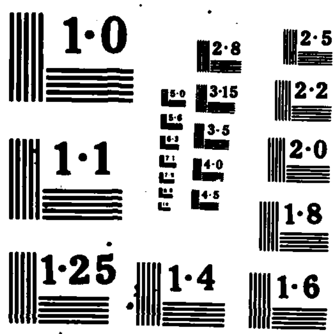
AD-A168 248 HARDWARE PROOFS USING LCF-LSM AND ELLA(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) W J CULLYER ET AL SEP 85 RSRE-MEMO-3832 DRIC-BR-99176 1/1

UNCLASSIFIED F/G 9/2 NL



NATIONAL BURFAU OF S
MICROCOPY RESOLUT TEST

RSRE MEMORANDUM No. 3832

ROYAL SIGNALS & RADAR ESTABLISHMENT

HARDWARE PROOFS USING LCF-LSM AND ELLA

Authors: W J Cullyer and C H Pygott

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3832

TITLE: HARDWARE PROOFS USING LCF - LSM AND ELLA

AUTHORS: W J CULLYER AND C H PYGOTT

DATE: SEPTEMBER 1985

SUMMARY

A method is described for writing the formal specification for a digital system, a high level design which satisfies this requirement and then a gate level realisation, using the languages LCF - LSM and ELLA. Given these formal descriptions, this paper shows how to carry out mathematical proofs to establish that the high level design respects the specification and the the gate level design agrees precisely with the high level design. By this means, a chain of correspondence proof is created from specification to the ultimate realisation, as required during the development of safety critical and security critical hardware. These methods have been used to develop formal proofs for the VIPER microproeessor and this paper forms an essential tutorial for those studying the validation of this new 32 - bit processor.

CONTENTS

1 INTRODUCTION

The central aim of the work of the High Integrity Computing Section of the Computing Division is to devise means of writing formal specifications for safety and security critical hardware and then prove that a particular realisation of this specification conforms with the top level requirement. In the course of developing a novel 32 - bit processor, VIPER, (1, 2, 3) the team has evolved methods of formal specification, design and proof based on the use of the languages LCF - LSM (4, 5) and ELLA (6). The language LCF - LSM was invented in the Computer Laboratory at Cambridge and is outlined in Annex A and ELLA was invented at RSRE, its main features being summarised in Annex B.

This Memorandum explains some of the techniques which have been developed, for the design and validation of synchronous logic, using LCF - LSM and ELLA. By adopting a tutorial style, it is believed that this paper will enable those who have not been exposed to the rigours of formal methods to appreciate the essence of the techniques, without themselves needing to be specialists in the mathematical disciplines involved. However, a detailed understanding of the techniques is crucial for those who will examine the proofs of correspondence between the various levels of documentation for the VIPER microprocessor and other high integrity chips which will be developed in future. Indeed, the need for this Memorandum became apparent when it was realised that even the first step of the proofs for VIPER (7) would be difficult to follow unless the reader had some fundamental guide to the techniques, based on a simple example.

2. TUTORIAL EXAMPLE - INFORMAL SPECIFICATION

The example chosen as the basis for this paper is a six bit counter holding a value "count" which is either retained at its present value, loaded with a new value from the external world, incremented once or incremented twice, depending on the signals on two control lines, denoted by "func", Figure 1. That is :-

```
func = 0  Do nothing: "count" unchanged
func = 1  Load "count" from a 6 - bit parallel input, "loadin"
func = 2  Increment "count":        ie count := count + 1
func = 3  Increment "count" twice: ie count := count + 2
```

This paper shows how to turn this informal specification into a formal description, carry out the design process and prove that the implementation respects the original requirement in every respect. The process of design is not "top down" but relies instead on iterations between various levels of documentation, guided by the knowledge that proofs of correspondence have to be produced between consecutive levels.

It is acknowledged from the start that the ultimate realisation of the counter will be viewed as unusual by those skilled in digital electronics. This is not caused by the proving method. A deliberately strange design has been produced to illustrate as many features of the verification process as possible in a single example.

3 TIMING, SEQUENCE AND FORMAL SPECIFICATION

An issue that must be dealt with in hardware proving is the influence of timing in practical electronic circuits and specifying the effects of delays on the overall functionality of a system. For the counter used as an example in this Memorandum the problem is viewed in the following domains;

a. The top level specification, which has no sense of time or sequence,

b. The "host machine", which has a concept of sequence of operations but no direct relationship with real time or clocks,

c. A high level design which implies underlying clocks and timing, yet is independent of any specific VLSI technology,

d. A specific realisation, which can be analysed to extract real timing, typically using the building blocks and CAD software of a particular VLSI process, eg UK 5000.

At the highest level, the requirement is for a device which performs some mapping such as

```
f(count,loadin,func) -> count
```

This is a pure function, without any implication of using either hardware or software for its implementation. Although not stated yet in formal terms, there is an implication that with defined initial conditions at time t0, the device will execute the function f() above and alter the state of the machine, producing a new value count1 at time t1 and so on, as shown in Figure 2. There is no implication to be drawn about the spacing of these time intervals. Some modes of operation, such as function 0, "do nothing", may require only one "clock tick" for completion, whilst "increment" might be performed by serial addition and use 6 or 7 "clock ticks". This should be compared with the situation when designing a microprocessor, where the designer knows at an early stage what each instruction must do but not the number of clock transitions needed.

Given the assumptions so far, it is possible to devise a formal specification which describes the state transitions produced between t0 and t1, t1 and t2, ...etc. In LCF - LSM, using the constructs listed in Annex A, the possible state transitions can be written down as a single function, as follows;

```
COUNTER  :(word6#word6#word2) -> word6
COUNTER(count,loadin,func) ==
LET funcnum = VAL2 func  IN
LET value= VAL6 count IN
(funcnum = 0 -> count  |
 funcnum = 1 -> loadin |
 funcnum = 2 -> (value = 63 -> WORD6 0 | WORD6(value + 1))
 funcnum = 3 -> (value = 63 -> WORD6 1 |
                 value = 62 -> WORD6 0 | WORD6(value + 2))
)
```

The first line is simply the definition of the modes or types involved, namely an input vector made up of the 6 - bit entities "count" and "loadin", plus the 2 - bit value of "func" which defines the operation to be performed. The function COUNTER delivers the new value of "count" as a single 6 - bit word.

The internal logic of the function has to take account of the 6-bit word length and ensure that there is no attempt to generate a representation of an integer greater than 63. This collateral knowledge of computer arithmetic has been built into the specification above and the known special cases have been given their own limbs in the definition. The particular form of the function above does not impose any constraints on the realisation. The double increment could be done in a single operation, by serial addition or by a large number of other conceivable implementations.

4. THE HOST MACHINE

4.1 General description

For any realistic hardware problem it is not sensible to try and move from a formal requirement directly to a gate level realisation, although for the simple example used in this paper it might be feasible. Following the work at Cambridge (5), it is wise to define a set of simpler functions, which when called in some defined sequence perform the required overall changes in the contents of the counter. Gordon has called this intermediate stepping stone the "host machine" level. Although the phrase is liable to be misunderstood in computing circles, where it is used most frequently in the context of "host /target" program development systems, the nomenclature is retained here for correlation with other published work on hardware proofs. In this context the "host machine" is a high level conceptual view of the counter specified above. By carrying out this first level of decomposition and expressing the conclusions in the form of a state transition diagram, Figure 3, it is possible to move to the next lower level of documentation in LCF - LSM.

The meaning of this state transition diagram is reasonably clear, if the nodes of Figure 3 are viewed as elementary machines, with the following informal attributes;

Node 0: A "FETCH machine" which looks at the value of "func" at the start of the complete counter operation to determine which internal functions should be invoked.

Node 1: An "INC1 machine" which performs a single increment

Node 2: An "INC2 machine" which also performs a single increment (and which internally uses the same mechanism as INC1).

Node 3: A "LOAD machine" which can overwrite the current contents of the counter with the value on the "loadin" lines.

The predicates which define the node to node transitions can be tabulated readily in terms of the values of "func" and a single Boolean variable "double" which is TRUE when a double increment is required. The conditions c0..c4, Figure 3 can then be defined as listed in Table 1. Note that the unlabelled arcs of Figure 3 are traversed unconditionally.

Table 1 Traversal conditions for host machine

Predicate	Expresssion
c0	func = 0
c1	func = 1
c2	func = 2 OR func = 3
c3	NOT double
c4	double

Viewed as a finite state machine, the complete host machine is shown in Figure 4. It should be noted that two additional pieces of state information are now required; "node" and "double". The element "node" indicates which node the host is currently in and "double" was introduced above to define the transition at the output of the INC1 machine. The description of the state of the host machine will be done in terms of the vector

(count, double, node)

each field of which corresponds to a memory element in the host machine; in hardware terms a flip - flop or register. Note that in the correspondence proofs only the element "count" will be involved in comparisons, since the top level specification has no concept of needing "double" or "node". As the design and documentation moves to progressively lower levels, the vectors needed to describe the states of the system grow ever longer.

4.2 Formal definition

From the top level specification and the state transition diagram, it is possible to design the nodes of Figure 3. The steps in deducing the LCF - LSM definitions can be followed from the text on the opposite page;

a. Definition of the vector "major" that defines the state of the host machine.

b. Auxiliary functions are created, such as the arithmetic function ADD1, which will be used in a number of places in the subsequent text.

c. The functionality of each node of Figure 3 is defined, ie the functions FETCH, LOAD, INC1 and INC2. The need for repeated definition of the "enumerated type" (fetchnode | inc1node | inc2node | loadnode), to name the nodes may seem unreasonable but is enforced by the absence of any concept of "global values" in LCF - LSM.

d. The function NEXT is defined to express all single host machine transitions, eg h01 to h02 in Figure 2.

