Technical Report

UCAM-CL-TR-123
ISSN 1476-2986

Number 123

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Case study of the Cambridge Fast Ring ECL chip using HOL

John Herbert

February 1988

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

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

© 1988 John Herbert

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

Case Study of the Cambridge Fast Ring ECL Chip using HOL

John Herbert
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG

Abstract
This article describes the formal specification and verification of an integrated circuit which is part of a local area network interface. A single formal language is used to describe the structure and behaviour at all levels in the design hierarchy, and an associated computerised proof assistant is used to generate all formal proofs. The implementation of the circuit, described as a structure of gates and flip-flops, is verified via a number of levels with respect to a high-level formal specification of required behaviour. The high-level formal specification is shown to be close to precise natural language description of the circuit behaviour.

The specification language used, HOL [Gordon85a], has the advantage of permitting partial specifications. It turns out that partial specification has an important effect on the specification and verification methodology, and this is presented. We have also evaluated aspects of conventional design, such as techniques for locating errors and the use of simulation, within the case study of formal methods. We assert that proof strategies must assist error location and that simulation has a rôle alongside formal verification.

Preface

This report is essentially a self-contained portion of my thesis [1]. The following is a copy of my acknowledgement from the thesis. In addition I would like to thank Dr. Mike Gordon and Dr. Mary Sheeran for their comments.

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

I wish to thank my supervisor, Dr. Andy Hopper, for his help and support. I am grateful to Professor Roger Needham, the head of the Computer Laboratory, for the opportunity to work at Cambridge and for his contribution to a good working environment. I have been fortunate to receive financial support through scholarships from the Royal Commission for the Exhibition of 1851 and Corpus Christi College, Cambridge, and an award from the Lundgren Research Fund.

I would like to thank those people who have read and commented on earlier drafts of this dissertation: Andy Hopper, Jeff Joyce, Miriam Leeser, Tom Melham, Larry Paulson and Frances Quigg. Frances Quigg read early drafts diligently and helped correct my presentation of ideas; Miriam Leeser commented closely on the chapters on timing.

The hardware verification group at Cambridge has provided a friendly and stimulating work environment. I am especially grateful to Mike Gordon for leading the way in hardware verification and giving me advice and encouragement when it was needed. Special thanks also to the other HVG roadshow members — Albert, Inder, Miriam and Tom.

[1] Application of Formal Methods to Digital System Design, J. M. J. Herbert, Ph D Thesis University of Cambridge, 1986

Contents

1 Introduction 3

2 HOL 3
- 2.1 Logic . . . 3
- 2.2 Specification and Verification of an Example Device . . . 4
  - 2.2.1 Behavioural specification of a device . . . 4
  - 2.2.2 Components . . . 5
  - 2.2.3 Representation of structure in HOL . . . 6
  - 2.2.4 Deduction of implementation behaviour . . . 7
  - 2.2.5 Proof of correctness . . . 7

3 The Cambridge Fast Ring ECL Chip 8

4 Methodology in Case Study 10
- 4.1 Behaviour of primitive components . . . 11
- 4.2 Specification and Verification of the Demodulator . . . 12
  - 4.2.1 Specification of demodulator . . . 12
  - 4.2.2 Demodulator implementation . . . 14
  - 4.2.3 Proof of correctness . . . 15

5 Implementation of the ECL Chip 16
- 5.1 Description of Implementation . . . 16

6 Specification and Verification of the ECL Chip 17
- 6.1 Top-level Specification . . . 17
- 6.2 Specification in Detail . . . 19
- 6.3 Specification Summary . . . 23
- 6.4 Verification of the ECL Chip . . . 23

7 Features of the Specification 24
- 7.1 HOL as a hardware description language . . . 24
- 7.2 State in the Specification . . . 25
- 7.3 Partial Specification . . . 25
- 7.4 Clock Cycles . . . 25
- 7.5 Complexity of Behaviour . . . 26
- 7.6 Modelling Difficulties . . . 26
- 7.7 Interface Specifications . . . 27

8 **Verification** 27

8.1 Proof size 27

8.2 Non State-Based Behavioural Descriptions 27

8.3 Proof Using a Formal Simulator 28

8.4 Proof of an n-Bit Shift Register 28

8.5 Verification of Partial Specifications 29

9 **Discussion** 31

9.1 Correctness of a Real Design 31

9.2 Methodology 31

1 Introduction

The need for formal techniques in the design of digital systems has been stated in recent years and a number of researchers have formally specified and verified digital designs. These designs have usually belonged to a standard class of designs, such as microprogrammed devices [Hunt85] or mathematical functions [Camilleri86], and are not usually fabricated or used in a real application.

In this case study we have specified and verified a real digital design, the ECL chip of the Cambridge Fast Ring. By *real* we mean that the integrated circuit was designed and fabricated for a practical purpose, independent of the case study. It is an application specific circuit and does not fit into any category of previously verified circuits.

All aspects of the circuit behaviour and structure are specified in a formal language, HOL [Gordon85a], and the proofs of correctness are achieved using the mechanised proof assistant for this language. A previous article [Gordon86] describes a case study of the same integrated circuit using a different specification language and different proof assistant. We will make some references to this study to illustrate some attributes of the newer formalism.

2 HOL

Prompted by the use of standard forms of logic to reason about hardware [Hanna83] [Moszkowski83], Gordon developed the HOL system for hardware specification and verification.

HOL is a computer based formalisation of higher-order logic. The HOL logic is a version of Church's Simple Type Theory with the addition of polymorphic types. Description of the HOL system and logic can be found in [Gordon85a] and [Gordon85b]. The following account is condensed from those sources.

2.1 Logic

Standard predicate calculus notation is used in HOL.

- "$P(x)$" means "x has property P",
- "$\neg t$" means "not t",
- "$t_1 \wedge t_2$" means "t_1 and t_2",

- "$t_1 \vee t_2$" means "t_1 or t_2",
- "$t_1 \Rightarrow t_2$" means "t_1 implies t_2",
- "$\forall x.\ t[x]$" means "for all x it is the case that $t[x]$",
- "$\exists x.\ t[x]$" means "for some x it is the case that $t[x]$",
- "$\exists! x.\ t[x]$" means "there is a unique x such that $t[x]$".

Here t, t_1 and t_2 stand for arbitrary *terms*, $t[\ ]$ stands for a *context* (a term with a 'hole') and $t[x]$ stands for the term resulting from putting x into the hole in $t[\ ]$.

There are four different kinds of terms: variables, constants, function applications and λ-terms. Variables are sequences of letters or digits beginning with a letter *e.g. variable0*. Constants have the same syntax as variables but stand for fixed values *e.g.* T and F are constants with respect to the theory `BOOL` of truth-values. Function applications have the general form $t_1\ t_2$ where t_1 and t_2 are terms, *e.g.* $P\ 0$. Binary function constants can be declared to be infixed *e.g.* one can write $t_1 + t_2$ instead of $+\ t_1\ t_2$. λ-terms denote functions and have the form $\lambda x.t$ (where x is a variable and t a term) *e.g.* $\lambda n.n+1$ denotes the successor function. The quantifiers $\forall$ and $\exists$ are polymorphic constants, $\forall$:$(\alpha \rightarrow bool) \rightarrow bool$ and $\exists$:$(\alpha \rightarrow bool) \rightarrow bool$ respectively. They are declared as binders which allows the usual syntax $\forall x.\ t$ to be used instead of $\forall(\lambda x.t)$.

Types are expressions that denote sets of values, they can be either *atomic* (*e.g.* *bool*) or *compound* (*e.g.* $\sigma_1 \rightarrow \sigma_2$).

2.2 Specification and Verification of an Example Device

2.2.1 Behavioural specification of a device

We present the specification in HOL of the behaviour of the device called `MEMORY`, Figure 1.

The behaviour of the memory element is specified by:

```
MEMORY(load,in,out) = (∀t. out(t+1) = (load t ⇒ in t | out t))
```

(`a ⇒ b | c` can be read *if* a *then* b *else* c.)

Signals are modelled in HOL by functions from time to boolean values. `num` is the type of natural numbers in HOL and is used to represent time instants. Each of `load,in,out` are functions of type `num` $\longrightarrow$ `bool`. At a particular time `t0`, `load t0` returns the boolean value on the corresponding physical line at that time. (We

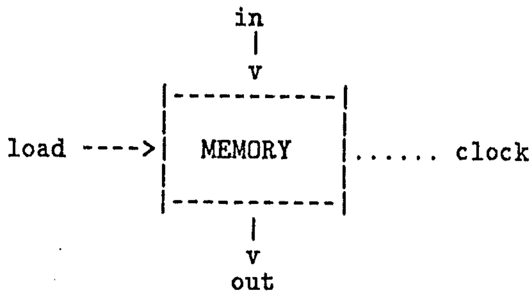


Figure 1: MEMORY device

may think of the time steps as being defined by an implicit synchronous clock and the signals we model as the real signals sampled at these time steps.)

MEMORY is a predicate which takes the three signal functions load,in,out as arguments. The left hand side of the specification, MEMORY(load,in,out), can be true or false. It is true if and only if the relationship on the right hand side of the specification is true. The right hand side

$$(\forall t.\ out(t+1) = (load\ t \Rightarrow in\ t \mid out\ t))$$

describes a relationship which must hold between the values on lines over time if the predicate MEMORY(load,in,out) is to be true.
The behaviour specified is:

> if load is high when the device is clocked then the next value on out is the present value of in;
> if load is not high then the next value on out is the same as its present value.

We now implement the memory device using a structure of basic components.

2.2.2 Components

The two components used to implement the memory device are the multiplexor and register, Figure2.

