Technical Report

UCAM-CL-TR-94
ISSN 1476-2986

Number 94

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

A mechanized proof of correctness of a simple counter

Avra Cohn, Mike Gordon

July 1986

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

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

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

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

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-94*

July 9, 1986

A Mechanized Proof of Correctness of a Simple Counter

Avra Cohn and Mike Gordon
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG

Abstract

The VIPER microprocessor designed at the Royal Signals and Radar Establishment (RSRE) is probably the first commercially produced computer to have been developed using modern formal methods. Details of VIPER are not in the public domain, but the approach used for its verification is fully explained in RSRE Memorandum 3832 entitled "Hardware Proofs using LCF_LSM and ELLA" by John Cullyer and Clive Pygott. In that report, a simple counter is used to illustrate the verification techniques developed by RSRE. Using the same counter, we illustrate the approach to hardware verification developed at Cambridge. This approach is based on the HOL system, an LCF-style proof generator for higher-order logic.

Contents

1 Introduction

The VIPER microprocessor designed at the Royal Signals and Radar Establishment (RSRE) [Kershaw] is probably the first commercially produced computer to have been developed using modern formal methods. Details of VIPER are not in the public domain, but the approach used for its verification is fully explained in RSRE Memorandum 3832 entitled "Hardware Proofs using LCF_LSM and ELLA" by John Cullyer and Clive Pygott [Cullyer *et al.*]. In that paper, a simple counter is used to illustrate the verification techniques developed by RSRE. Using the same counter, we illustrate the approach to hardware verification developed at Cambridge. This approach is based on the HOL system [Gordon85(a)], an LCF-style proof generator [Gordon *et al.*] for higher-order logic.

In Cullyer and Pygott's paper, the implementation of the counter is specified at three levels of decreasing abstractness.

- As a state-transition system called the *host machine*.
- As an interconnection of functional blocks called the *high-level design*.
- As an interconnection of gates and registers called the *circuit*.

Ultimately, it is the circuit that will be built and it is the correctness of this that is most important. However, the host machine and high-level design represent successive stages in the development of the implementation and one would like to know if they are correct also. It is possible, at least in principle, for the circuit to be correct even if the host machine (and/or high-level design) is wrong; this could happen if the circuit failed to implement the host (and/or high-level design) but did happen to implement the counter. If such were the case, one would want to know.

Cullyer and Pygott formalize the state transitions of the counter and the host machine in LCF_LSM [Gordon83]; they formalize the high-level design in both LCF_LSM and ELLA [Morison *et al.*] and they formalize the circuit in ELLA. They prove informally that that state transitions of the counter are correctly implemented by certain sequences of transitions of the host machine, and that the host machine is equivalent to the high-level design. The verification that the high-level design is correctly implemented by the circuit is done using a simulation technique called "intelligent exhaustion" [Pygott].

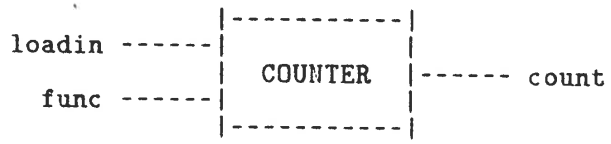
In this paper we formulate and prove a single correctness statement that must be true if the circuit is to implement the counter. The proof consists of three main

lemmas which, respectively, establish the correctness of the host machine (see Sections 3, 4 and 5), the equivalence of the high-level design to the host machine (see Sections 6 and 7), and the equivalence of the circuit to the high-level design (see Sections 8 and 9). These lemmas are combined to yield the correctness of the counter in Section 10.

In what follows, proofs are presented informally; the *formal* proofs (sequences of basic inference steps) have all been generated mechanically by invoking ML procedures. We do not discuss these ML procedures (tactics, conversions *etc.*) in this paper, but they can all be found in the extensively commented Appendix.

2 Specification of the Counter

Cullyer and Pygott's counter is a device with two inputs and one output.