For single transitions, this is an adequate description of the host but to define the behaviour in the presence of time varying signals from the outside world, it is necessary to compose a number of successive calls of the function NEXT, as described in Section 7.

```
MAJOR :(word6#bool#word2)

ADD1 :word6 -> word6
ADD1(x) ==
LET xval = (VAL6 x) IN  (xval = 63 -> (WORD6 0) | WORD6(xval + 1))

FETCH :(word6#bool#word6#word2) -> major
FETCH(count, double, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
LET funcnum = VAL2 func IN
LET fetchnode = WORD2 0 IN
LET inclnode  = WORD2 1 IN
LET loadnode  = WORD2 3 IN
(funcnum = 0 -> (count, twice, fetchnode) |
 funcnum = 1 -> (count, twice, loadnode)  |
                (count, twice, inclnode)        {funcnum = 2 or 3}
)

LOAD: (word6#bool#word6#word2) -> major
LOAD(count, double, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
LET fetchnode = WORD2 0 IN
(loadin, twice, fetchnode)

INC1: (word6#bool#word6#word2) -> major
INC1(count, double, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
LET fetchnode = WORD2 0 IN
LET inc2node  = WORD2 2 IN
(double -> ((ADD1 count), twice, inc2node) |      {double increment}
           ((ADD1 count), twice, fetchnode)       {single increment}
)

INC2: (word6#bool#word6#word2) -> major
INC2(count, double, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
LET fetchnode = WORD2 0 IN
((ADD1 count), twice, fetchnode)

NEXT :(major#word6#word2) -> major
NEXT((count, double, node), loadin, func) ==
LET nodenum = VAL2 node IN
(nodenum = 0 -> (FETCH count double loadin func) |
 nodenum = 1 -> (INC1  count double loadin func) |
 nodenum = 2 -> (INC2  count double loadin func) |
 nodenum = 3 -> (LOAD  count double loadin func)
)
```

5. HIGH LEVEL DESIGN

5.1 Description using LCF - LSM

The next, creative step in the process is to produce a block diagram which implements the host machine, preferably without constraining the ultimate gate level design to a specific technology. However, in practice, a gate level design is sketched using the building blocks of the chosen VLSI design system and this is abstracted to produce the block diagram. Whilst this is essentially a technology dependent design, experience with VIPER has shown that the resulting block diagram can provide the basis for gate level designs in different VLSI technologies. Note from the above description that this paper does not describe a "top down" method of design but a combination of a number of levels of thought, guided by the overall knowledge that formal proofs have to be generated between the various levels.

The high level design for the counter is shown in Figure 5. As can be seen from the facing page, the LCF - LSM description of this design is simple, the internal building blocks being merged in the function COUNTLOGIC to conform with the connectivity of Figure 5. This function describes one "clock tick" only and therefore is directly analogous to NEXT for the host machine. Note the latches on the output lines, which create the global feedback of the finite state machine of Figure 4. The formal mathematics remains in the world of nested function calls and it follows that these synchronously clocked latches must exist in the realisation of the circuit.

```
MULTIPLEX :(word6#word6#bool) -> word6
MULTIPLEX(incout, loadin, mplxsel) ==
(mplxsel -> incout | loadin)


INCLOGIC :(word6#bool) -> word6
INCLOGIC(count, noinc) ==
LET countval = VAL6 count IN
(noinc -> count |
          (countval = 63 -> WORD6 0 | WORD6 (countval + 1))
)


MPLXCON :word2 -> bool                    {multiplexer control}
MPLXCON(node) ==  (NOT(VAL2 node = 3))


INCCON :word2 -> bool
INCCON(node) ==  (VAL2 node = 0)          {increment control}

NEXTNODE :(word2#word2#bool) -> word2   {transition to next node}
NEXTNODE(node, func, double) ==
LET funcnum = VAL2 func IN
LET nodenum = VAL2 node IN
LET fetchnode = WORD2 0 IN
LET inclnode  = WORD2 1 IN
LET inc2node  = WORD2 2 IN
LET loadnode  = WORD2 3 IN
(nodenum = 0 -> (funcnum = 0 -> fetchnode |
                 funcnum = 1 -> loadnode  | inclnode
                ) |
 nodenum = 1 -> (double -> inc2node | fetchnode) |
                 fetchnode               {nodenum = 2 or 3}
)


COUNTLOGIC :(major#word6#word2) -> major
COUNTLOGIC((count, double, node), loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
((MULTIPLEX (INCLOGIC count (INCCON node)) loadin (MPLXCON node)),
 twice,
 (NEXTNODE node func double)
)
```

<u>Note</u>

COUNTLOGIC will be compared with the function NEXT, page 6, when establishing that the high level design conforms to the requirements for the host machine.

5.2 Description using ELLA

This is the point at which the transition to ELLA is made. To simplify this transition, the LCF - LSM describing the high level design of Figure 5 is coded in ELLA, using the library defined in Reference (8). This library interprets the LCF - LSM primitive functions, such as BITS2, VAL6 and so on as ELLA functions. The resulting ELLA description of the high level design is shown on the next four pages. Note that lexically the ELLA text for each block is almost identical to the LCF - LSM given on page 8.

The crucial concept which is introduced in this ELLA text is the definition of a Boolean as (T | F | X | I) where 'X' represents "don't know" and 'I' represents "illegal". The concept of "not knowing" the value of a signal does not exist in LCF - LSM and this is one of the primary reasons why the extra descriptive power of ELLA is essential, the nearer the designer moves to the electronic circuit level. Attempts to work at circuit level using two-level (T | F) Boolean variables tend to lack credibility, because real logic does settle into non - deterministic states when power is switched on and there are many points in hardware design and proving where the sense of a signal is irrelevant and the 'X' in the ELLA text stands for "don't care".

The complete list of ELLA types and functions used in this paper is given below. The reader does not need to know the internal details of the functions to understand the subsequent definitions of the high level design, the circuit description or the method of proof described in Section 9.

5.2.1 Types used

The following types are supported, with the indicated values: -

a) bool = (t | f | i | x)
This is used as the type for boolean signals, and has values: - 't' = true, 'f' = false, 'i' = an illegal or indeterminate signal, 'x' = an irrelevant signal used as an input during testing.

b) word(n)
This is a row of (n) 'bool's, the least significant being element 0, and the most significant being element (n-1).

c) num = number/0, number/1 etc and illegalnum
This is the set of positive integers, plus 'illegalnum' which is used to indicate an illegal or indeterminate value, such as would be obtained by trying to interpret a 'word(n)' as a number if one of the bits was 'i' or 'x'.

d) result = (ok | xxxbadspec | xxxwrongxxx)
This is a type used as the result of comparing the specification of a function with its implementation. The value 'ok' is self explanatory, 'xxxbadspec' is delivered if the function specification has delivered an unexpected value ('i' or 'illegalnum'), and 'xxxwrongxxx' is delivered if the implementation delivers either a different value from the specification or an illegal value (indicating that the implementation depends upon some input it is not meant to).

5.2.2 Functions used

a) AND: (bool,bool) -> bool
This function provides an 'AND' function between two 'bool's. This function is defined such that if both inputs are 't' the result is 't' or if either input is 'f' the result is 'f' (as would be expected) but if one input is 't' and the other is unknown ('i' or 'x') then the result is indeterminate, 'i'.

b) OR: (bool,bool) -> bool
This provides an 'OR' function similar to the 'AND' above.

c) NOT: bool -> bool
This provides an 'NOT' function similar to the 'AND' above.

d) EQUIV: (bool,bool) -> bool
This provides an function similar to the 'AND' above which indicates when two 'bool's are equivalent.

e) EQUAL: (num,num) -> bool
This function compares two 'num's for numerical equality. It should be noted that any 'num' compared with 'illegalnum' gives an indeterminate result.

f) PLUS: (num,num) -> num
Numerical addition. Note that 'illegalnum' plus anything is 'illegalnum'.

g) DIVIDE: (num,num) -> num
Numerical division, similar to 'PLUS'. DIVIDE(a, b) equals the integer division of 'a' by 'b'. Anything divided by zero gives 'illegalnum'.

h) REMAINDER: (num,num) -> num
REMAINDER(a, b) is the numerical remainder left after the integer division of 'a' by 'b'.

i) COMPBOOL: (bool,bool) -> result
This function compares two 'bool's and delivers a 'result'. Note that the order of the inputs is (specification, implementation).

j) COMPNUM: (num,num) -> result
As COMPBOOL, but comparing two 'num's.

k) COMPJOIN: (result,result) -> result
This provides the equivalent of an 'AND' function for 'result's. That is COMPJOIN(ok, ok) delivers 'ok', but all other inputs give either 'xxxbadspec' or 'xxxwrongxxx'.

l) TESTCOUNT: bool -> num
This function delivers a sequence of 'num' starting at 'number/0'. The input parameter has no effect on the delivered value, and is only there because ELLA does not allow a function with a NULL input list.

m) TESTWORD: num -> [14]bool
This function converts a number into a row of 14 'bool's and is used with TESTCOUNT to generate a sequence of 'bool' rows.

The following functions exist as instantiations for different values of {n} and can be compared with the primitives of LCF - LSM listed in Annex A. For example, the LCF - LSM primitive 'WORD6' becomes the ELLA 'WORD{n}' with {n} equal to 6.

n) WORD{n}: num -> word{n}
This function converts a 'num' to a 'word{n}'. If the number is too big to be represented by {n} bits (ie greater than 2**n - 1) or is 'illegalnum' then the result is '[n]i'. It is comparable with the LCF - LSM functions WORD1, WORD2,.....

o) VAL{n}: word{n} -> num
This function converts a 'word{n}' to a 'num'. If the 'word{n}' contains any unknown bits ('x' or 'i') then the result is 'illegalnum'. This mimics the LCF - LSM functions VAL1, VAL2,...

p) EL{n}: (num,word{n}) -> bool
This function delivers the indicated element of the 'word{n}' and is equivalent to the LCF - LSM functions (EL num (BITSn wordn))

q) COMPWORD{n}: (word{n},word{n}) -> result
As for COMPBOOL, but comparing two 'word{n}'s, which is the same as '=' defined between two values of type 'wordn' in LCF - LSM.