The predicate, MUX, which specifies the behaviour of the multiplexor is:

$$MUX(switch,i1,i2) = (\forall t.\ o\ t = ((switch\ t) \Rightarrow (i1\ t)|(i2\ t))$$

The predicate REG is specified by:

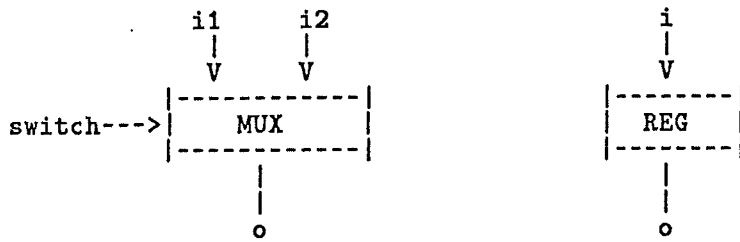


Figure 2: Components

```
REG(i,o) = (∀t. o(t+1) = i t)
```

2.2.3 Representation of structure in HOL

A structure consisting of a multiplexor and register is shown in Figure3. This circuit implements the memory device.

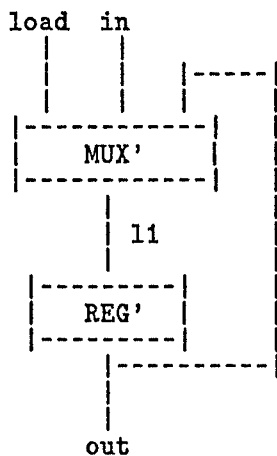


Figure 3: Structure

Both of the components, `MUX'` and `REG'`, place constraints on the allowed sequences of values on their inputs and outputs. For example, the register's input at any time equals its output at the next time step. The structure of devices imposes simultaneously the constraints of each device and this can be represented in logic

by the conjunction of the predicates which specify the behaviour of each device. The combined behaviour is thus described by:

```
MUX(load, in, out, l1) ∧  REG(l1,out)
```

(Explicit renaming is not necessary. For example, having defined MUX we can construct any term MUX(l0,l1,l2,l3) provided l0,l1,l2 and l3 are of the correct type.)

When a structure is created it is common to make certain lines internal to the composite device. For the above structure, we would like to make l1 an internal line and eliminate it from the description of external device behaviour. We say that the external behaviour of a device is consistent with the behaviour of the internal devices if there exist waveforms for the internal lines which permit the constraints of the internal devices to be fulfilled. In our logical representation this means existentially quantifying the functions representing internal signals.

We can represent the external behaviour of the composite device, where line l1 is internal, using the predicate MEM_STRUCT defined as:

```
MEM_STRUCT(load,in,out) = (∃ l1. MUX(load, in, out, l1) ∧  REG(l1,out))
```

2.2.4 Deduction of implementation behaviour

For a structure, such as that described above, we can apply rules of inference to deduce the overall behaviour. It is trivial to deduce that:

```
MEM_STRUCT(load,in,out) =  (∀ t. out(t+1) = (load t ⇒ in t | out t))
```

This behaviour must then be compared to the specified behaviour of the device.

2.2.5 Proof of correctness

The specification of the memory device was:

```
MEMORY(load,in,out) =  (∀ t. out(t+1) = (load t ⇒ in t | out t))
```

It is easy to see that the specified behaviour and the behaviour of the implementation are equivalent.We can prove that:

```
MEM_STRUCT(load,in,out)  =  MEMORY(load,in,out)
```

(In section4.2.3 a simple proof of correctness is described more fully.)

In less trivial digital designs, we do not prove the exact equivalence of the specification and implementation behaviours. Instead we prove correct a partial specification of the digital design. In general, the correctness theorems proved are of the form:

```
IMPLEMENTATION_BEHAVIOUR  ⟹  SPECIFIED_BEHAVIOUR
```

For the memory device we might form a partial specification consisting of:

```
PARTIAL_MEM(load,in,out) =  (∀t. (¬(load t)) ⟹ (out(t+1) =  out t))
```

(This partial specification states that the device retains its value when the load line is low.)
We cannot prove that this specification and the implementation behaviour are equivalent, but we can deduce that:

```
MEM_STRUCT(load,in,out) ⟹ PARTIAL_MEM(load,in,out)
```

This result can be read as stating that whenever the predicate `MEM_STRUCT` is true, then `PARTIAL_MEM` is also true.

3 The Cambridge Fast Ring ECL Chip

The ECL chip of the Cambridge Fast Ring is the subject of this case study of formal specification and verification. We now give an informal description of the chip and its environment.

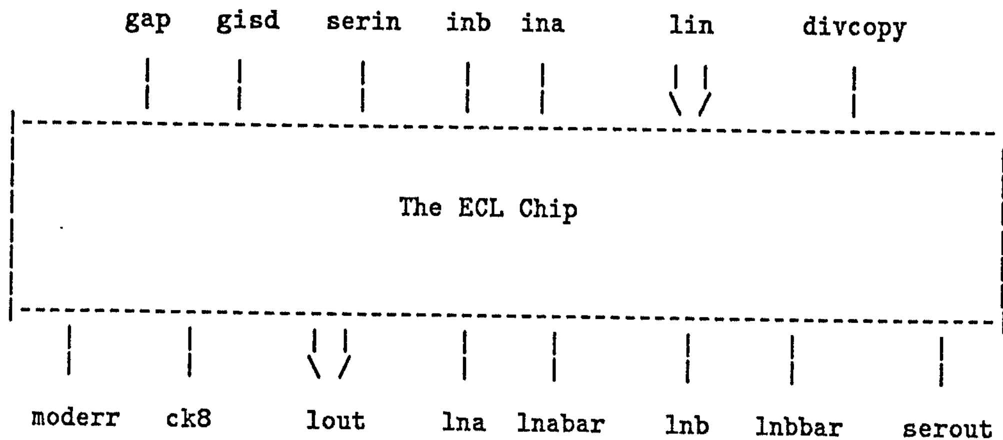
The Cambridge Fast Ring is a system for interconnecting digital devices [Hopper86]. It provides a closed loop communication path on which packets circulate and to which devices can be attached. The network is designed to operate at around 100MHz and is implemented using several chips which can be configured in a number of ways. The main components are a high speed ECL chip, a CMOS chip and a 64k DRAM.

The 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. 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 Cambridge Fast Ring ECL chip was designed in the Computer Laboratory by Dr. Andrew Hopper and has a complexity equivalent to about 360 gates. (We refer to this integrated circuit simply as "the ECL chip".)

4 Methodology in Case Study

Before dealing with the overall specification and implementation of the ECL chip, we present the basic elements of the use of HOL to specify and verify the chip. We firstly characterise the devices that are primitive components for the ECL chip implementation. We than use the demodulator part of the chip as an example to illustrate the techniques of specification and verification used.

4.1 Behaviour of primitive components

The descriptions of primitive component behaviour in HOL are mostly similar to the descriptions in LSM. The major difference is the lack of state in the memory elements in HOL. Another difference is in the more careful treatment of clocking in HOL. In the LSM proof we had to refer to the derived clocks ckl and ckr. In HOL we will be able to relate these to the main clock and in the overall specification of the chip, ckl and ckr will be hidden.

A new type trigger (of which there are two constants ON and OFF) is defined. Derived clock signals can be *on* or *off* at each tick of the main clock and are modelled by functions of type num ⟶ trigger. The predicate MAINCLOCK is true of a clock which is active at every implicit tick. MAINCLOCK is defined simply by:

```
MAINCLOCK(ck) = (∀t. ck t = ON)
```

Primitive devices used in the implementation are inverters, gates and flip-flops. Predicates which specify the behaviour of these elements are as follows:

```
INV(in, out)             = ∀t. out t = ¬(in t);
NOR2(in1, in2, out)      = ∀t. out t = ¬(in1 t ∨ in2 t)
NOR3(in1,in2,in3,out)    = ∀t. out t = ¬(in1 t ∨ (in2 t ∨ in3 t))

DTYPE11(d, q)            = ∀t. q t = (d(t-1))

DTYPE21(d1, d2, q)       = ∀t. (q t = (d1(t-1) ∨ d2(t-1)))

DTYPE21B(d1,d2,qbar)     = ∀t. qbar t = ¬(d1(t-1) ∨ d2(t-1))

DTYPE12(d, q, qbar)      = (∀t. (q t = d(t-1))) ∧ (∀t. (qbar t = ¬(q t)))

DTYPE22(d1,d2,q,qbar)    = (∀t. (q t = (d1(t-1) ∨ d2(t-1)))) ∧
                           (∀t. qbar t = ¬(q t))


CLOCKNOR(g,ckin,ckout) = ∀t. ckout t = (g t ⇒ OFF | ckin t)

DTYPE1C1(d, ck, q)      = ∀t.q t = ((ck(t-1) = ON) ⇒ d(t-1) | q(t-1))

DTYPE1C2(d,ck,q,qbar)  =
              (∀t. (q t   = ((ck(t-1) = ON) ⇒ d(t-1) | q(t-1)))) ∧
               ∀t. qbar t = ¬(q t)

DTYPE22CK(d1,d2,ckout,q,qbar) =
           (∀t. (q t   = (d1(t-1) ∨ d2(t-1)))) ∧
           (∀t. qbar t = ¬(q t)) ∧
           (∀t. (ckout t = ON ) = ((d1(t) ∨ d2(t)) ∧ ¬(q t)))
```

The first eight predicates follow the standard pattern for describing the behaviour of gates and flip-flops. The predicates CLOCKNOR, DTYPE1C1, DTYPE1C2, and DTYPE22CK are different because of the special treatment of clocking. CLOCKNOR describes the behaviour of a NOR gate used to gate a clock rather than compute the *nor* function. If the gating signal g is high then no output clocking event occurs, otherwise the output clock follows the input clock. (Clock skew caused by signal inversion is ignored at this modeling level.)

The devices described by predicates DTYPE1C1 and DTYPE1C2 have an explicit clock input because they are not clocked by the main clock. If the clock is active then new output value(s) are computed, otherwise the previous output values persist. The predicate DTYPE22CK describes a two input D flip-flop that has a clock output as well as the usual q and qbar. The clock output is not a real extra output but is used to model edges on the q output line. The output q carries boolean values, whereas the output ckout carries trigger values. The clock is active when a positive edge occurs. The immediate nature of the edge is modelled by testing that the "present" value of q is F and the "next" is T.

