LSM specification of the ECL chip is:

```
SPEC_ECL(stpreva,stchangea,stmoderr,stprevb,stchangeb,
         stserdata,stdata,stdshift4,stgap,stgapend,
         sth,strc,stdl,stdr,stleft,stlna,stlnb,stphase,stserout) ==
dev{ina,inb,serin,gisd,moderr,ck8,gap,lin,divcopy,lout,lna,lnabar,lnb,lnbbar,serout}.
   {moderr     = stmoderr,
    exringdata = (gisd-> stserdata | (stchangea AND stchangeb)),
    d4         = B4EL3 stdshift4,
    state0h    = MK_BOOL2(T,T),
    nogapbar   = (sth = state0h) AND (NOT reset),
    regdata2   = (EL2 stdl) OR (EL2 stdr),
    reset      = (stgapend AND regdata2),
    p0         = (B4EL0 strc) AND (B4EL3 strc),
    state0rc   = MK_BOOL4(T,T,T,T),
    state0to3  = B4EL3 strc,
    state4to7  = NOT(B4EL3 strc),
    state7rc   = MK_BOOL4(T,T,T,F),
    ckcond     = B4EL2 strc AND ((NOT stgapend) OR state0to3),
    ck8        = (state4to7 -> (reset->ON|(ckcond->ON|OFF)) | OFF),
    left       = stleft,
    right      = NOT stleft,
    ckl        = ((left OR p0)  -> ON | OFF),
    ckr        = ((right OR p0) -> ON | OFF),
    dataout    = (divcopy -> (right->(EL7 stdl)|(EL7 stdr)) | exringdata),
    lout       = (right-> stdr | stdl),
    lna        = stlna,
    lnabar     = NOT stlna,
    lnb        = stlnb,
    lnbbar     = NOT stlnb,
    serout     = stserout} ;
   SPEC_ECL(ina,
            NOT (ina EQV stpreva),
            (NOT stchangea) AND (NOT stchangeb),
            inb,
            NOT (inb EQV stprevb),
            serin,
            exringdata,
            SHIFTUP4(stdshift4, stdata),
            (CLOCKEVENT ck8 -> gap | stgap),
            (NOT reset ->
            (stgapend->stgapend|(stgap AND nogapbar->exringdata|F))|F),
            (CLOCKEVENT ck8 -> (nogapbar->state0h| RINGCOUNT2 sth) | sth),
            (reset -> state0 | (stgapend AND state4to7 ->
                                ((strc=state7rc)->state7rc|RINGCOUNT4 strc)|
                                RINGCOUNT4 strc)),
            (CLOCKEVENT ckl -> (left ->lin|SHIFTUP8(stdl,d4)) | stdl),
            (CLOCKEVENT ckr -> (right->lin|SHIFTUP8(stdr,d4)) | stdr),
            (CLOCKEVENT ck8 -> NOT stleft | stleft),
            ((stphase OR dataout) -> NOT stlna | stlna),
            ((NOT stphase) OR dataout) -> NOT stlnb | stlnb),
            NOT stphase,
            dataout)
```