On the following two pages the translations of the LCF - LSM descriptions of the blocks of the counter (page 8) are given, converted into ELLA using the library functions described above. This enables subsequent comparison with the gate level description in ELLA to be carried out.

```
\**** c.f. LCF - LSM functions on page 10 ****\

FN MULTIPLEX = (word6: incout loadin, bool: mplxsel) -> word6:
   CASE mplxsel OF t: incout, f: loadin ELSE [6]i ESAC.

FN INCLOGIC = (word6: count, bool: noinc) -> word6:
BEGIN LET countval = VAL6 count.
      OUTPUT CASE noinc OF
             t: count,
             f: CASE EQUAL(countval, number/63) OF
                     t: WORD6 number/0,
                     f: WORD6( PLUS (countval, number/1))
                  ELSE [6]i
                ESAC
            ELSE [6]i
            ESAC
END.

FN MPLXCON = (word2: node) -> bool:
   NOT(EQUAL(VAL2 node, number/3)).

FN INCCON = (word2: node) -> bool: EQUAL(VAL2 node, number/0).

FN NEXTNODE = (word2: node func, bool: double) -> word2:
BEGIN LET funcnum = VAL2 func.
      LET nodenum = VAL2 node.
      LET fetchnode = WORD2 number/0.
      LET inc1node  = WORD2 number/1.
      LET inc2node  = WORD2 number/2.
      LET loadnode  = WORD2 number/3.
OUTPUT CASE nodenum OF
       number/0: CASE funcnum OF
                      number/0: fetchnode,
                      number/1: loadnode,
                      number/2: inc1node,
                      number/3: inc1node
                      ELSE [2]i
                ESAC,
      number/1: CASE double OF
                     t: inc2node,
                     f: fetchnode
                     ELSE [2]i
                ESAC,
      number/2: fetchnode,
      number/3: fetchnode
      ELSE [2]i
      ESAC
END.
```

```
\ ****    cf. LCF-LSM function COUNTLOGIC, page 8  **** \

FN COUNTLOGIC = (word6: count, bool: double, word2: node,
                 word6: loadin, word2: func)-> (word6,bool,word2):
(MULTIPLEX( INCLOGIC(count, INCCON(node)), loadin, MPLXCON(node)),
 EL2(number/0, func),
 NEXTNODE(node, func, double)
).
```

Whilst not required for verification, the complete counter circuit can be modelled using the ELLA simulator as follows;

```
FN COUNTER = (word6: loadin, word2: func) -> (word6, bool, word2):
BEGIN FN DELBOOL  = (bool)  -> bool:  DELAY(i,1).
      FN DELWORD2 = (word2) -> word2: DELAY([2]f,1).
      FN DELWORD6 = (word6) -> word6: DELAY([6]i,1).
      MAKE DELWORD6: count, DELWORD2: node, DELBOOL: double.
      LET cl = COUNTLOGIC(count, double, node, loadin, func).
      JOIN cl[1] -> count,
           cl[2] -> double,
           cl[3] -> node.
      OUTPUT cl
END.
```

Note that the ELLA 'DELAY' function used above provide a one cycle delay for a particular signal. Hence in the above example they provide the memory implied in the counter finite state machine, Figure 4.

6. CIRCUIT DESCRIPTION

The final circuit is shown in Figure 6. The correspondence between groups of gates and the building blocks of Figure 5 is shown also and it should be noted that one gate belongs to two building blocks, to avoid the need to generate the signal (n0 NAND n1) twice. Generating this design from the block diagram of Figure 5 is a human, creative activity.

Description of the circuit diagram of Figure 6 in ELLA is straightforward, using NAND gates, NOR gates, inverters and latches. The complete text is given on the next page. Having done this coding, the ELLA simulator (6) can be used to model the design to gain confidence independently of the formal proofs. If the simulation reveals undesirable or unexpected characteristics this implies either that the proofs will show an inconsistency or that the top level specification does not reflect the true operational requirement. In the latter case the specification will have to be amended and the implementation repeated.

That completes the description of the "forward" design process. The rest of this paper is devoted to the "backward" validation techniques needed to check conformity between the following layers of documentation:

a. The top level specification, in LCF - LSM

b. The host machine definition, in LCF - LSM

c. The high level design, expressed as a block diagram, in LCF - LSM <u>and</u> ELLA

d. The circuit diagram, described in ELLA.

The three sets of proofs are described in Sections 7,8 and 9, respectively.

The following is the description of the counter circuit, Figure 6, expressed in ELLA. Note that the circuit primitives (NAND gates etc) are described in terms of library functions, Section 5, whilst the circuit is described using these circuit primitives only.

```
\****  Circuit elements required for implementation **** \

FN   INV = (bool:a)          -> bool: NOT a.

FN  NOR2 = (bool: a b)       -> bool: NOT(a OR b).

FN NAND2 = (bool: a b)       -> bool: NOT(a AND b).

FN NAND3 = (bool: a b c)     -> bool: NOT(a AND b AND c).

FN NAND4 = (bool: a b c d) -> bool: NOT(a AND b AND c AND d).

FN  XNOR = (bool: a b)       -> bool: EQUIV(a, b).

\**** The circuit description, note that the specification ****\
\**** functions INCCON,MPLXCON and NEXTNODE are combined  ****\
\**** into a single function ****\

FN MPLEXCIRC = (word6: incout loadin, bool: mplxsel) -> word6:
BEGIN FN BITSEL = (bool: lbit lsel incbit incsel) -> bool:
                 NAND2(NAND2(lbit, lsel), NAND2(incbit, incsel)).
      LET mplxselbar = INV mplxsel.
OUTPUT [INT k=1..6]BITSEL(loadin[k],mplxselbar,incout[k],mplxsel)
END.

FN INCCIRC = (word6: count, bool: noinc) -> word6:
BEGIN LET noincbar = INV noinc.
      LET ic1 = XNOR(count[1], noinc).
      LET ic2 = XNOR(count[2],NAND2(noincbar, count[1])).
      LET ic3 = XNOR(count[3],NAND3(noincbar,count[1],count[2])).
      LET carry4bar = NAND4(noincbar,count[1],count[2],count[3]).
      LET ic4 = XNOR(count[4], carry4bar).
      LET carry4 = INV carry4bar.
      LET ic5 = XNOR(count[5],NAND2(carry4, count[4])).
      LET ic6 = XNOR(count[6],NAND3(carry4, count[4], count[5])).
      OUTPUT(ic1, ic2, ic3, ic4, ic5, ic6)
END.

FN CONTROLCIR = (word2: node func, bool: double) ->
                (bool,bool,word2):
BEGIN LET    inccon = NOR2(node[1], node[2]).
      LET   mplxcon = NAND2(node[1], node[2]).
      LET    common = NAND3(inccon, INV func[2], func[1]).
      LET nextnode1 = NAND2(common, NAND2(inccon, func[2])).
      LET nextnode2 =
                NAND2(common, NAND3(double,node[1],INV node[2])).
      OUTPUT (inccon, mplxcon, (nextnode1, nextnode2))
END.
```

7. CORRESPONDENCE BETWEEN SPECIFICATION AND HOST MACHINE

7.1 Technique

The first link in the chain of proofs is to establish that the host machine defined on page 6 conforms to the top level specification on page 3. The steps in the proofs are;

a. Generation of a "spanning tree", which shows all possible paths through the state transition diagram.

b. Elementary algebraic substitutions in the functions for the host machine to derive one partial function for each branch of this tree.

c. Matching substitutions in the top-level specification to produce a partial specification.

c. Comparison of the value of "count" delivered by these skeleton descriptions to check that the host implies the specification.

7.2 Generation of spanning tree

To break the proof down into easily managed cases, the spanning tree is derived, to illustrate all possible "walks" from the "fetch" node around the state transition diagram, Figure 2. The resulting tree for the host machine is shown in Figure 7. For such a simple example, the spanning tree can be obtained by inspection but formally the evaluation of the tree is done using the connectivity matrix;

		To node			
		0	1	2	3
From node	0	1	1	0	1
	1	1	0	1	0
	2	1	1	0	0
	3	1	0	0	0

Standard algorithms exist for deriving the spanning tree from this matrix (9). Consider any branch of this tree, for example, branch C. To follow this route it must be true that to travel from node 0 to node 1 the condition on exit from node 0 must be ((func=2) OR (func=3)). Equally, if the next transition is to be to node 0, the predicate (NOT double) must be TRUE on exit from node 1. To create the overall predicate for this path to be traversed a notation is needed which expresses the time varying nature of the various signals, which determine the values of "func" and "double".

7.3 Sequences of signals

Reference to Figure 2 will show that a notation is required for the values of the various signals at times t00,t01... in the host machine. In formal terms, the successive values are defined as a sequence of signals, represented in LCF - LSM using list constructors, Annex A, to form types such as "wordn list", ie a sequence of n - bit values. For this example, let;

```
loadinsigs = loadin00, loadin01, loadin2........
funcsigs   = func00   , func01   , func02......
```

where the formal mode of both "loadinsigs" and "funcsigs" is "word6 list". Note from Figure 2 that the points at which the members of these sequences are assumed to exist may not be spaced evenly in real time. All that is required at this level of the proof is that the sequences can be assumed to exist and that the selection of the nth elements of both "loadinsigs" "funcsigs" will produce a pair of inputs which are stable and coexist at the same instant in the real world.

7.4 "Hoisting" of exit conditions

Consider the conditions under which path C is executed. The first function application in the host machine will be a call of NEXT, which internally causes a call of FETCH. If path C is to be followed this must result in the exit condition ((func=2) OR (func=3)). By inspection of the description of the host machine on page 6 this means that the application of the zero elements of the sequences "loadinsigs" and "funcsigs" to the function FETCH must deliver the result (count, twice, inclnode). In words, the requirement for this to be true are as follows;

```
"FOR ALL
     possible values of 'count' resulting from the last operation
 AND both possible current values of 'double' (just before fetch)
 AND any sequence of 'loadin' signals
 AND any sequence of 'func' signals
 SUCH THAT the high order bit of the first value of 'func' is set
           (func=2 OR func= 3), ie condition c2 in Table 1 is TRUE
 FETCH delivers (count, twice, inclnode)".
```

In predicate calculus the universal quantifier "FOR ALL" can precede any list of variables and is represented in this paper by an exclamation mark (!). The qualifier "SUCH THAT" precedes the defining predicate and is typed as a full stop (.). This means that the long winded statement above can be written out formally by substitution in FETCH, page 6, yielding;

```
!count double loadinsigs funcsigs.(func= 2 OR func=3) ->
(FETCH count double (EL 0 loadinsigs) (EL 0 funcsigs)) ->
(count, (EL 0 (BITS2 (EL 0 funcsigs))), inclnode)
```

There are several points to note about this expression;

a. The exit condition has been "hoisted" into the defining predicate and become an entry condition, expressed in terms of the state of the system before FETCH is invoked.

b. The use of (EL 0 ...) to extract the first member of a sequence of signals.

c. The use of (EL 0 (BITS2 (EL 0 ......))) to extract the least significant bit of the first element in "funcsigs" in order to determine the value of "twice."