4.2 Specification and Verification of the Demodulator

Before presenting the specification and verification in HOL of the ECL chip, we use the demodulator module as a detailed example of the techniques used. In describing some simple concepts used in the specification of the demodulator we will refer to the natural language description of the ECL chip in section 3. This was written as a general informal description of the chip behaviour prior to the HOL case study.

4.2.1 Specification of demodulator

The specification of the demodulator is presented in terms of formalising the English description of the behaviour given earlier.
We said previously:

> .. boolean values (denoted by T and F) are encoded by the changes on the lines at successive clock ticks.

We define the predicate CHANGED to state precisely what is meant by a change on a line:

```
CHANGED ina t = ¬(ina t = ina(t-1))
```

Demodulated and modulation error were described thus:

> 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).

Although this describes a valid form of demodulation, it does not accurately describe the demodulation procedure used in this design. The appropriate definitions of predicates DEMODULATE and DEMODULATE_ERR are:

```
DEMODULATE(ina,inb)t     = CHANGED ina t ∧ CHANGED inb t
DEMODULATE_ERR(ina,inb)t = ¬ CHANGED ina t ∧ ¬ CHANGED inb t
```

From these definitions we see that a change on both lines corresponds to T, a change on one line *or* neither line changing corresponds to F. Neither line changing is a modulation error. The error is signalled to the external logic but the internal logic receives the value F as if a valid piece of data had arrived. The error in our English description seems trivial but trivial misunderstandings can cause major problems. Formulating our concepts in logic enforces precise descriptions and subsequent formal verification detects any inconsistencies.

We specify the demodulator by describing the desired relationship between its input and output signals. We define a predicate DEMOD_SPEC:

```
DEMOD_SPEC(ina,inb,gisd,serin,moderr,data,exringdata) =
  (∀t. moderr t = DEMODULATE_ERR(ina,inb)(t - 2)) ∧
  (∀t. data t = exringdata(t - 1)) ∧
  (∀t.
    exringdata t =
    (gisd t ⇒ serin(t - 1) | DEMODULATE(ina,inb)(t - 1)))
```

exringdata is the data received from the ring. If the input gisd (gate in serial data) is high then exringdata is the value on the input pin serin at the previous tick, otherwise it is the value of the demodulated ina and inb signals at the previous clock tick. The output pin moderr indicates a modulation error on inputs ina and inb two clock ticks previously. The output pin data is the value of exringdata at the previous clock tick.

Notice how precise we must be about the time dimension in the specification. The function of the moderr pin was described as (cf. section 3):

> moderr is asserted if a modulation error has been detected in the modulated data received from the ring.

The formal specification states that `moderr` is asserted two clock cycles after the modulation error has occurred.

The specification of the demodulator does not mention any internal state; the need for storage devices is implicit in our references to "previous" values of signals. For this example specifying behaviour by the relationships between sequences of values on the inputs and outputs seems to be clearer than using state. A previous state-based specification [Gordon86] described seven internal state variables. The state values are derived from the inputs and other state values; the outputs are derived in turn from state values. The states encumber the specification by obscuring the input-output relationship.

4.2.2 Demodulator implementation

The implementation of the demodulator is depicted in the circuit diagrams in Appendix 1. A repeated sub-structure of the implementation is described by the predicate DEMOD_SLICE:

```
DEMOD_SLICE(ina,q0,q0b) =
  (∃ l0 l1 l2 l3 l4.
    INV(ina,l0) ∧
    DTYPE12(l0,l1,l2) ∧
    NOR2(ina,l1,l3) ∧
    NOR2(l0,l2,l4) ∧
    DTYPE22(l3,l4,q0,q0b))
```

The signals which are existentially qualified on the right are those local to the sub-block. Using the definitions for the gates and flip-flops and applying inference rules of HOL we can deduce:

```
DEMOD_SLICE(ina,q0,q0b) =
 (∀t. q0 t  = CHANGED ina (t-1) ∧
 (∀t. q0b t = ¬ (CHANGED ina (t-1))
```

The total structure of the demodulator is described by:

```
DEMOD_IMP(ina,inb,gisd,serin,moderr,di0,di1,data) =
  (∃ s0 s1 s2 s3 qa qab qb qbb.
    DEMOD_SLICE(ina,qa,qab) ∧
    DEMOD_SLICE(inb,qb,qbb) ∧
    NOR2(qa,qb,s3) ∧
    DTYPE11(s3,moderr) ∧
    NOR3(qab,qbb,gisd,di0) ∧
    INV(gisd,s2) ∧
    INV(serin,s0) ∧
    DTYPE11(s0,s1) ∧
    NOR2(s1,s2,di1) ∧
    DTYPE21(di0,di1,data))
```

The above implementation has outputs di0 and di1 and no output exringdata. At any time only one of di0 and di1 is active and in our specification exringdata corresponds to whatever signal is active. At a given time exringdata is the disjunction of the values of di0 and di1 at that time. Adding the condition

```
(∀t. exringdata t = di0 t ∨ di1 t)
```

allows us to relate the implementation and specification.

One can think of exringdata as a "virtual" signal not carried by any physical line but useful in the description of behaviour.

4.2.3 Proof of correctness

The following procedure is used to verify the demodulator.

- The predicates describing the implementation components (*e.g.* INV) and the higher-order functions in the specification (*e.g.* DEMODULATE) are expanded.
- The existential quantifiers in the implementation are eliminated.
- The conditional on gisd t in the specification is related to boolean expressions involving gisd t in the implementation by:
  - Generating a case split on gisd t.
  - Simplification using basic axioms and theorems of booleans. For example, F ⇒ a | b simplifies to b; ¬ (T ∨ c) simplifies to F.
- Rewriting, involving manipulation of boolean expressions, is used to complete the proof of correctness.

The statement of correctness is as follows:

```
DEMOD_IMP(ina,inb,gisd,serin,moderr,di0,di1,data) ∧
(∀t. exringdata t = di0 t ∨ di1 t) ⟹
 DEMOD_SPEC(ina,inb,gisd,serin,moderr,data,exringdata)
```

This can be read as:
If the structural predicate describing the demodulator DEMOD_IMP holds, and the relationship between exringdata and di0, di1 is true, then the desired behaviour specified by DEMOD_SPEC also holds.

5 Implementation of the ECL Chip

The total structure of the ECL chip can be described in a manner similar to that described for the module DEMOD. A block diagram of the ECL chip and circuit diagrams of individual modules are given in Appendix 1.

5.1 Description of Implementation

The higher-order function describing the implementation, ECL_IMPLEMENTATION, takes as arguments a tuple of input and output signals and a tuple of internal signals. This serves to distinguish the external signals from the internal signals which correspond to internal state.

ECL_IMPLEMENTATION is defined as follows:

```
ECL_IMPLEMENTATION
    (ina,inb,gisd,serin,moderr,di0,di1,d4,gap,ck8,ck,lin,lout,
     divcopy,dataout,lna,lnabar,lnb,lnbbar,serout)
    (h0,h1,gapendbar,rc0,rc1,rc2,rc3) =
    (∃data d2l_d2r reset blk p0 right left ckl ckr qtop qbot.
      DEMOD_IMP(ina,inb,gisd,serin,moderr,di0,di1,data) ∧
      SHIFT4(data,d4,d2l_d2r) ∧
      HOH1(reset,h0,h1,blk) ∧
      DETGAP_IMP(di0,di1,gap,d2l_d2r,reset,gapendbar,blk) ∧
      RINGCOUNTER_IMP(reset,gapendbar,p0,ck8,rc0,rc1,rc2,rc3) ∧
      CLOCKS_IMP(p0,ck8,ck,right,left,ckl,ckr) ∧
      DATAIN(d4,qtop 0,qbot 0) ∧
      LISTAND 7(λp. SLICE_CK p(lin,ckl,qtop,right,left)) ∧
      LISTAND 7(λp. SLICE_CK p(lin,ckr,qbot,left,right)) ∧
      LISTAND 7(λp. SLICE_OUT1 p(qtop,qbot,right,left,lout)) ∧
      DATAOUT(divcopy,di0,di1,left,right,qtop 8,qbot 8,dataout) ∧
      MODUL_IMP(dataout,lna,lnabar,lnb,lnbbar,serout) ∧
      MAINCLOCK ck)
```

The predicate MAINCLOCK has been described in section 4.1. The term MAINCLOCK ck has been included in the structural description to assert that ck is the main clock. The predicates SHIFT4, DETGAP_IMP, HOH1, RINGCOUNTER_IMP, CLOCKS_IMP, DATAOUT, DATAIN and MODUL_IMP describe the implementation of modules in the same manner as DEMOD_IMP.

A new higher-order function LISTAND is used to describe repeated structure in the module called SHIFTREGS.

```
LISTAND 7(λp. SLICE_CK p(lin,ckl,qtop,right,left))
```

describes a structure consisting of 8 repeated SLICE_CK modules.

LISTAND is defined as follows:

```
LISTAND 0 p       = p 0 ∧
LISTAND (SUC n) p = LISTAND n p ∧ (p(SUC n))
```

Given an expression p which takes a single number as argument, the function LISTAND applied to n gives a predicate which corresponds to the conjunction of p applied to all numbers from 0 to n inclusive. *i.e.*

```
LISTAND x p  = p 0 ∧ p 1 ∧ .... p(x-1) ∧ p x
```

(A similar higher-order function LISTOR was also defined and used in the proof, although it does not appear in the overall specifications.)

The repeated structure in the SHIFTREGS module motivates the use of LISTAND and also the description of certain groups of signals as *busses*. A *bus* is modelled by a function of type num ⟶ num ⟶ bool. lin models the parallel input lines; lin 0 corresponds to the signal at the low end of the shift register, lin 1 the next signal *etc.*. SLICE_CK is defined in such a way that the nth SLICE_CK is a predicate which describes the structure inter-connecting the nth signals on the busses.

The use of LISTAND to facilitate a proof of the behaviour of parameterised shift registers is presented in a later section.