The input lines `loadin` and `func` carry 6-bit and 2-bit words respectively; the output line `count` carries 6-bit words. The counter stores a 6-bit word which can be read via the output line `count`. The counter is a finite state machine; its next state is determined by the value input at `func`.

- `func` = 0: the value stored in the counter is unchanged.
- `func` = 1: the value stored in the counter is the value input at `loadin`.
- `func` = 2: the value stored in the counter is incremented once.
- `func` = 3: the value stored in the counter is incremented twice.

This behaviour can be formalized in the HOL logic by defining a function `COUNTER` taking as arguments the stored value and the two input values, and returning as result the new stored value. Before giving the formal definition of this function we briefly summarize HOL notation.

2.1 Brief Summary of HOL Notation

A *term* of higher-order logic can be one of four kinds.

1. A *variable* (*e.g.* `x`, `x1`, `count`).

2. A *constant* (*e.g.* `1`, `T`, `#011001`).

3. A *function application* of the form t_1t_2 where t_1 is called the *operator* and t_2 the *operand*.

4. An *abstraction* of the form `\x.t` where `x` is called the *bound variable* and `t` the *body* (`\` is the ASCII representation of λ).

To make terms more readable, HOL allows a number of alternative syntactic forms. The term $t_1t_2\cdots t_n$ abbreviates $(\cdots(t_1t_2)\cdots t_n)$. Some two argument function constants can be infixed; for example, t_1`/\`t_2 can be written instead of `/\`t_1t_2. Some terms of the form `f(\x.t)` can be written as `f x.t`; when this is the case `f` is called a *binder*. For example, `!` (the ASCII representation of $\forall$) is a binder, so `!x.t` can be written instead of `!(\x.t)`.

The HOL system uses the ASCII characters `~`, `\/` and `/\`, `==>`, `!` and `?` to represent the logical symbols $\neg$, $\vee$, $\wedge$, $\supset$, $\forall$ and $\exists$, respectively (`\/`, `/\`, `==>` are infixes and `!` and `?` are binders). A term of the form `(t=>`t_1`|`t_2`)` is a *conditional* with *if-part* `t`, *then-part* t_1 and *else-part* t_2; it abbreviates `COND t` t_1 t_2. A term of the form `let x =` t_1 `in` t_2 abbreviates `(\x.`t_2`)`t_1. The two truth-values *true* and *false* are represented by the boolean constant symbols `T` and `F` respectively. Constant terms of the form `#`$b_{n-1}\cdots b_0$ (where b_i is either `0` or `1`) denote n-bit words; b_0 is the least significant bit.

All terms in HOL have a *type*. We write `t:ty` if term `t` has type `ty`; for example, we can write `T:bool` and `F:bool` to indicate that the truth-values `T` and `F` have type `bool`. Numbers have type `num` and n-bit words have type `word`n: for example, `3:num` and `#101101:word6`.

If `ty` is a type then `(ty)list` (also written `ty list`) is the type of lists whose components have type `ty`. If ty_1 and ty_2 are types then ty_1`->`ty_2 is the type of functions whose arguments have type ty_1 and results of type ty_2. The cartesian product operator is represented by `#`, so that ty_1`#`ty_2 is the type of pairs whose first components have type ty_1 and second, ty_2. (The product operator `#` associates to right and binds more tightly than the operator `->`.) For example, the function we

shall use to model the counter has type `word6#word6#word2->word6` which abbreviates `(word6#(word6#word2))->word6`. Types can contain the type variables; these are *, **, *** *etc.* and can be instantiated to any types.

The HOL system has a number of predefined constants. The ones used in this paper are listed below.