Now repeat the exercise for node 1 on branch C of the spanning tree, calling the function INC1 with the <u>second</u> signals in the sequences "loadinsigs" and "funcsigs" as parameters, to create the exit conditions such that the collateral delivered is;

((ADD1 count), twice, fetchnode)

However, this is the state created when "double" on <u>input</u> to INC1 is FALSE (ie c3 = TRUE, Table 1). Therefore in dealing with this second function call there is no need to "hoist" exit conditions into the predicate qualifying the call of INC1. Remembering that the predicate c2 = (func = 2 OR func = 3) is definitive for this path already, the defining predicate for this second function application must be the intersection (c2 AND c3) In formal terms;

```
!count double loadinsigs funcsigs.(c2 AND c3) ->
(INC1 count F (EL 1 loadinsigs) (EL 1 funcsigs)) ->
((ADD1 count, (EL 0 (BITS2 (EL 1 funcsigs))), fetchnode)
```

The direct substitution "double = F" can be made in the call of INC1 by virtue of c3. Notice that the value of "twice" delivered depends on the second signal in the sequence "funcsigs" The proofs should establish that this new value is irrelevant.

All that remains to describe path C in the spanning tree precisely is the composition of the calls of FETCH followed by INC1, which involves two steps;

a. The "hoisting" of the entry condition for the call of INC1, (NOT double) backwards so that it becomes an entry condition for the initial call of FETCH, and

b. Substitution of the outputs from FETCH as inputs of INC1.

Carrying out both steps gives a partial function for the host machine, HOST'C, with the following definition;

```
!count double loadinsigs funcsigs.(c2 AND c3) ->
HOST'C(count, double, loadinsigs, funcsigs) ==
((ADD1 count), (EL 0 (BITS2 (EL 1 funcsigs))), fetchnode)
```

The composite entry condition for (c2 AND c3) in terms of the universally quantified inputs is;

```
(c2 AND c3) == (EL 1 (BITS2 (EL 0 funcsigs))) = T AND
               (EL 0 (BITS2 (EL 0 funcsigs))) = F
```

This can be recognised in integer form as (func00 = 2), which is the value expected from the top level specification for the single increment operation.

Although these operations have been explained in exhaustive detail for tutorial purposes, the steps become largely automatic once a number of proofs have been completed. Given the partial function for HOST'C, comparison with the corresponding partial specification should yield one limb of the proof that the host machine conforms with the requirement in the top level specification, as given in the next section.

7.5 Proof for limb C

The analysis in 7.4 has shown that path C can be defined uniquely by the expression;

```
!count double loadinsigs funcsigs.(c2 AND c3) ->
((ADD1 count), (EL 0 (EL 1 funcsigs)), fetchnode)
```

The algebra from this point is elementary. Expand ADD1, down to the underlying primitive LCF - LSM functions giving;

```
!count double loadinsigs funcsigs.(c2 AND c3) ->
((VAL6 count)=63 -> (WORD6 0,(EL 0 (EL 1 funcsigs)),fetchnode)|
    (WORD6((VAL6 count) + 1),(EL 0 (EL 1 funcsigs)),fetchnode))
```

Now apply the same compound predicate to the specification on page 3 to from a partial specification;

```
!count loadin func.(c2 AND c3) ->
COUNTER'C(count, loadin, func) ==
((VAL6 count)=63 -> (WORD6 0 | WORD6((VAL6 count)+1))
```

By comparison of the underlined expressions for the value to be delivered it is clear that the host machine obeys the specification for the single increment mode of the counter.

By using the same technique for branches A, B and D of the spanning tree, the complete proof of correspondence between the host machine and the top level specification can be produced. The details are given in Annex C. It should be noted that although the example presented in this Section involved only a single comparison of host and specification functions, some proofs of this kind have to be split into internal cases. Branch D of the spanning tree for this example has three such internal cases and some of the VIPER proofs involve 6 internal limbs, to cope with various combinations of predicates. The issue of applying automatic theorem provers to this work is discussed later.

8 CORRESPONDENCE BETWEEN HOST MACHINE AND HIGH LEVEL DESIGN

8.1 Technique

This is a different type of proof. Whereas the proofs from the host machine back to the specification required the derivation of partial functions on paths through the state transition diagram, this is not needed at the next level. The correctness of the high level electronic design must be established by showing that the function COUNTLOGIC on page 12 provides exactly the same functionality as the host machine function NEXT, (page 6) in all circumstances. A brief study of NEXT, with its case limbs corresponding to each node of the state transition diagram, shows that the best strategy for this proof is to take a node at a time.

As in Section 7, the proof for one node is explained in detail in this Section, with the remaining details being given in Annex D. As the level of the proofs moves towards the gate level design of the counter, the details become ever more repetitive and tedious. However, it is vital to persevere with the algebra, because errors can occur at any level of the documentation.

8.2 Proof for node 0

Note that the formal mode of the function NEXT for the host machine is identical to that for COUNTLOGIC in the description of the high level design. This means that precise equality can be established between the two levels. (In the VIPER proofs the state vector for the high level design is longer and has more detail than that for the host machine and therefore the proofs are by implication, as in Section 7).

By substituting node = 0 in the function NEXT for the host machine it is easy to show that,

```
NEXT(count, double, #00, loadin, func) ==
(FETCH count double loadin func)
```

Expand FETCH,

```
NEXT(count, double, #00, loadin func) ==
LET twice = (EL 0 (BITS2 func)) IN
LET funcnum = VAL2 func IN
LET fetchnode = WORD2 0 IN
LET inclnode  = WORD2 1 IN
LET loadnode  = WORD2 3 IN
(funcnum = 0 -> (count, twice, fetchnode) |
 funcnum = 1 -> (count, twice, loadnode)  |
 funcnum = 2 -> (count, twice, inclnode)  |
                (count, twice, inclnode))              (H.0)
```

Now form the corresponding function for the high level design,

```
COUNTLOGIC(count, double, #00, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
((MULTIPLEXER (INCLOGIC count (INCCON #00)) loadin
  (MPLXCON #00)), twice, (NEXTNODE #00 func double))
```

But (INCCON #00) = T, (MPLXCON #00) = T and (INCLOGIC count T) = count and therefore,

```
COUNTLOGIC(count, double, #00, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
(count, twice, (NEXTNODE #00 func double))             (D.0)
```

Comparing equations (H.0) and (D.0), it remains to be shown that the expressions for the next node to be visited are the same. By expansion of NEXTNODE, and comparison of the underlined elements in equations (H.0) and (D.0) it can be proved that the high level design reproduces the behaviour of node 0 in the host machine.

Annex D gives details of the proofs for nodes 1, 2 and 3. From this it can be deduced that the high level electronic design illustrated in Figure 6 is a valid way of implementing the host machine. At this stage in the proofs the first three layers of documents have been related, ie high level design -> host machine -> top level specification. The proof that the gate level design agrees with the high level design remains to be done.

Given that the high level design can be represented equally well in ELLA, as described in Section 5, the last step in the proofs can be done using ELLA itself. Admittedly, there should be a formal proof that the LCF - LSM and ELLA descriptions of the high level design are mathematically identical, but this cannot be attempted until the constructs of ELLA have been described in first order logic in a more systematic way. For the moment, the reader will have to accept that the writing of the LCF - LSM primitive functions in ELLA gives a sound interface, although not proven mathematically.

9 CORRESPONDENCE BETWEEN THE HIGH LEVEL DESIGN AND CIRCUIT DESCRIPTION

9.1 Technique

The method used to prove the circuit description agrees with the high level design is known as "intelligent exhaustion" and is described in Reference (8). Essentially, the method involves comparing the results delivered by the 'high level' specification functions (as detailed in Section 5) and the equivalent 'circuit description' functions (as detailed in Section 6), for all input states. This circuit consists of three blocks, and so the proof of correspondence is also in three parts.

As an example, Section 9.2 shows the testing necessary to prove that one of these blocks, namely the multiplexer circuit MPLEXCIRC, agrees with the 'high level' multiplexer function MULTIPLEX. If simple exhaustive testing was used in this case, with 13 inputs to these functions, a total of 8192 separate tests would need to be performed to prove their correspondence. However, from the specification of the multiplex (MULTIPLEX) it is known that if one considers a single bit of the output, then that should depend solely upon the state of the equivalent bit in the selected input. That is all inputs apart from the 'select' line and the appropriate bit of the selected input word should have no effect on the output.

The method of intelligent exhaustion allows the multiplexer to be tested in precisely this way. For each output bit, tests are made to ensure that which ever input word is selected, both an input bit = t and input bit = f, gives the correct output. That is four tests are needed for each output bit, or a total of 24 tests. All the inputs regarded as being irrelevant are given the value 'x'. Should the circuit design be wrong, so that under some circumstance an output bit depends upon an input bit that was thought to be irrelevant, this will be detected by the function delivering an indeterminate value 'i' instead of the expected value. This will of course cause the comparison to fail.

It should be noted that all these tests have been run and produce the correct values for the multiplexer. All the corresponding proofs by "intelligent exhaustion" for the other building blocks were successful and it is concluded that the gate level circuit agrees precisely with the higher level block diagram.