6 Specification and Verification of the ECL Chip

In conducting a case study of specification and verification we would like ideally to give an overall specification of the desired behaviour of the design and then proceed to build and verify a particular implementation. In reality formal specification and verification often follows after the design of the hardware. This is true of this case study but we will follow the more usual route of presenting the overall specification first and then the implementation and proof of correctness.

6.1 Top-level Specification

Figure 4 presents the view of the ECL chip contained in the specification. The only state variables which are in the specification are h0, h1, gapendbar, rc0, rc1, rc2, and rc3. The only internal signals which are part of the specification are exringdata, d4 and dataout. d4 is just exringdata delayed by a number of cycles and so could be eliminated easily. We specify the transfer of data from the ring by referring to d4 because that is the input signal for the pair of shift registers used for data transfer. We can describe the byte boundaries easily by referring to d4.

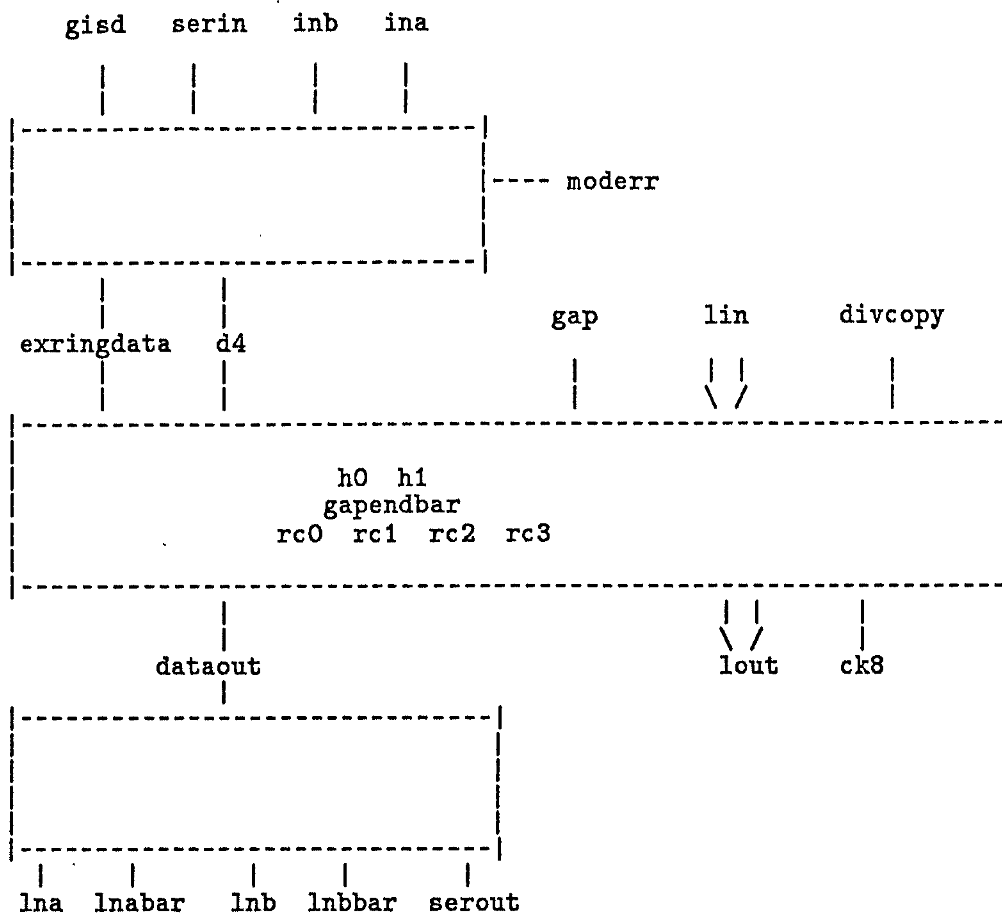


Figure 4: Conceptual view of ECL specification

General purpose higher-order functions used in the specification are SERIAL_BYTE, PAR_BYTE and NEXT_AFTER.

These are defined as follows:

```
SERIAL_BYTE sig t0 =
  mk_bool8
  (sig(t0 + 7),sig(t0 + 6),sig(t0 + 5),sig(t0 + 4),sig(t0 + 3),
   sig(t0 + 2),sig(t0 + 1),sig t0)

PAR_BYTE sig_bus t0 =
  mk_bool8
  (sig_bus 0 t0,sig_bus 1 t0,sig_bus 2 t0,sig_bus 3 t0,sig_bus 4 t0,
   sig_bus 5 t0,sig_bus 6 t0,sig_bus 7 t0)

NEXT_AFTER(t1,t2)f =
      t1 < t2 ∧ (∀t. t1 < t ∧ t < t2 ⟹ ¬f t) ∧ f t2
```

SERIAL_BYTE applied to a signal and time t0 gives an object of type bool8 which is composed of the signal values at times t0 up to t0 + 7.

PAR_BYTE applied to a bus and time t0 gives an object of type bool8 which is composed of the values of the eight lowest elements of the bus at time t0.

NEXT_AFTER applied to a pair of numbers (t1,t2) gives a predicate which is true of its argument f if t2 is the next time after t1 that f t is true.

The definition of the predicate ECL_SPECIFICATION which specifies the ECL chip behaviour is presented in Figure 5. We have named the sections of the specification to facilitate explanation. We will now go through the specification sections explaining the special purpose functions used, saying what behaviour each section specifies and comparing this with the natural language description in section 3.

6.2 Specification in Detail

DEMODULATE section:

The specification of the demodulator has been dealt with in detail in section 4.2. The signal exringdata is the serial or demodulated data received from the communications ring.

JUST_COPY section:

This specifies the simple requirement that when divcopy is low the input data is copied to the output.

OUT_OF_GAP section:

The special functions used in this section are defined thus:

```
div_ASSERT(divcopy,t0,del) =
      (∀t. (t0 + 1) ≤ t ∧ t ≤ ((t0 + del) + 16) ⟹ divcopy t)
```

div_ASSERT is true if divcopy is asserted over the interval t0 + 1 to ((t0 + 16) + del) inclusive.

```
OUT_OF_GAP(gap,gapendbar)t0 =
  (∀t. t0 ≤ t ∧ t ≤ (t0 + 7) ⟹ ¬gap(t - 1)) ∧ gapendbar t0
```

OUT_OF_GAP is true of signals gap, gapendbar and a time t0 if gap is low from t0 - 1 until t0 + 6 and gapendbar is high at time t0. OUT_OF_GAP holds when the chip is transmitting data as normal.

```
NOT_GAP_END(exringdata,gapendbar)t0 =
  (∀t. t0 ≤ t ∧ t ≤ (t0 + 7) ⟹ ¬exringdata t) ∧ gapendbar t0
```

NOT_GAP_END is true of signals exringdata, gapendbar and a time t0 if exringdata is low from t0 until t0 + 7 and gapendbar is high at time t0.

```
ECL_SPECIFICATION
       (ina,inb,gisd,serin,moderr,exringdata,d4,gap,ck8,ck,lin,lout,
        divcopy,dataout,lna,lnabar,lnb,lnbbar,serout)
       (h0,h1,gapendbar,rc0,rc1,rc2,rc3) =

% ****            DEMODULATE            **** %
       ((∀t. moderr t = DEMODULATE_ERR(ina,inb)(t - 2)) ∧
        (∀t.
          exringdata t =
          (gisd t ⇒ serin(t - 1) | DEMODULATE(ina,inb)(t - 1))) ∧
        (∀t. d4 t = exringdata(t - 5)) ∧

% ****            JUST_COPY             **** %
        (∀t. ¬divcopy t ⟹ (dataout t = exringdata t))) ∧

% ****            OUT_OF_GAP            **** %
       (∀t0.
         div_ASSERT(divcopy,t0,0) ∧
         (OUT_OF_GAP(gap,gapendbar)t0 ∨
          NOT_GAP_END(exringdata,gapendbar)t0) ∧
         CK8_STATE(rc0,rc1,rc2,rc3)t0 ∧
         (OUT_OF_GAP(gap,gapendbar)(t0 + 8) ∨
          NOT_GAP_END(exringdata,gapendbar)(t0 + 8)) ⟹
         (SERIAL_BYTE dataout(t0 + 1) = PAR_BYTE lin t0) ∧
         (SERIAL_BYTE dataout(t0 + 9) = PAR_BYTE lin(t0 + 8)) ∧
         (∀t.
           t ≤ 6 ⟹
           (PAR_BYTE lout((t0 + 9) + t) = SERIAL_BYTE d4(t0 + 1))) ∧
         (ck8 t0 = ON) ∧
         NEXT_AFTER(t0,t0 + 8)(λt. ck8 t = ON) ∧
         NEXT_AFTER(t0 + 8,t0 + 16)(λt. ck8 t = ON)) ∧

% ****            AT_GAP_END            **** %
       (∀t0 del.
         div_ASSERT(divcopy,t0,del) ∧
         AT_GAP_END(exringdata,gap,gapendbar,h0,h1,del)t0 ∧
         CK8_STATE(rc0,rc1,rc2,rc3)t0 ∧
         AFTER_GAP gap((t0 + 8) + del) ⟹
         (SERIAL_BYTE dataout(t0 + 1) = PAR_BYTE lin t0) ∧
         (SERIAL_BYTE dataout((t0 + 9) + del) =
          PAR_BYTE lin((t0 + 8) + del)) ∧
         (∀t.
           t ≤ 6 ⟹
           (PAR_BYTE lout(((t0 + 9) + del) + t) =
            SERIAL_BYTE d4((t0 + 1) + del))) ∧
         (ck8 t0 = ON) ∧
         NEXT_AFTER(t0,(t0 + 8) + del)(λt. ck8 t = ON) ∧
         NEXT_AFTER((t0 + 8) + del,(t0 + 16) + del)(λt. ck8 t = ON)) ∧

% ****          MODULATE              **** %
       MODULATE_REL(dataout,lna,lnb) ∧
       (∀t. lnabar t = ¬lna t) ∧
       (∀t. lnbbar t = ¬lnb t) ∧
       (∀t. serout t = dataout(t - 1))
```

Figure 5: HOL Specification of the ECL chip behaviour

```
CK8_STATE(rc0,rc1,rc2,rc3)t0 =
  (((rc0 t0 = T) ∧ (rc1 t0 = T)) ∧ (rc2 t0 = T)) ∧ (rc3 t0 = F)
```

CK8_STATE is true of signals rc0,rc1,rc2,rc3 and time t0 if the signals have the values T, T, T, F at that time.