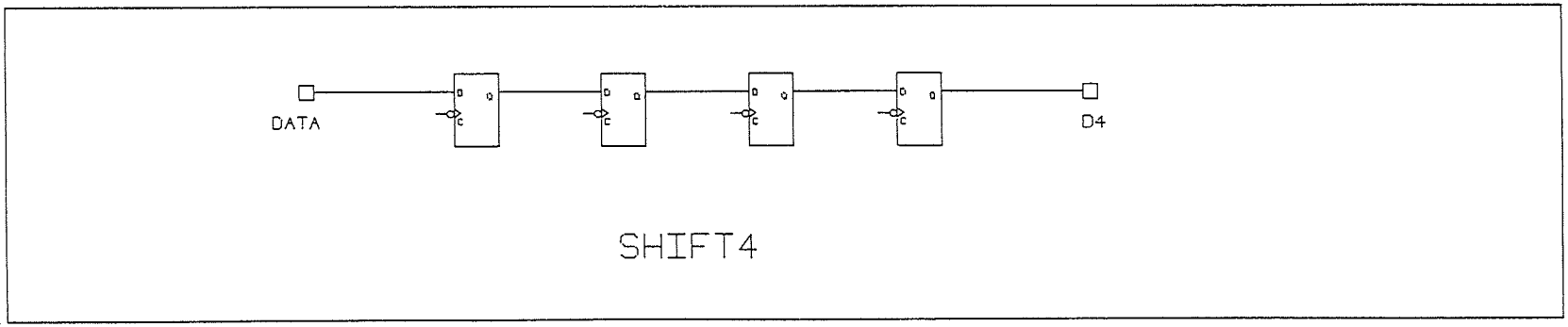
13. APPENDIX 2: Circuit Diagrams



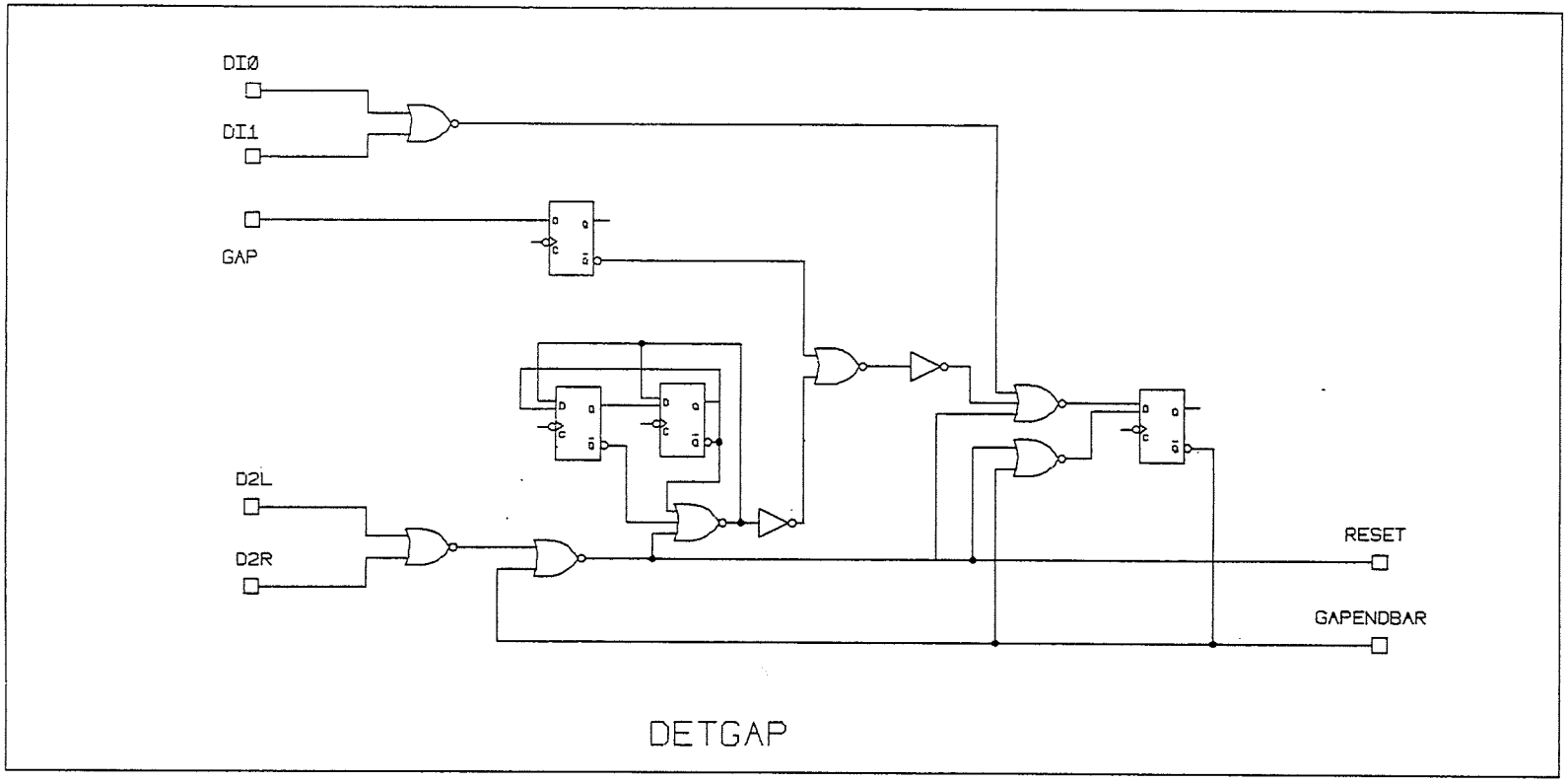
BLOCK DIAGRAM OF ECL CHIP