9.2 Testing the MULTIPLEX logic

In order to test the multiplexer, without using the 'method of intelligent exhaustion', all 8192 inputs states would have to be examined. Using this method, the same effect is achieved with just 24 tests. Tests 0 to 3 examine bit 0 of the multiplexer. The four tests are as follows: -

```
test0: 'loadin' bit 0 = f, 'mplxsel' = f,  all other inputs = x
test1: 'loadin' bit 0 = t, 'mplxsel' = f,  all other inputs = x
test2: 'incout' bit 0 = f, 'mplxsel' = t,  all other inputs = x
test3: 'incout' bit 0 = t, 'mplxsel' = t,  all other inputs = x
```

That is, tests 0 and 1 ensure that when 'loadin' is selected, an input of f or t gives the correct result, whilst tests 2 and 3 do the same for 'inccon'. As only bit 0 is being tested 'EL6' is used in RUNTESTS to pick the appropriate output from the two 'word(6)'s. Tests 4 to 7 repeat the process for bit 1 etc.

```
FN TESTVECTORS = (bool: dummy) -> (num, (word6, word6, bool)):
BEGIN LET testnumber = REMAINDER(TESTCOUNT dummy, test/24). \0..23\
      LET testbits = CASE REMAINDER(testnumber, test/4) OF
                          test/0: (f,f), test/1: (t,f),
                           test/2: (f,t), test/3: (t,t)
                     ESAC.
      LET bitnum = DIVIDE(testnumber, test/4).
      LET ip  = CASE bitnum OF
                     test/0: (testbits[1], x, x, x, x, x),
                     test/1: (x, testbits[1], x, x, x, x),
                     test/2: (x, x, testbits[1], x, x, x),
                     test/3: (x, x, x, testbits[1], x, x),
                     test/4: (x, x, x, x, testbits[1], x),
                     test/5: (x, x, x, x, x, testbits[1])
                ESAC.
      LET ip1 = CASE testbits[2] OF f: [6]x, t: ip   ESAC.
      LET ip2 = CASE testbits[2] OF f: ip,   t: [6]x ESAC.
      OUTPUT (bitnum, ( ip1, ip2, testbits[2]))
END.

FN RUNTESTS = (bool: dummy) -> restype:
BEGIN LET  vector = TESTVECTORS dummy.
      LET    spec = EL6( vector[1], MULTIPLEX vector[2]).
      LET    calc = EL6( vector[1], MPLEXCIRC vector[2]).
      LET compare = COMPBOOL(spec, calc).
      OUTPUT (vector[1], spec, calc, compare)
END.
```

All the tests using this ELLA text gave correct answers and thereby multiplexer circuit represented by the function MPLEXCIRC is known to be a correct implementation of the higher level function MULTIPLEX.

10. CONCLUSIONS

By combining Gordon's work on LCF - LSM with Pygott's novel work using ELLA a formal method has been created for the specification, design and validation of complex digital circuits. This work is based on the use of first order logic and the techniques are suitable for synchronous circuits only. The change of language, from LCF - LSM to ELLA, is straightforward and should prove acceptable to designers. Sc far, all the proofs of correspondence at the higher levels, in LCF - LSM, have been done by hand, without automated assistance. The next phase of the research will involve investigations into the use of automated aids to assist in verification.

This whole topic has increasing importance in the VLSI industry. Although the RSRE work is motivated mainly by the need to produce safe chips for use in safety critical situations, the introduction of the disciplines described in this paper into VLSI design procedures could prevent costly iterations in the designs of more widely applicable devices.

11. ACKNOWLEDGEMENTS

This work relies in part on Dr M Gordon's original research in Cambridge on hardware specification and proof and Mr J Morison's invention of ELLA at RSRE. Dr J M Foster and Mr I Currie have provided valuable guidance on the formal mathematical methods used. The techniques described in this paper have been developed in the High Integrity Computing Section of the Computing Division of RSRE by a team made up of Dr J Kershaw and the authors.

12. REFERENCES

1. KERSHAW, J "VIPER: a microprocessor for safety - critical applications"
RSRE Memorandum 3754, September 1984

2. PYGOTT, C H "Electrical, environmental and timing specification of VIPER microprocessor"
RSRE Memorandum 3753, September 1984

3. CULLYER, W J "Formal specification of VIPER microprocessor"
RSRE Memorandum 3738, September 1984

4. GORDON, M "LCF - LSM"
University of Cambridge Computing Laboratory, Technical Report 41

5. GORDON, M "Proving a computer correct"
University of Cambridge Computing Laboratory, Technical Report 42

6. MORISON, J D, PEELING N E and THORP T L,
"*ELLA*: Hardware description or specification?"
Proceedings IEEE International Conference, CAD-84, Santa Clara, November 1984

7. CULLYER, W J "VIPER - Correspondence between specification and host machine"
RSRE Memorandum 3801, June 1985

8. PYGOTT, C H "Formal proof of correspondence between a hardware module and its gate level implementation"
RSRE Report 85012, June 1985

9. CARRE, B A "Graphs and networks"
Oxford University Press, Computer Science Series, 1982

ANNEX A. LCF - LSM

The material in this Annex is a very brief digest of that presented by Gordon in Reference 5 and contains enough detail to enable the text of Section 3 to be read and understood.

The form of LCF-LSM in use at Cambridge has the following built-in types;

T, F	Values of type 'bool'
0, 1, 2, ...	Numbers of type 'num'
b1..bn	Words of type 'wordn', elements being 0 or 1
[]	Empty list of type '* list', where * is any other type

Certain built-in operators are provided, with associated axioms;

=	equality between values	*#* -> bool
+	addition on values of type num	num#num -> num
OR	disjunction	bool#bool -> bool
AND	conjunction	bool#bool -> bool
XOR	exclusive OR	bool#bool -> bool

This paper makes use of a further group of functions supported by rules to permit simplification during subsequent theorem proving.

NOT	negation	bool -> bool
CONS	list constructor	* -> * list -> * list
HD	head of list	* list -> *
TL	tail of list	* list -> * list
EL	n th element of list	num -> * list -> *
SEG	sublist of a list	(num#num) -> * list -> * list
V	number denoted by a bit list	bool list -> num

Both EL and SEG use words numbered from 0 at the least significant end, as employed throughout the main text of this paper, i.e.

```
EL i [tn...t0]  -> ti
SEG (i,j) [tn...t0] -> [tj...ti]
```

with obvious need for exception handling in the tools if the bounds are violated.

ANNEX B ELLA: A brief introduction to its syntax

This annex will outline those features of ELLA used in this paper. It contains more detail than could be included in the body of the paper, but is not intended to be a complete description of the syntax.

All ELLA programs consist of type declarations and functions only. There are no global variables or constants in ELLA. Also, ELLA has no in-built data types, hence all data types to be used must be declared explicitly.

B.1 Primitive data types

Two sorts of primitive data types can be declared; enumeration and integer types. The declaration of an enumeration type consists of the type name, such as "bool", and a list of the values it may have, such as "t" "f" "x" and "i". That is: -

```
TYPE bool = NEW( t | f | x | i ).
```

Note the order of the values in the declaration is irrelevant. Integer types consist of the type name, such as "countint", a prefix name, such as "count" and the range of values that integers of this type may have, such as 0 to 10. That is: -

```
TYPE countint = NEW count/(0..10).
```

Objects of type "countint" can have the values "count/0" "count/1" etc to "count/10". The prefix "count" distinguishes integers of type "countint" from integers of any other type (which would have a different prefix).

B.2 Compound data types

Compound data types are collections of primitive data types or other compound data types. They are of two forms; rows and structures. A row is a collection of identical data types. For example, "(t, f, t)" is a row of three "bool"s, the type of this object can be expressed as either "(bool, bool, bool)", or "[3]bool". Structures are collections of different data types, such as "(t, count/0, (t, f))". The type of this object is "(bool, countint, [2]bool)".

Both rows and structures are indexed in the same way. If the above example of a structure was called "struct", then "struct[1]" would be the first element of the structure, that is the "bool" with value "t". Similarly, "struct[2]" is the "countint" with value "count/0", and "struct[3]" is the "[2]bool" with value "(t, f)". This row can be indexed in the same manner, such that "struct[3][1]" is a "bool" with value "t", and "struct[3][2]" is a "bool" with value "f".

B.3 Functions

Functions in ELLA are similar to mathematical functions, in that they deliver a value and can only operate upon those values passed to the function when it is used. That is, there are no global variables.

A function consists of a heading and a body. The heading describes the types of the objects that the function will operate upon, together with their local names (ie the names by which the parameters are known within the function body) and the type of the object delivered by the function. For example, consider a function called "TIMEOUTS", which is to operate on two objects of type "bool", and one object of type "[3]bool" and is to deliver a structure with "bool" and "[3]bool" elements. The names, within this function, of the objects to be operated upon are "reset" "inc" and "current". That is the function heading is: -

```
FN TIMEOUTS = (bool: reset inc, [3]bool: current) -> (bool,[3]bool):
```

A function body consists of either a single 'expression' (qv), or "BEGIN" followed by a number of 'statement's (qv) "OUTPUT" followed by an expression and "END.". The value delivered by the function in the first case is the value of the expression, and in the second is the value of the expression between "OUTPUT" and "END.". The type of the delivered value must be the same as that indicated in the function heading.

B.4 Expressions

There are four types of ELLA expression. All of them have the property that they deliver a value. They are; simple, function calls, CASE and ARITH.

a) Simple expressions

These are structures composed of explicit data values, names local to the function containing the expression, or other expressions. Explicit values are those values declared as being a particular data type, such "t" or "count/4" in section 1. Local names are the values associated with the parameters named in the function heading or the value associated with a named expression (see LET section 5). Note that a single value can also be a simple expression. For example: -

t, inc, (reset, inc, (t,t,t)) are simple expressions.

b) Function calls

An expression can be the result delivered by applying a function to a particular set of values. The values operated upon can be any sort of expression including simple expressions (ie explicit values or local names). Given a function called OR that operates on two "bool"s and delivers a "bool", the OR of "a" and "b" (where "a" and "b" are local named values of type "bool") is given "OR(a, b)".

There are two exceptions to this rule. If the function has a single parameter, the brackets are not needed. So "NOT(a)" can be written as "NOT a". If the function has two parameters, it can be placed between the values it is to operate upon. So "OR(a, b)" can be written as "a OR b". Similarly "a OR b OR c OR d" is the same as "OR(OR(OR(a, b), c), d)".

c) CASE expressions

The structure of a CASE expression is: -

CASE expression OF {value: expression} ELSE expression ESAC

Where {...} means repeated any number of times.