The OUT_OF_GAP section specifies how the chip should operate when it is not in the gap between bytes or when it is in the gap but not at the end of the gap. Assuming that divcopy is asserted for a suitable interval, that the OUT_OF_GAP - NOT_GAP_END conditions are satisfied at t0 and (t0+8), and that rc0,rc1,rc2 and rc3 are in the clocking state at t0, then the serial bytes on dataout starting after t0 and after (t0+8) are the parallel bytes presented on lin at t0 and (t0+8) and the serial byte coming in on d4 starting at (t0+1) is presented as a parallel byte on lout for 7 ticks starting at (t0+9) and the next times ck8 is active after t0 are (t0+8) and (t0+16).

The English description (section 3) contained the following:

> 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 clock at the byte frequency is produced by the chip.

Our formal specification reflects the serial to parallel and parallel to serial behaviour mentioned above. What is not obvious from the informal description is that we must specify the behaviour over two successive byte cycles. The parallel byte presented to the slower logic is that received from the ring during the previous byte cycle.

The form of the condition for not being at the end of the gap NOT_GAP_END was a surprise. When the hardware is looking for the end of the gap but has not found it (i.e. a high has not been received) data is transferred as normal. In the definition of NOT_GAP_END we have not specified anything about the hardware looking for the end of the gap. What has emerged is that low data coming in is a sufficient condition for the behaviour we wish to deduce irrespective of whether the hardware is looking for the end of gap or not.

AT_GAP_END section:

The functions used in this section are AT_GAP_END and AFTER_GAP.

```
AT_GAP_END(exringdata,gap,gapendbar,h0,h1,del)t0 =
  (∀t.
    ((t0 + del) - 7) ≤ t ∧ t ≤ (t0 + (del - 1)) ⟹ ¬exringdata t) ∧
  exringdata(t0 + del) ∧
  gap(t0 + (del - 1)) ∧
  gapendbar t0 ∧
  h0 t0 ∧
  h1 t0 ∧
  del ≤ 7
```

AT_GAP_END describes the conditions on signals exringdata,gap,gapendbar,h0,h1 that the end of gap is reached del ticks after t0, the time of the byte boundary. exringdata must have been low from time ((t0+del)-7) to time (t0+(del-1)) and must go high at (t0+del). exringdata going high indicates the end of the gap. gapendbar,h0 and h1 must be high at time t0. The input signal gap must be high at time t0+(del-1) to require the chip to look for the end of gap.

```
AFTER_GAP gap t0 = (∀t. t0 ≤ t ∧ t ≤ (t0 + 7) ⟹ ¬gap(t - 1))
```

AFTER_GAP defines the condition that after the end of gap the signal gap should be low. This means that the chip is not required to look for the end of a gap.

The AT_GAP_END section specifies how the chip should operate when it is at the end of the gap between bytes. The gap is assumed to end del units after a byte boundary. Assuming that divcopy is asserted and the AT_GAP_END conditions are satisfied at t0 and the AFTER_GAP conditions at ((t0+8)+del), and that (rc0,rc1,rc2,rc3) at t0 is in the state for clocking, then the output serial bytes starting after t0 and after (t0+(9+del)) are the parallel bytes presented on lin at t0 and (t0+(8+del)), and the serial byte coming in on d4 starting at (t0+(del+1)) is presented as a parallel byte on lout for 7 ticks starting at (t0+(9+del)), and the next times ck8 is active after t0 are (t0+8)+del and (t0+16)+del.

The English description (section 3) contained the following:

> 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 clock at the byte frequency is produced by the chip. At the end of a gap this clock must be reset.

The informal description again is imprecise about the transfer of data and does not indicate that the clock is in fact stretched by a number of units equal to the extra number of bits in the long "byte" at the end of the gap.

MODULATE section:

A special predicate describing modulation is defined as:

```
MODULATE_REL(d,lin_1,lin_2) =
   (∃ phase.
     (∀ t.
       (d(t-1) ⇒
        (CHANGED lin_1 t ∧ CHANGED lin_2 t) |
        (phase(t-1) ⇒
         (CHANGED lin_1 t ∧ ¬ CHANGED lin_2 t) |
         (¬ CHANGED lin_1 t ∧ CHANGED lin_2 t)))) ∧
     (∀ t. phase t = ¬ phase(t-1)))
```

MODULATE_REL describes how the outputs lin_1 and lin_2 are produced from the data value on d. In modulating a signal we want to encode high and low values by changes on the outputs and we also want to balance the changes on the lines. If d is high at time t-1 then lin_1 and lin_2 must both change at time t, otherwise if phase is high then only lin_1 is changed and if phase is low only lin_2 is changed. At each clock tick phase is inverted. phase is hidden by existentially quantifying it on the right hand side of the definition. We do not care what value the signal phase has, but just require that it alternates the changes in the manner described.

The MODULATE section specifies how the output data dataout is modulated to give lna and lnb. lnabar and lnbbar present the inverse of lna and lnb respectively, and serout outputs the value of dataout at the previous tick.

6.3 Specification Summary

We have presented the HOL specification of the ECL chip in detail so that the reader can appreciate the relationship between the formal description and the informal understanding of the behaviour. HOL allows us to construct a precise specification of complex behaviour. By using suitable higher-order functions and by only describing the partial behaviour that interests us, we construct a formal specification that matches closely a precise informal description.

6.4 Verification of the ECL Chip

To verify that the structure does implement the specified behaviour, we must add an assertion about the relationship between exringdata and di0, di1.
This is:

```
(∀ t. exringdata t = di0 t ∨ di1 t)
```

(*i.e.* the "timewise" disjunction of signals di0 and di1 is exringdata)
The following correctness theorem has been proven.

```
ECL_IMPLEMENTATION
 (ina,inb,gisd,serin,moderr,di0,di1,d4,gap,ck8,ck,lin,lout,
  divcopy,dataout,lna,lnabar,lnb,lnbbar,serout)
 (h0,h1,gapendbar,rc0,rc1,rc2,rc3) ∧
 (∀t. exringdata t = di0 t ∨ di1 t) ⟹
 ECL_SPECIFICATION
 (ina,inb,gisd,serin,moderr,exringdata,d4,gap,ck8,ck,lin,lout,
  divcopy,dataout,lna,lnabar,lnb,lnbbar,serout)
 (h0,h1,gapendbar,rc0,rc1,rc2,rc3)
```

We have proved that the implementation does achieve the specified behaviour. The specification we have verified is a partial one and should be considered closely. Given certain conditions on the state of the chip and its external signals we specify certain desired behaviour. We have not presented theorems stating safety or liveness properties of the chip. Informally, a *safety property* states that the device does not do something bad; a *liveness property* states that the device does do something good. For example, we have not shown that the chip does not get into an unwanted state (safety) or that it does enter a desired state (liveness).

We have deduced a number of liveness properties but have not included them in the specification for the sake of clarity. For example, we have deduced the new state of the chip after the OUT_OF_GAP and AT_GAP_END behaviours are exhibited. This allows us to verify certain conditions on the internal states.

Formulating and verifying a sufficient set of liveness and safety properties is a difficult task. We have verified the most important behaviour of the ECL chip and some of its safety properties. We have not verified sufficient safety and liveness properties. We should verify that the chip gets into a desired state. What happens if the chip powers up in a strange state ? The start-up state depends on a global reset that we have not modelled, and therefore we have not deduced any start-up behaviour.

7 Features of the Specification

7.1 HOL as a hardware description language

The HOL specification corresponds to a precise formulation of the natural language description of behaviour. The expressiveness of HOL allows this clearer style of description. Higher-order functions can be defined to describe "meaningful"

concepts so that the HOL specification can be easily read and understood by comparison with the natural language description.

7.2 State in the Specification

A notable feature of the HOL specification is the almost total lack of state. There are 7 state variables h0,h1,gapendbar,rc0,rc1,rc2 and rc3 which correspond to 7 flip-flops in the implementation. In HOL one can specify behaviour directly in terms of the input-output relationships and avoid much state. In contrast, the state-based specification in the previous case study [Gordon86] had state variables corresponding to all 39 flip-flops in that implementation. States can obscure the specification by forcing one to specify how states are derived from input and other states, and how the outputs relate to the states and inputs. For example, it is not clear from a state-based description that the chip can transform parallel data bytes to serial bytes.

Specifications in HOL are more abstract than logical state machine specifications because many more (non-trivial) implementations can be shown to satisfy the HOL specification than satisfy the equivalent state-based specification.

7.3 Partial Specification

HOL offers the advantage of partial specification. We can specify the part of the device behaviour that interests us. This is quite different to the LSM specification which is a description of the total behaviour of the device.

A partial specification is more appropriate for the ECL chip. The chip is used in an environment consisting of another communications chip and connections to a local area network. The desired chip behaviour is dictated by the requirements of this environment and is, in part, generated by suitable signals from the environment. The partial specification reflects this understanding of the chip environment.

The influence of partial specification on verification is discussed in section 8.5

7.4 Clock Cycles

A major feature of the HOL specification is that it describes behaviour over multiple clock cycles. One part of the specification (the OUT_OF_GAP section) specifies the behaviour over two successive bytes - a total of 16 clock cycles. Another section (AT_GAP_END) specifies the behaviour over an interval which can be anything from 16 to 23 clock cycles long, depending on where the end of gap occurs. Particularly

useful is the ability to describe the behaviour at the byte level. We can describe operations on data bytes *e.g.* a byte is transformed from serial to parallel.

Many hardware description languages deal mainly with behaviour at the register transfer level and describe what happens on successive clock cycles. The description of compound behaviour over a number of clock cycles is difficult. For certain chips (for example, communication circuits like the ECL chip), behaviour above the register transfer level should be described since this presents a more accurate overall view of what is happening. In HOL we can easily devise specifications which describe compound behaviour over multiple clock cycles and the ECL chip has been specified in this manner.