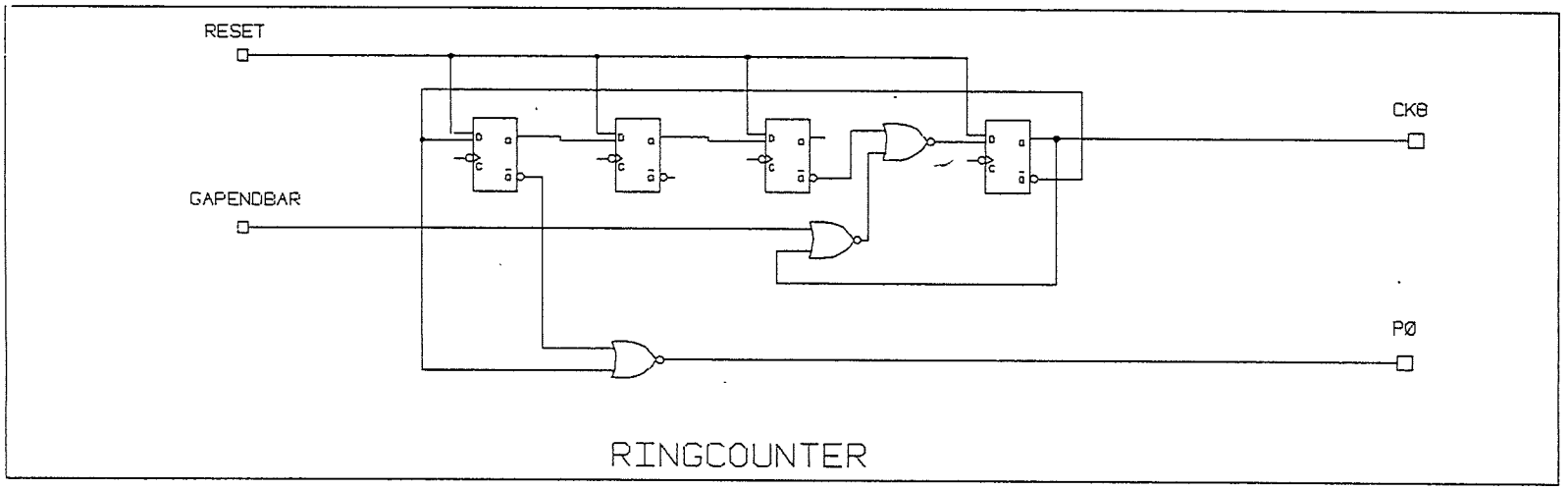
DEMOD



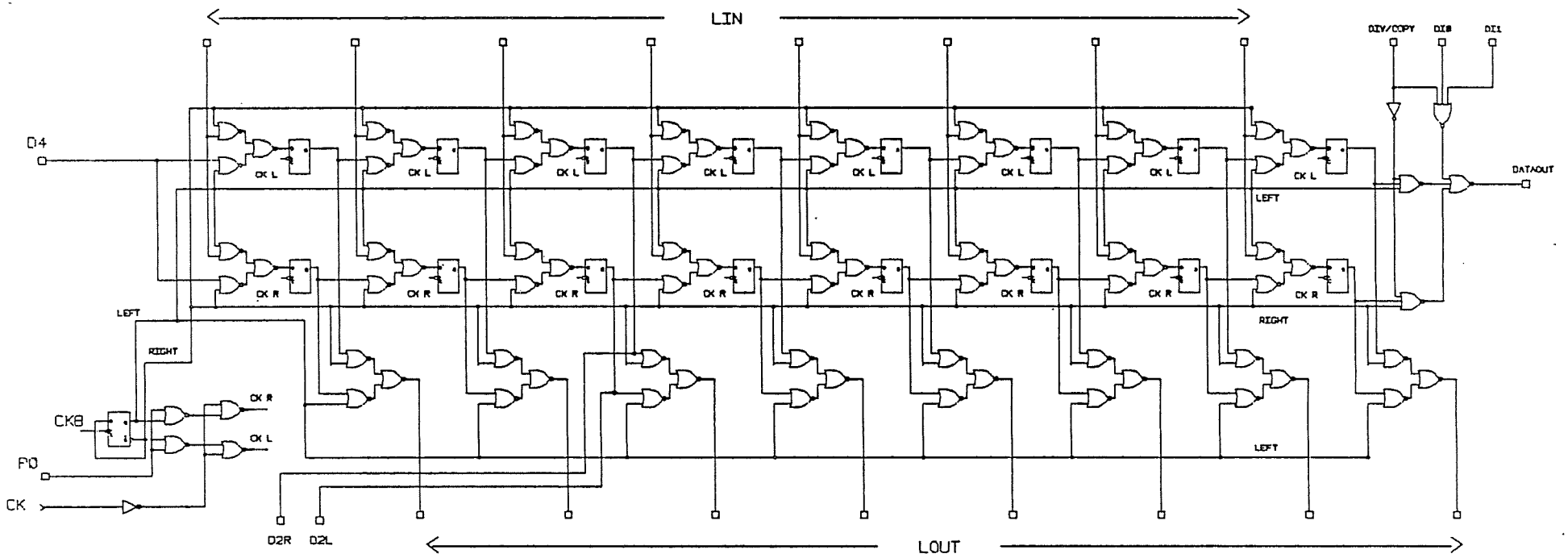
SHIFT4



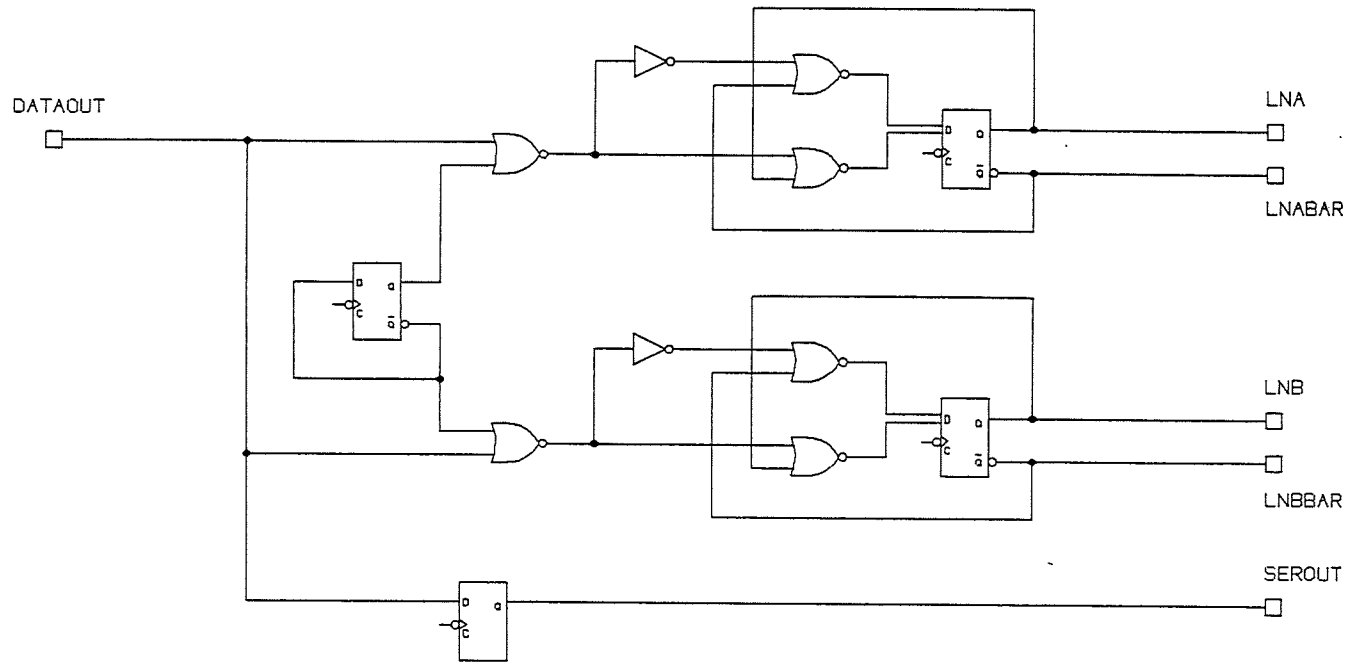
DETGAP



RINGCOUNTER


LIN
DIV/COPY
DI8
DI1
DI4
DATAOUT
CK L
CK R
LEFT
RIGHT
CK8
P0
CK
D2R
D2L
LOUT

SHIFTREGS



MODUL