The first expression is evaluated and the resulting value is compared with the 'value' component of the 'value: expression' pairs. If these are equal, the value delivered by the CASE expression is the value of the 'expression' component. If none of the 'values' are equal to the evaluated value, the value of the expression between ELSE and ESAC is delivered. If it is known that the 'value: expression' pairs cover all possible values the "ELSE expression" term may be omitted.

d) ARITH expressions

If a function is required to perform arithmetic operations on integer types, and deliver an integer type result, the body of the function may be a single ARITH expression. The required arithmetic operation may be expressed in an ALGOL like manner using the operators "+", "-", "*", "%" etc. Note that "%" is used for integer division as "/" is already part of the name of each ELLA integer. Condition clauses may be formed using an IF..THEN..ELSE..FI construct.

B.5 Statements

There are three types of statement that may form the body of a function. Note that when a number of expressions of the same type is required, the statement indicator (LET MAKE or JOIN) is only required once. So that, "LET a = t, b = f." is the same as "LET a = t. LET b = f.".

a) LET
The LET statement allows a name to be associated with the value of an expression. So that "LET aorb = a OR b." means that "aorb" is now a recognised local name associated with the value of the expression "a OR b".

b) MAKE and JOIN
In all the above examples, local names must have been declared as the parameter of a function or a LET statement, before they could be used in an expression. Without some means of overcoming this restriction it would be impossible to model circuits with feedback (and hence memory). Consider a pair of cross coupled NAND gates,Figure 8. The description of this circuit as: -

```
FN RSLATCH = (bool: a b) -> bool:
   BEGIN LET nand1 = NAND(a, nand2),
             nand2 = NAND(b, nand1).
         OUTPUT nand1
   END.
```

is illegal, as "nand2" is used in an expression before it is declared. MAKE allows a name to be associated with the output of a particular call of a function before the inputs to that function are available. JOIN allows the required inputs to a function named by MAKE to be connected after they have been declared. Hence, a legal version of the same function would be:

```
FN RSLATCH = (bool: a b) -> bool:
BEGIN MAKE NAND: nand2.
      LET nand1 = NAND(a, nand2).
      JOIN (b, nand1) -> nand2.
      OUTPUT nand1
END.
```

B.6 DELAY

A special expression, "DELAY", exists which is used to create functions which will act to delay a signal for a number of 'clock ticks'. It is used as:

```
FN DELAYANY = (anytype) -> anytype: DELAY(value, integer).
```

This defines a function, DELAYANY, that delays a signal of type 'anytype' for 'integer' clock ticks. The parameter 'value' gives the initial value of the output of DELAYANY.

ANNEX C DETAILS OF CORRESPONDENCE BETWEEN HOST MACHINE AND SPECIFICATION

PROOF LIMB A

```
!count double loadinsigs funcsigs.c0 ->
HOST'A(count, double, loadinsigs, funcsigs) —
(FETCH count double (EL 0 loadinsigs) #00)
```

Substituting in FETCH,

```
!count double loadinsigs funcsigs.c0 -> (count, F, fetchnode)
                                                        (HA.1)
```

The corresponding partial function from the specification is;

```
!count loadin func.c0 -> count                          (SA.1)
```

Therefore the specification is satisfied.

PROOF LIMB B

```
HOST:
!count double loadinsigs funcsigs.c1 ->
HOST'B(count double loadinsigs funcsigs) —
LET loadnode = WORD2 3 IN
LET major1 = (count, T, loadnode) IN
(LOAD count T (EL 1 loadinsigs) (EL 1 funcsigs))
```

Finally for the host, expand the function LOAD, giving the result;

```
!count double loadinsigs funcsigs.c1 ->
((EL 1 loadinsigs),(EL 0 (BITS2 (EL 1 funcsigs))),fetchnode) (HB.1)
```

where the underlined element is the new value of "count." Now create the corresponding partial specification, by substituting the predicate c1 in the function COUNTER on page 3. Elementary algebra gives;

```
!count loadin func.c1 ->
COUNTER'B(count,loadin,func) — loadin                         (SB.1)
```

Comparing the underlined values of "count" in equations (HB.1) and (SB.1),it is clear that the host machine implies the specification. Notice that the specific value of "loadin" used is the second in the sequence, at time t01.

PROOF LIMB C

This has been proved already, in Section 7.5

PROOF LIMB D

This is harder, since a proof is required that the two single increment operations in the host machine produce the correct numerical results. The proof is split into three cases, corresponding to entry conditions,

1. VAL6 count = 63
2. VAL6 count = 62
3. VAL6 count < 62

PROOF LIMB D1 (VAL6 count = 63)

By successive substitution,

```
!double loadinsigs funcsigs.(c2 AND c4 AND count = #111111) ->
((WORD6 1), (EL 0 (EL 2 funcsigs)), fetchnode)          (HD.1)
```

From the specification,

```
!loadin func.(c2 AND c4 AND count = #111111) ->
(WORD6 1)                                                (SD.1)
```

Case 1 is thereby proven.

PROOF LIMB D2

By the same substitutions as case D1, with "count" = 62;

```
!double loadinsigs funcsigs.(c2 AND c4 AND count = #111110) ->
((WORD6 0), (EL 0 (EL 2 funcsigs), fetchnode)           (HD.2)
```

Whilst the specification delivers,

```
!loadin func.(c2 AND c4 AND count = #111110) ->
(WORD6 0)                                                (SD.2)
```

Case 2 is proven.

PROOF LIMB D3

Expanding down to and including the underlying ADD1 functions,

```
!count double loadinsigs funcsigs.(c2 AND c4 AND
 NOT (count = #111111) AND NOT (count = #111110)) ->
 (WORD6(VAL6(WORD6(VAL6 count + 1)) + 1),
  (EL 0 (BITS2 (EL 2 funcsigs))), fetchnode)
```

The relatively complicated expression for "count" can be simplified using an LCF - LSM axiom for integers < 64;

```
!w.(w < 64) -> VAL6(WORD6 w) == w                    (AX.1)
```

Therefore,

```
!count double loadinsigs funcsigs.(c2 AND c4 AND
 NOT (count = #111111) AND NOT (count = #111110)) ->
 (WORD6(((VAL6 count) + 1) + 1), (EL 0 (BITS2(EL 2 funcsigs))),
  fetchnode)                                          (HD.3)
```

For comparison, the specification requires,

```
!count loadin func.(c2 AND c4 AND NOT (count = #111111) AND
 NOT (count = #111110)) ->
 (WORD6((VAL6 count) + 2))                            (SD.3)
```

By equality in the world of integers, ie the LCF - LSM mode "num", the two expressions for "count" are identical and therefore correspondence between the host machine and the specification has been established for the double increment operation.

ANNEX D CORRESPONDENCE BETWEEN HIGH LEVEL DESIGN AND HOST MACHINE

PROOF LIMB 0, FETCH NODE

This has been documented already, in Section 7.

PROOF LIMB 1, INC1 NODE

The host machine provides,

```
NEXT(count, double, #01, loadin, func) ==
(INC1 count double loadin #01)
```

Expanding INC1,

```
NEXT(count, double, #01, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
LET fetchnode = WORD2 0 IN
LET inc2node  = WORD2 2 IN
(double ->  ((ADD1 count), twice, inc2node) |
            ((ADD1 count), twice, fetchnode))            (H.1)
```

Where,

```
(ADD1 count) == ((VAL6 count) = 63 -> WORD6 0 |
                                      WORD6((VAL6 count) + 1))
```

For the implementation,

```
COUNTLOGIC(count, double, #01, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
((MULTIPLEXER (INCLOGIC count (INCCON #01)) loadin
  (MPLXCON #01)) twice (NEXTNODE #01 func double))
```

In this node, (INCCON #01) = F, (MPLXCON #01) = T and

```
(INCLOGIC count F) == ((VAL6 count) = 63 -> WORD6 0 |
                                            WORD6((VAL6 count) + 1))

(NEXTNODE #01 func double) == (double -> inc2node | fetchnode)
```

Hence,

```
COUNTLOGIC(count, double, #01, loadin, func) ==
(((VAL6 count = 63) -> WORD6 0 | WORD6((VAL6 count) + 1)),
 twice, (double -> inc2node | fetchnode))                 (D.1)
```

Since (H.1) and (D.1) are identical, this case is proved.

PROOF LIMB 2, INC2 NODE

The host machine provides,

```
NEXT(count, double, #10, loadin, func) ==
(INC2 count double loadin  func)
```

Expand INC2,

```
NEXT(count, double, #10, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
((ADD1 count), twice, fetchnode)                    (H.2)
```

From the description of the implementation,

```
COUNTLOGIC(count, double, #10, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) In
LET fetchnode = WORD2 0 IN
((MULTIPLEXER (INCLOGIC count F) loadin T), twice, fetchnode)
                                                    (D.2)
```

But is was shown in PROOF 1 that,

```
(ADD1 count) = (MULTIPLEXER (INCLOGIC count F) loadin T)
```

Therefore, equations (H.2) and (D.2) are identical and the high level design is correct for this node.

PROOF LIMB 3, LOAD NODE

The load operation in the host machine is represented as,

```
NEXT(count, double, #11, loadin, func) ==
(LOAD count double loadin func)
```

Expansion of LOAD gives,

```
NEXT(count, double, #11, loadin, func) ==
LET twice = (EL 0 (BITS2 func)) IN
LET fetchnode = WORD2 0 IN
(loadin, twice, fetchnode)                          (H.3)
```

By simple substitution in the function COUNTLOGIC,

```
COUNTLOGIC(count, double, #11, loadin, func) =
LET twice = (EL 0 (BITS2 func)) IN
LET fetchnode = WORD2 0 IN
((MULTIPLEXER (INCLOGIC count F) loadin F), twice, fetchnode)
                                                         (D.3)
```

The value delivered is (loadin, twice, fetchnode) and therefore the implementation and host machine agree for LOAD.

All four limbs of the high level design to host machine proof have now been completed and by virtue of the contents of Annex C and Annex D the chain of proof, block diagram -> host -> specification has been established.