We call behaviour of digital systems over multiple clock cycles *super register transfer* behaviour.

7.5 Complexity of Behaviour

The complexity of behaviour of the ECL chip as specified in HOL is not apparent from the size of the chip. We believe that the number of gates and flip-flops in a design is not an accurate reflection of the behavioural complexity. In the ECL chip the loops in the structure, derived clocks and operation over multiple clock cycles all contribute to the complexity of the behaviour.

The complexity of behaviour over a number of clock cycles is not usually quantified. One could regard the behavioural complexity for n gates over m cycles to be of order $n \times m$. Using this metric, the ECL chip over 16 cycles, has similar complexity to a 6,000 gate chip over a single cycle.

7.6 Modelling Difficulties

A number of features of the ECL chip do not map easily into the synchronous models used in HOL. The generation of a clock signal, `ck8`, using a flip-flop required the introduction of a special flip-flop model. The gating of a clock signal with a boolean signal also had to be treated using a special model. These models were *ad hoc* solutions to model behaviour which did not fit into our framework.

Problems arise because there are a number of races in the logic. The relative delays of combinational elements determines clock skews and the gating of boolean signals with clock signals. Since these delays are not modelled, we must choose a synchronous level behaviour which corresponds to the behaviour caused by the actual relative delays. The behaviour we choose must be accepted without proof because the delays and behaviour at the lower level are not being modelled.

This is an unsatisfactory way of modelling digital behaviour and a number of attempts were needed before we discovered the relative delays which enforced the behaviour intended by the chip designer.

7.7 Interface Specifications

Forming accurate interface specifications at the module level for the ECL chip was found to be difficult. Timing problems, caused by relative delays outside our model, contributed to this (cf. 7.6). However, it can be difficult to form a set of interdependent partial specifications of modules. The problem arises because we use a technique of partial specification throughout the design hierarchy.

In the ECL chip we have a chain of modules with the outputs of one module feeding the next one on the chain. For each module, we deduce that under certain constraints a partial behaviour is generated. When trying to compose modules at the next level in the design we can fail because of under or over-specification of the modules. In under-specification the partial behaviour deduced for one module is not sufficient to verify the constraints on another module. In over-specification the constraints used to deduce a partial behaviour are too restrictive and cannot be met by other modules. Failure to compose modules means that we must re-formulate and verify the module specifications and then re-compose the modules.

Without a precise knowledge of module behaviour and interaction we cannot form an accurate partial specification of behaviour. When dealing with a chain of interdependent modules, under and over-specification has a knock-on effect which leads to an iterative process of specification and verification.

8 Verification

8.1 Proof size

8.2 Non State-Based Behavioural Descriptions

The size of the total source code for the HOL specification and proofs is about 18000 lines. This is almost 9 times greater than the code to verify the state-based specification in the previous case study [Gordon86] . The greater difficulty in doing the proof is directly related to the HOL specification. In HOL we have constructed a more abstract top-level specification; this describes the compound behaviour of the ECL chip over 16 to 23 clock cycles, and hides much of the chip state.

The rigid framework of logical state machines provided a firm basis for verification in the previous case study. The specification and implementation are equivalent if the corresponding output equations and next-state equations are equivalent. This in effect is verifying the behaviour for a single clock cycle. In HOL we do not have states around which to build the proof and we have to verify the compound behaviour over a number of clock cycles.

8.3 Proof Using a Formal Simulator

In doing proofs it is sometimes easier though tedious to get results by doing exhaustive case analysis which corresponds to a simulation of the digital device. A formal simulator capable of symbolic and value simulation in HOL has been built. Given some input values, the simulator uses inference to deduce the values of internal and output signals. A series of input values can be given and output signals can be traced over time. If an output does not have a simple boolean value the simulator presents the simplified expression for the output using the other deduced values. In this way both symbolic and value simulation are integrated cleanly.

The simulator was used to deduce the behaviour of the ringcounter in the HOL proof of correctness.

8.4 Proof of an n-Bit Shift Register

Using the higher-order function LISTAND, defined in section 5.1, we are able to describe the 8-bit wide shift registers used in the ECL chip. In doing the proof of correctness we have deduced the behaviour of n-bit wide shift registers and then specialised the results for 8 bits. The more general theorems allow us to easily deduce the behaviour if the data transfer unit is changed from 8 bits to 16, 32 or any other number of bits.

In the specification of behaviour we parameterise the timing behaviour of the input signals by n and we parameterise the length of the shift registers in the implementation by n. We then use induction to prove that the desired behaviour (also characterised by n) is achieved. The parameterisation in both time and space is a little tricky to formulate but provides a result which is of general use.

A theorem stating the behaviour of an n-bit shift register in the ECL chip is:

```
(ckl t1 = ON) ∧
left t1 ∧
(∀t. right t = ¬ left t) ∧
LISTAND n(λt. ckl((SUC t1) + t) = ON) ∧
LISTAND n(λt. ¬ left((SUC t1) + t)) ∧
```

```
(∀t. right t = ¬left t) ∧
LISTAND n(λp. SLICE_CK p(lin,ckl,q,right,left)) ⟹
  ∀x. x ≤ n ⟹ (q(SUC n)((SUC t1) + x) = lin(n - x)t1)
```

Note that the behaviour of signals `ckl` and `left` is parameterised by n and the structure consists of n `SLICE_CK` register slices. The resulting output behaviour is characterised by n.

To clarify the above theorem of behaviour, we take n to be 7 corresponding to the 8-bit wide shift register. For $n = 7$ we can simplify the above theorem to:

```
(ckl t1 = ON) ∧
left t1 ∧
(∀t. right t = ¬left t) ∧
LISTAND 7(λt. ckl((SUC t1) + t) = ON) ∧
LISTAND 7(λt. ¬left((SUC t1) + t)) ∧
(∀t. right t = ¬left t) ∧
LISTAND 7(λp. SLICE_CK p(lin,ckl,q,right,left)) ⟹
  SERIAL_BYTE q 8 (t1+1)  = PAR_BYTE lin  t1
```

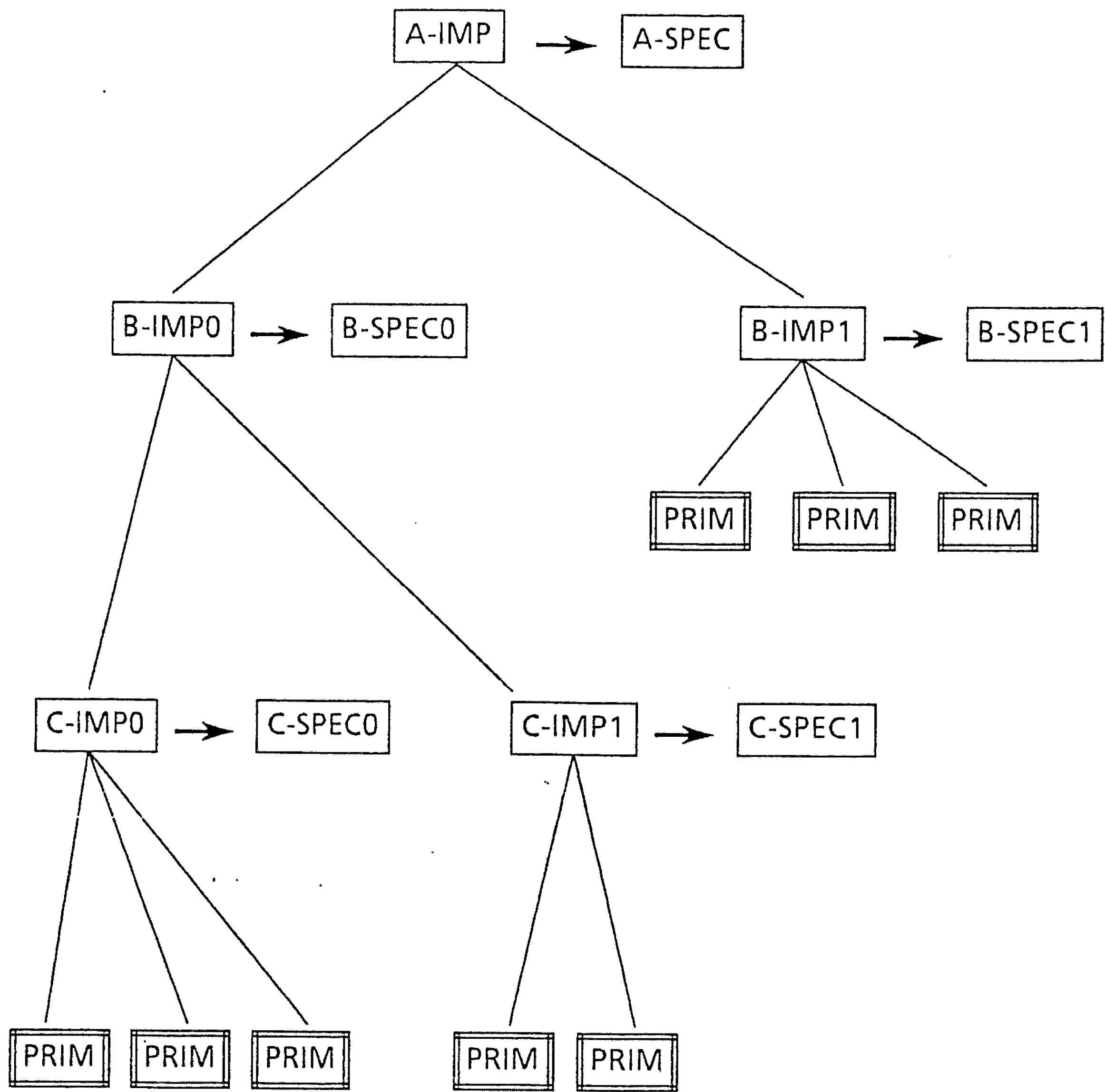
Given a sequence of 8 suitable input signals and a structure of 8 register slices we can deduce that the serial byte appearing at top end of the shift register is the byte that was presented on the parallel inputs at an earlier time.

Higher-order functions facilitate the formation of parameterised specifications. However, the lack of data types parameterised on numbers and the lack (so far) of a full theory of lists means that carrying the parameterised behavioural specifications into the overall specification is difficult.

8.5 Verification of Partial Specifications

We have already claimed that partial specification allows a natural specification of behaviour for the ECL chip. Partial specifications can be used at all levels in the design hierarchy. Figure 6 depicts the design hierarchy indicating the implementations and partial specifications at each level. We assume a bottom-up verification process. This consists of verifying implementations with respect to the specifications for lower level modules, using these specifications rather than the corresponding implementations to verify a specification at the next level and so on.

The use of partial specifications rather than full specifications has an important effect on the verification process. When we combine sub-modules with full specifications we can prove a full specification for the composite module. If we cannot verify the specification of the composite device we know that the implementation



KEY:

*-IMP	Implementation
*-SPEC	Specification
→	satisfies
PRIM	Primitive Component

Figure 6: Verification of Design

is wrong. When using partial specifications, we can fail to verify a compound device because the partial specification of a sub-module is inadequate rather than because the implementation of the sub-module is wrong.

This results in an iterative process of specification and verification which converges on correct partial specifications.

9 Discussion

9.1 Correctness of a Real Design

In the case studies we have specified and verified a real digital design. By *real* we mean that the ECL chip was designed and fabricated for a practical purpose, independent of the case study. The ECL chip was first manufactured in 1984 and has been used since then in the development of the Cambridge Fast Ring. A communications system, incorporating the ECL chip, is now commercially available.

Modelling and verifying a real design brings forth issues not dealt with in contrived examples. It is dangerous, however, to claim that a design has real world correctness. If the mathematical system is sound then a proof of correctness is valid. However, the mathematical system employs models of real devices and their interconnections. The user must accept that the proofs are based on these models and do not verify the behaviour of real devices outside these models. This is no different from other CAD tools, which are all based on models which the user can accept or reject as accurate models.

In the real world the ECL chip sometimes malfunctions. The malfunctions are due to badly behaved signals from its environment. The partial specification in HOL demands certain behaviour from the environment to ensure the desired output behaviour; the behaviour in the presence of badly behaved environment signals is not described.

The malfunctions of the ECL chip do not reflect on the validity of the formal proofs of correctness. In the presence of well behaved signals the chip functions correctly. It is important however that the user appreciates the limitations as well as the power of the formal techniques.

9.2 Methodology

A methodology for using formal techniques of specification and verification in the design process is needed. We have indicated above some of the difficulties found in this case study. Although many problems could be avoided by a restricted

design style, we believe that some basic problems endure. It is difficult to devise an accurate but abstract formal specification of a complex device. It is necessary to refine the formal specification at the higher levels as the specifications at lower levels become fixed.

An enduring methodology for using formal methods can only evolve with more widespread use of these techniques to specify and verify a range of digital designs. A practical CAD system incorporating formal methods is necessary to make these techniques accessible to practitioners in the field.

References

[Camilleri86] A. Camilleri, M. Gordon and T. Melham, "Hardware Verification using Higher-Order Logic", Technical Report No. 91, Computer Laboratory, University of Cambridge, U.K., 1986.

[Gordon85a] M. J. C. Gordon, "HOL A Machine Oriented Formulation of Higher Order Logic", Technical Report No. 68, Computer Laboratory, University of Cambridge, Cambridge, U.K., 1985.

[Gordon85b] M. J. C. Gordon, "Why Higher-Order Logic is a Good Formulism for Specifying and Verifying Hardware", Technical Report No. 77, Computer Laboratory, University of Cambridge, Cambridge, U.K., 1985.

[Gordon86] M. J. C. Gordon and J. M. J Herbert, "Formal hardware verification methodology and its application to a network interface chip", *IEE PROCEEDINGS*, Vol. 133, Pt.E, No. 5, September 1986.

[Hanna83] F. K. Hanna, "Overview of the Veritas Project", Internal Report, University of Kent, U.K., 1983.

[Hopper86] A. Hopper and R. M. Needham, "The Cambridge Fast Ring Networking System", Technical Report No. 90, Computer Laboratory, University of Cambridge, U.K., 1986.

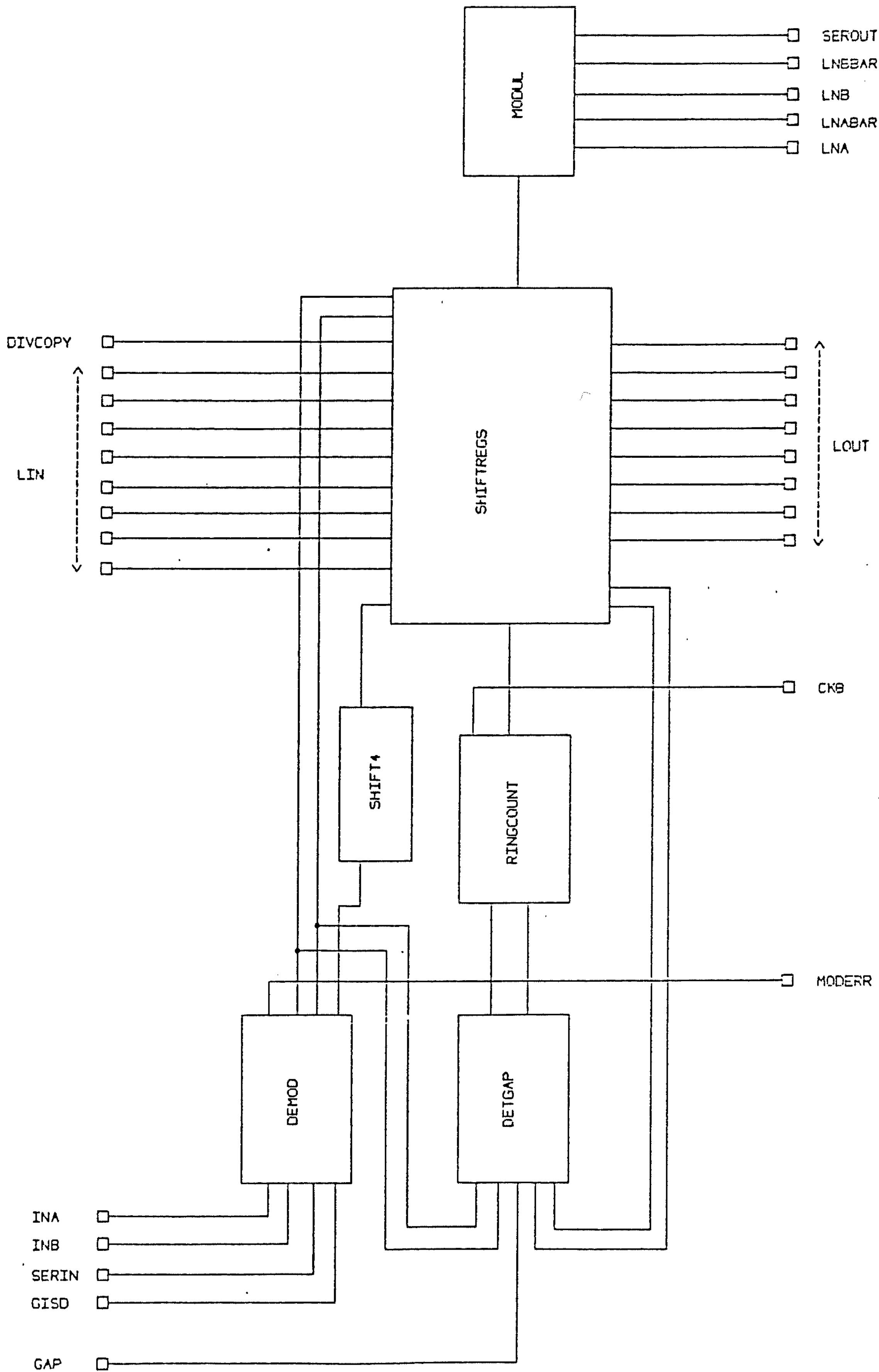
[Hunt85] W. A. Hunt Jr., "FM8501: A Verified Microprocessor", Technical Report 47, University of Texas at Austin, December 1985.

[Moszkowski83] B. C. Moszkowski, "A Temporal Logic for Multi-Level Reasoning about Hardware", *Proceedings of the 6-th International Sympo-*

sium on Computer Hardware Description Languages, North Holland Publishing Co., Pittsburgh, Pennsylvania, May 1983. pp. 79–90.