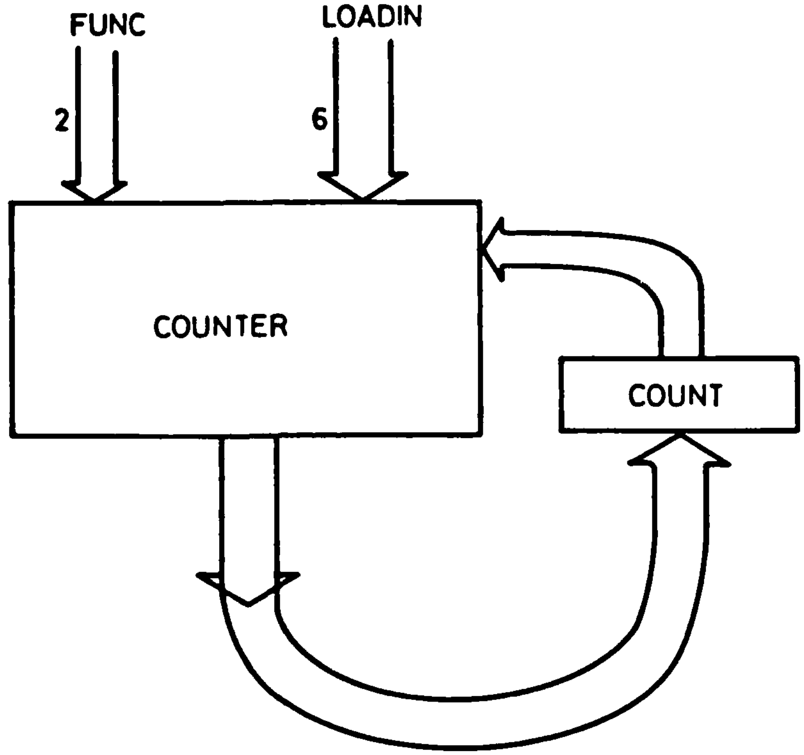


FIG.1 COUNTER, AS A FINITE STATE MACHINE

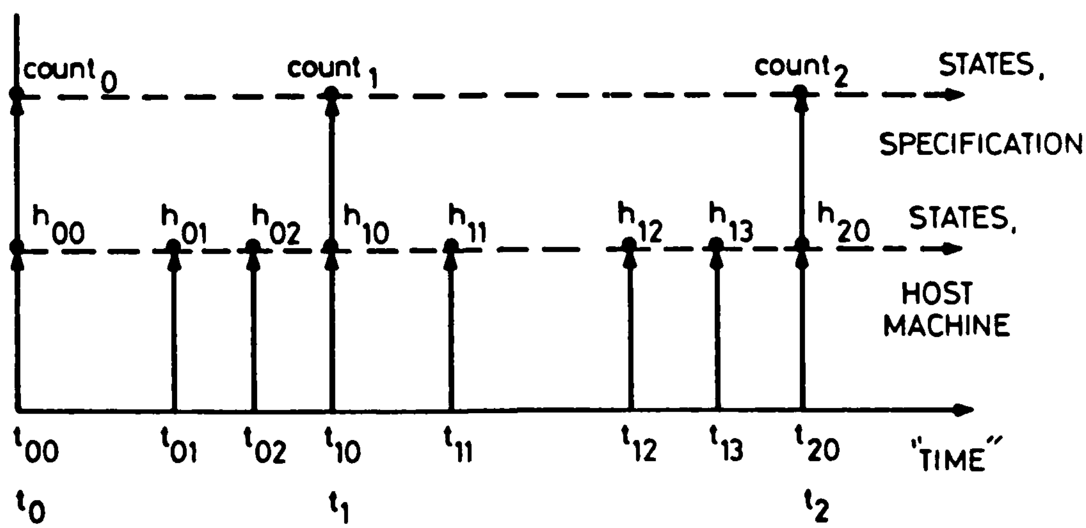


FIG.2 TRANSITIONS AS A FUNCTION OF TIME

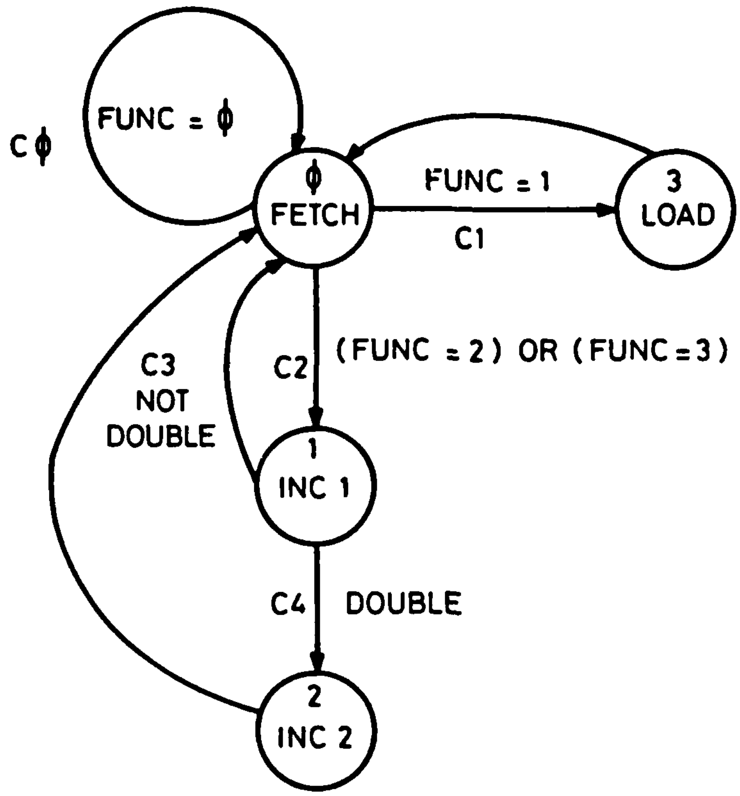


FIG.3 STATE TRANSITIONS IN HOST MACHINE

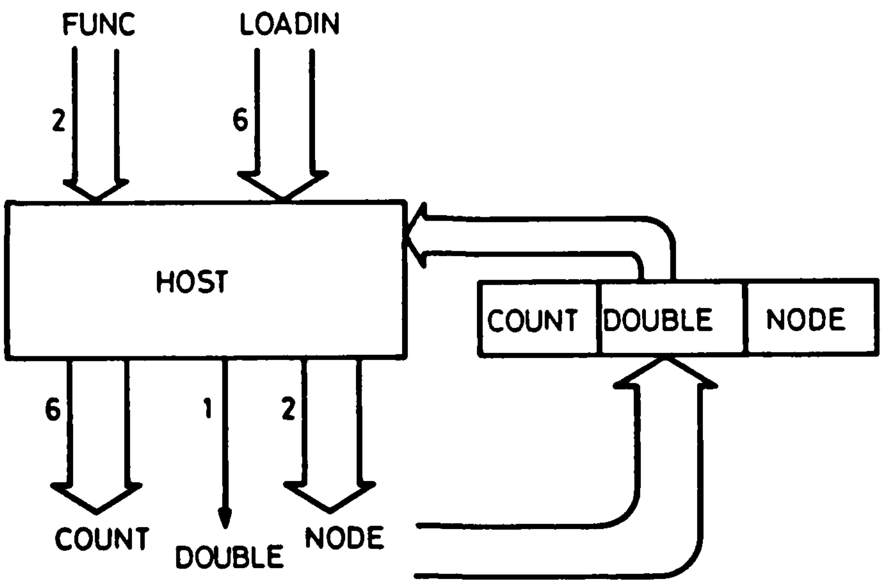


FIG.4 HOST MACHINE

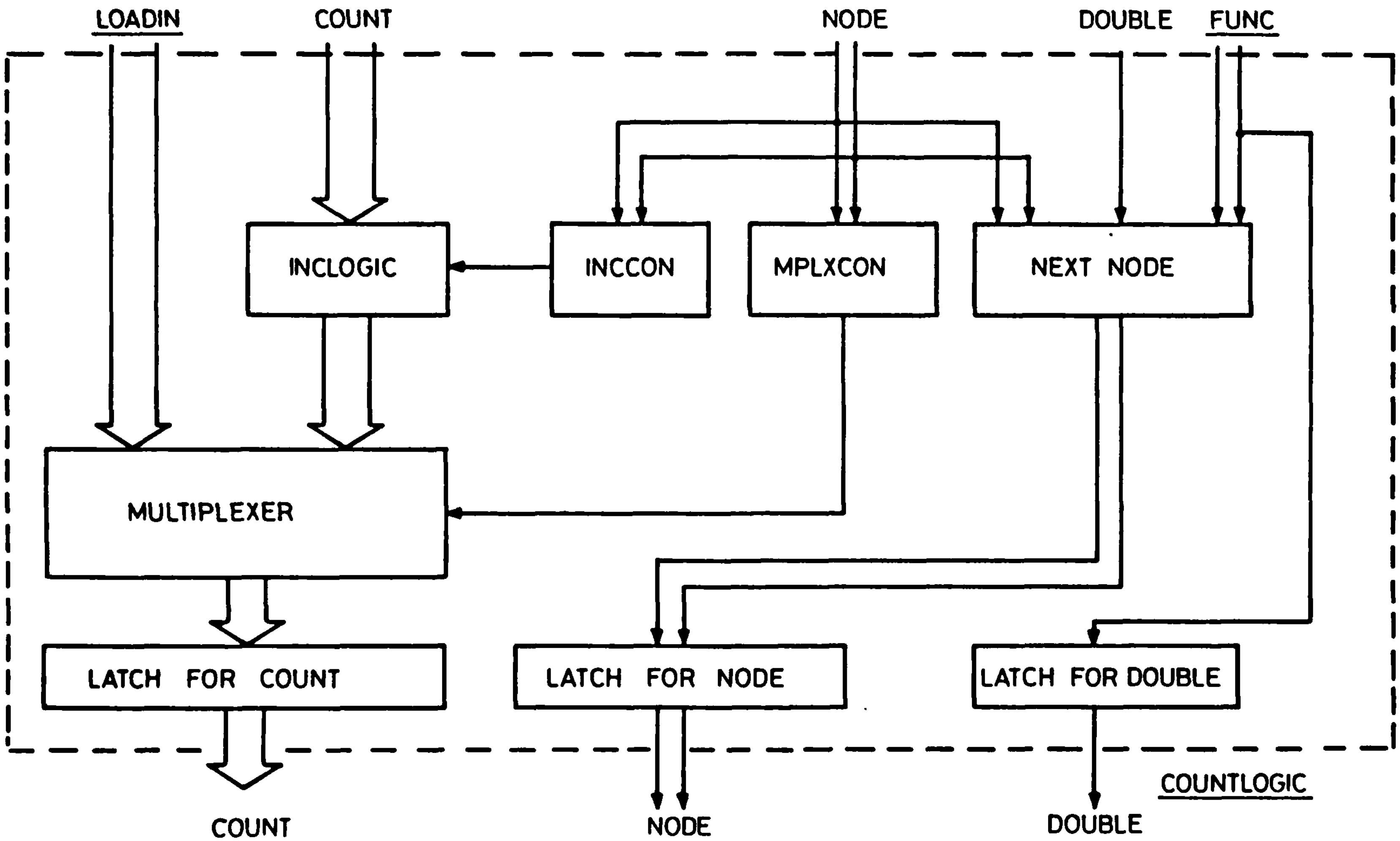


FIG. 5 HIGH LEVEL DESIGN

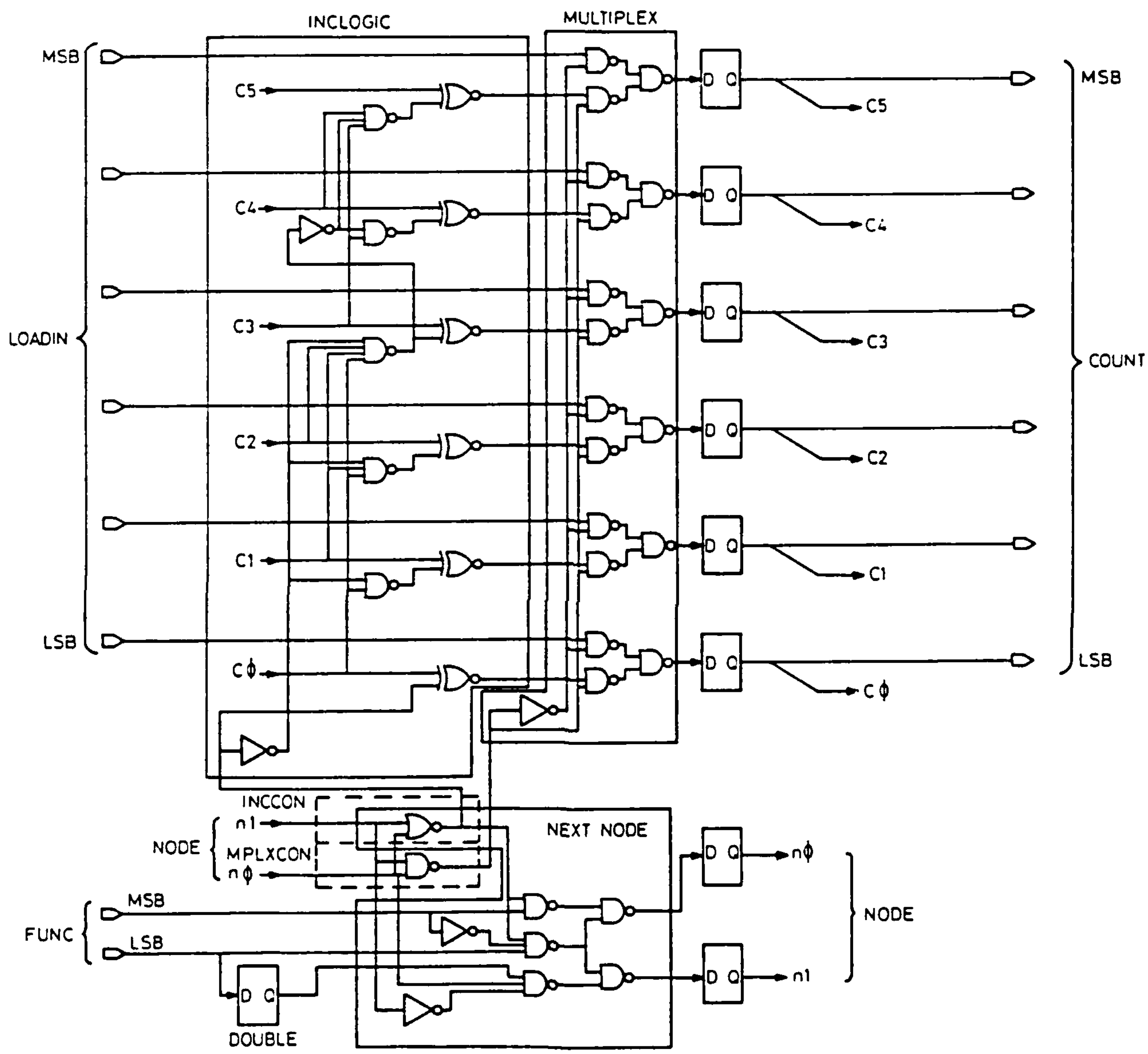


FIG. 6 CIRCUIT DIAGRAM

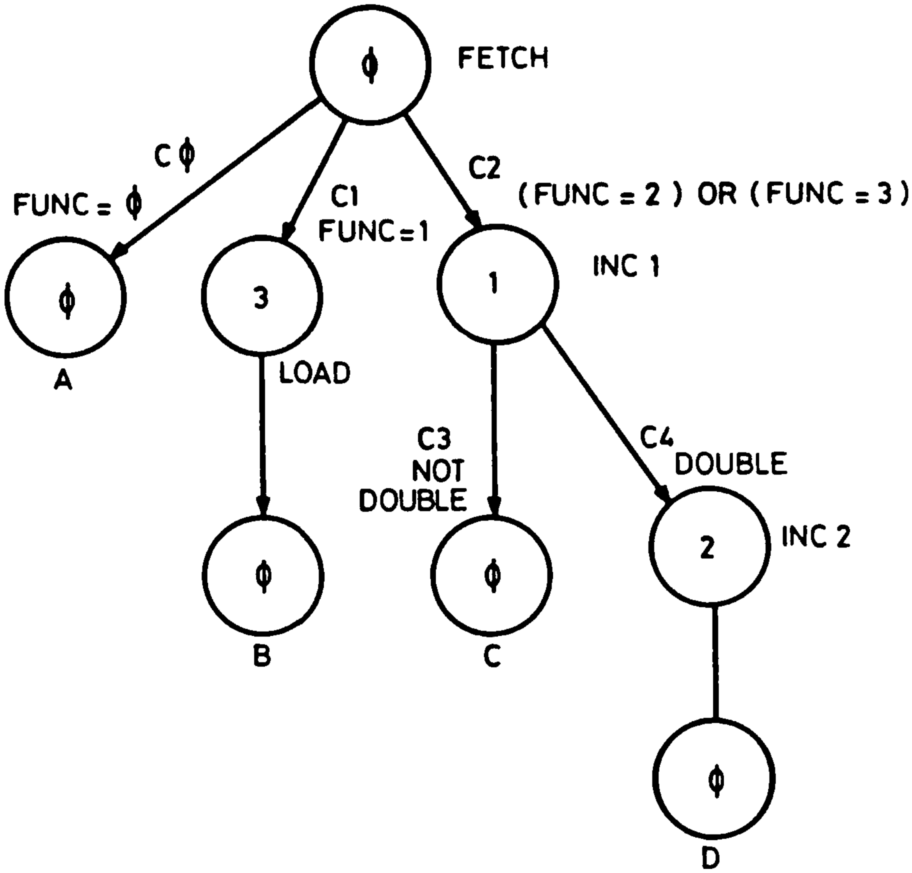


FIG.7 SPANNING TREE FOR HOST MACHINE

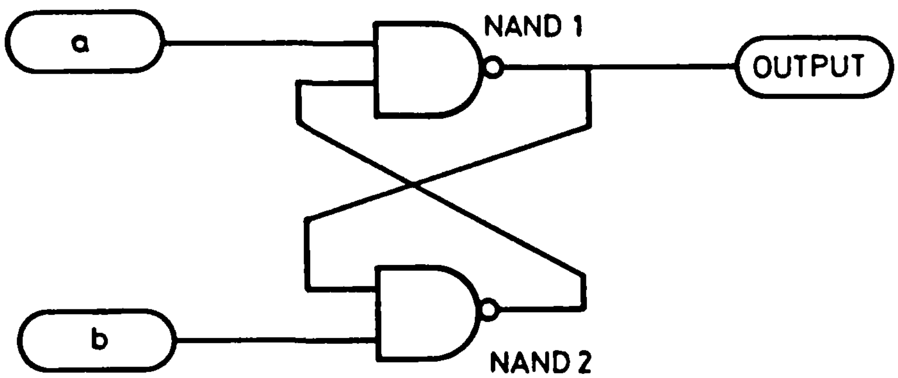


FIG.8 CROSS - COUPLED NAND GATES

ADA 168248

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..........UNCLASSIFIED.................................

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference MEMORANDUM 3832	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS AND RADAR ESTABLISHMENT		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

HARDWARE PROOFS USING LCF- LSM AND ELLA

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
CULLYER, W.J.	PYGOTT, C.H.			

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

HEAD CC GROUP

Descriptors (or keywords)

continue on separate piece of paper

Abstract A method is described for writing the formal specification for a digital system, a high level design which satisfies this requitement and then a gate level realisation, using the languages LCF - LSM and ELLA. Given these formal descriptions, this paper shows how toc arry out mathematical proofs to establish that the high level design respects the specification and the gate level design agrees with precisely with the high level design. By this means, a chain of correspondence proof is created from specification to the ultimate realisation, as required during the development of safety critical and security critical hardware. There methods have been used to develop formal proofs for the VIPER microprocessor and this paper forms an essential tutorial for those studying the validation of this new 32 - bit processor.

S80/48

END

DTIC

7-86