Appendix 1

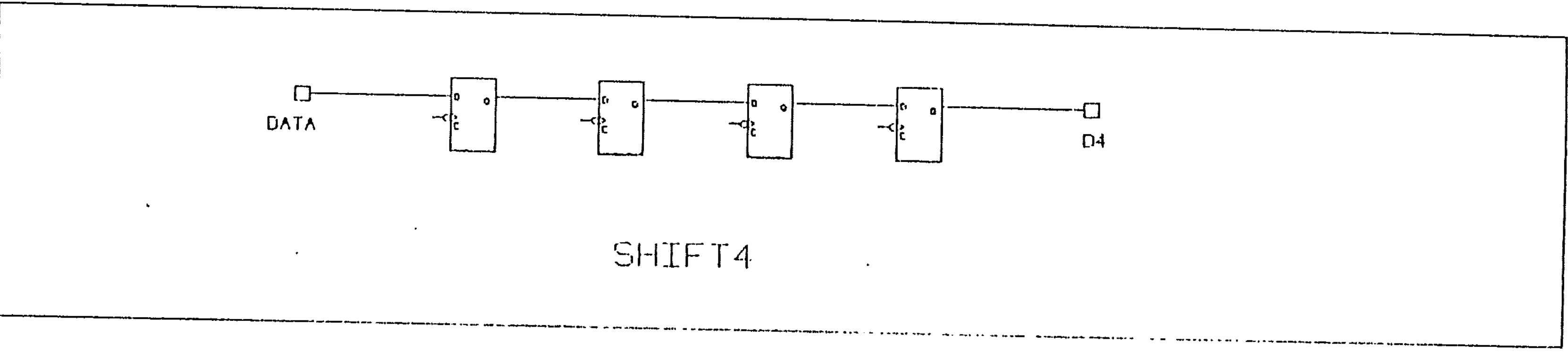
Circuit Diagrams of the Cambridge Fast Ring ECL Chip



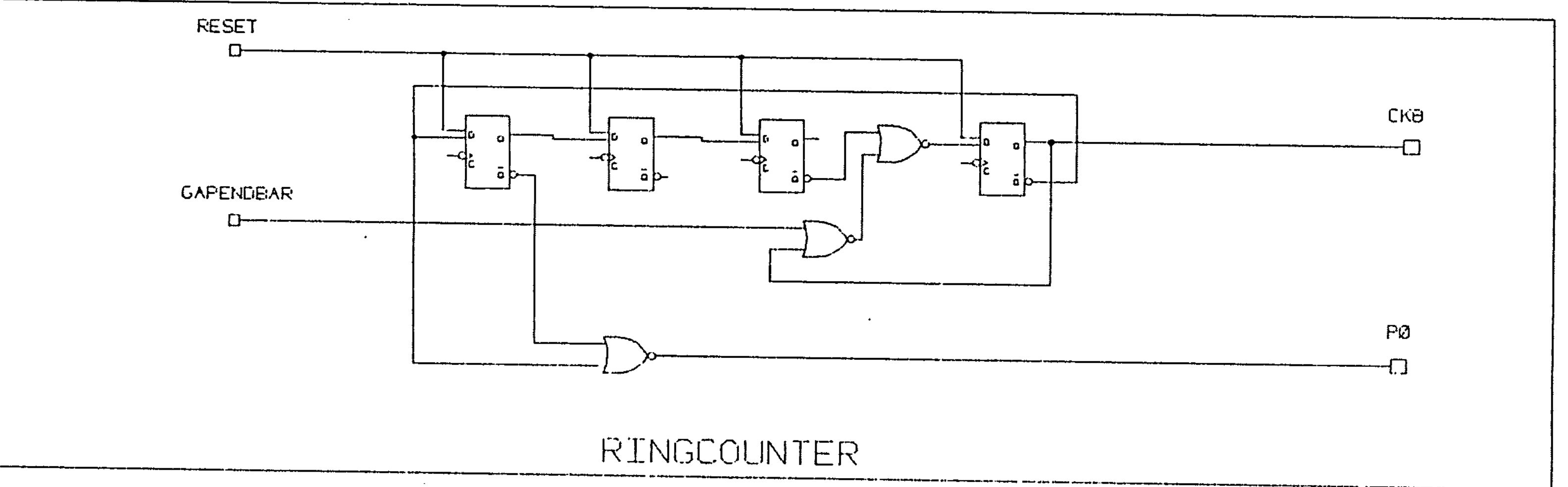
BLOCK DIAGRAM OF ECL CHIP

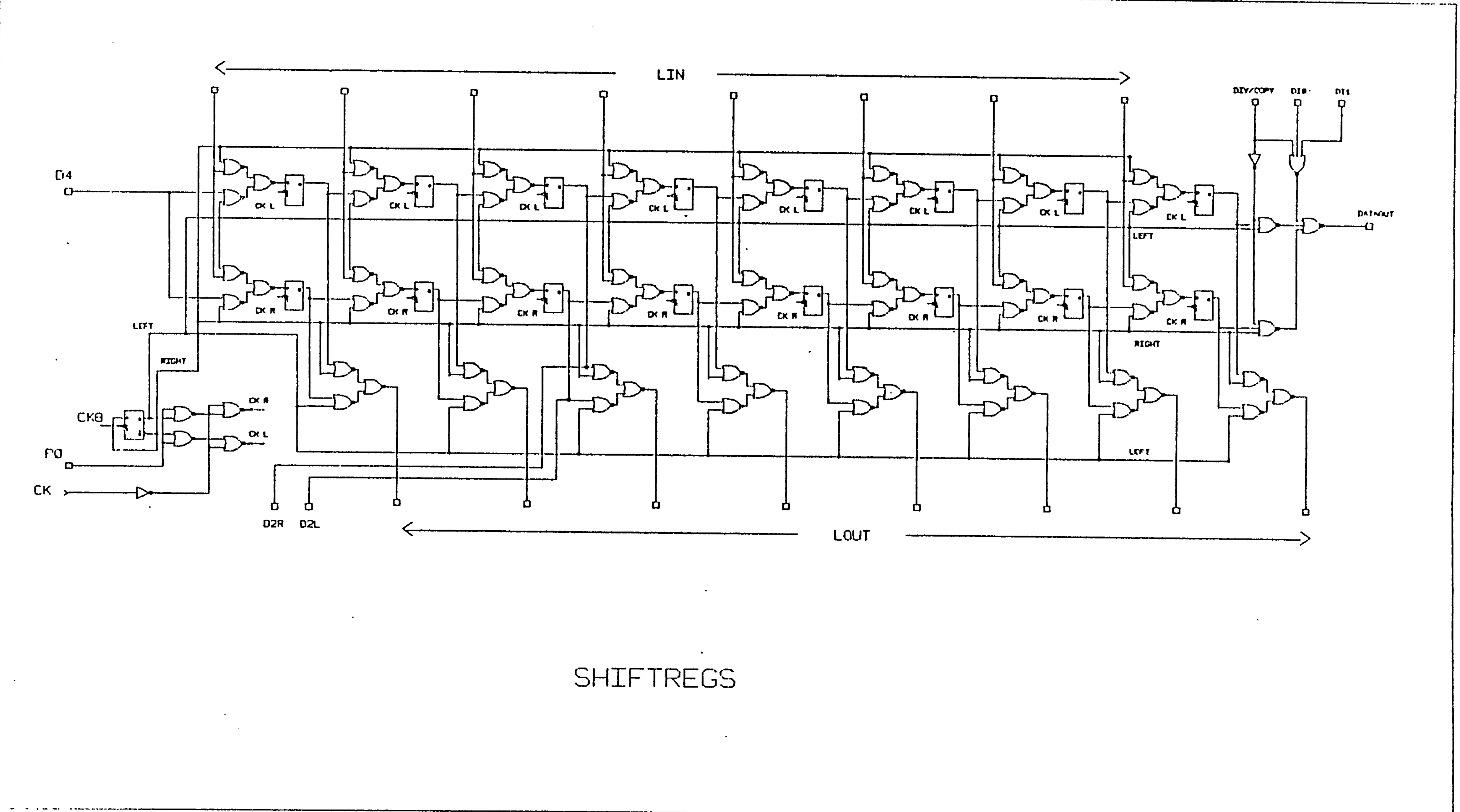
INA
INB
GISD
SERIN
MODERR
DI0
DI1
DATA

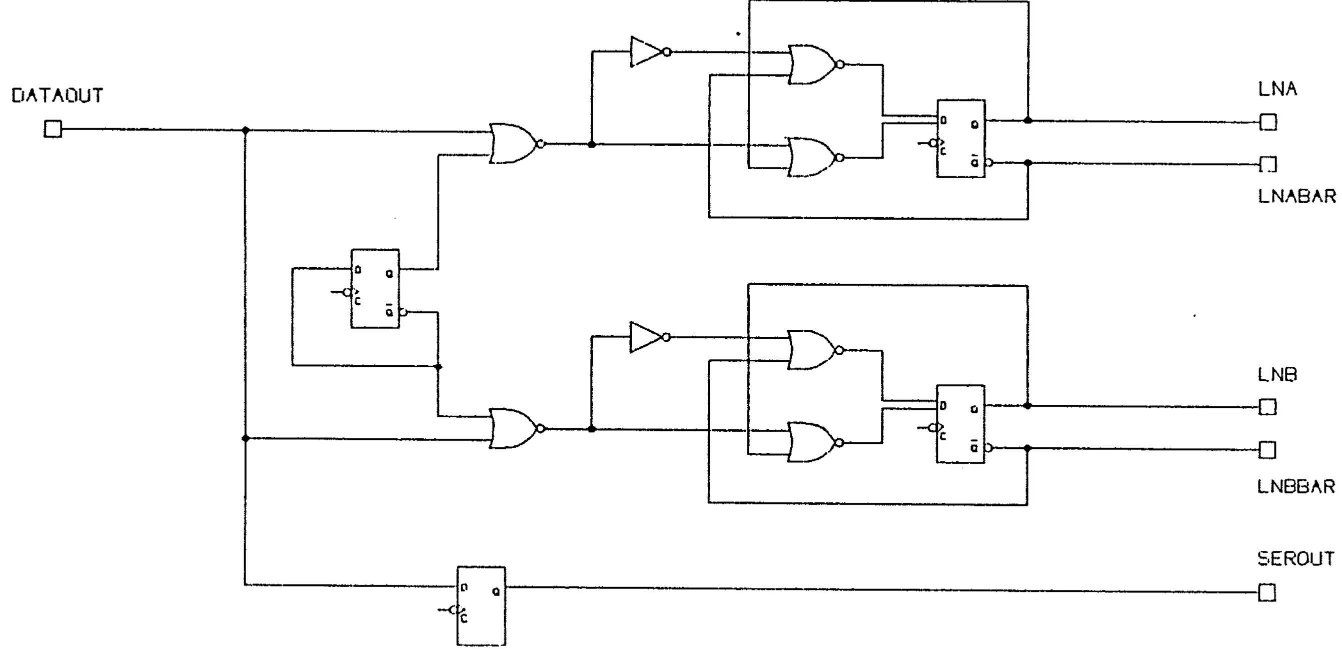
DEMOD



DIØ
DI1
GAP
D2L
D2R
RESET
GAPENDBAR
DETGAP




LIN
D4
CK L
CK R
LEFT
RIGHT
CK8
P0
CK
D2R
D2L
LOUT
DATAOUT
SHIFTREGS


DATAOUT
LNA
LNABAR
LNB
LNBBAR
SEROUT
MODUL