- `V:(bool)list->num` converts a list of truth-values to a number.
- `EL:num->((*)list->*)` selects a member of a list (`EL i` $[t_{n-1};\ldots;t_0]$ `=` t_i).
- `VAL`n`:word`n`->num` converts an n-bit word to a number.
- `BIT`n`:word`n`->num->bool` selects a bit of a word (`BIT`n `#`$b_{n-1}\cdots b_0$ `i =` b_i).
- `BITS`n`:word`n`->(bool)list` converts an n-bit word to a list of booleans.
- `WORD`n`:num->word`n converts a number to an n-bit word.
- `TUPLE`n`:(*)list->*#···#*` converts a list of length n to an n-tuple.
- `V`n`:bool#···#bool->word`n converts an n-tuple of truth-values to an n-bit word.
- `VS`n`:(num->bool)#···#(num->bool)->(num->word`n`)` converts an n-tuple of functions to a single n-bit valued function (`VS`$n$$(s_1,\ldots,s_n)$`x = V`$n$$(s_1$`x`$,\ldots,s_n$`x)`).

2.2 HOL Specification of the Counter

The definition of `COUNTER` below is the same, except for minor notational changes, as the one given in Cullyer and Pygott's paper.

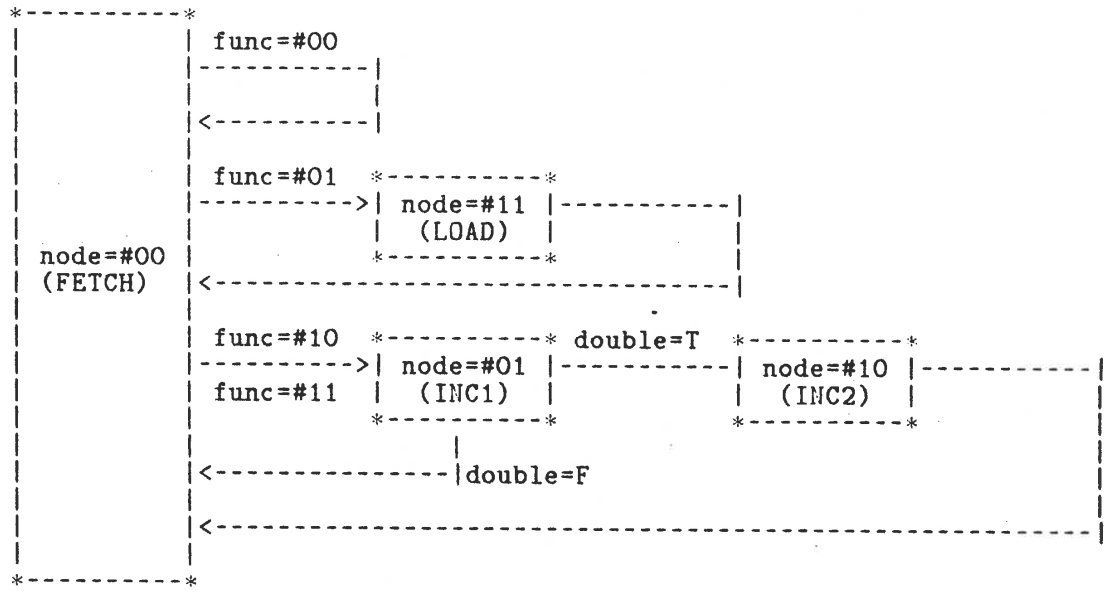
```
COUNTER(count,loadin,func) =
 let funcnum = VAL2 func  in
 let value   = VAL6 count in
 ((funcnum=0) => count                                          |
  (funcnum=1) => loadin                                         |
  (funcnum=2) => ((value=63) => WORD6 0 | WORD6(value+1))       |
  (funcnum=3) => ((value=63) => WORD6 1 |
                  (value=62) => WORD6 0 | WORD6(value+2)) | ARB)
```

The differences between this definition and the one in the RSRE Memorandum are:

1. `let`, `in` and `=` are used instead of `LET`, `IN` and `==` respectively, and
2. the conditional has an extra arm containing an arbitrary value `ARB`. (In both HOL and LCF_LSM, conditionals must always have an else-clause.)

3 Specification of the Host Machine

Cullyer and Pygott explain the behaviour of the host machine with the following state transition diagram.



The state of the host machine is fully specified by a triple `(count,double,node)` where:

- `count` is the 6-bit word being stored by the counter,
- `double` is a 1-bit boolean flag that determines whether the counter is to add one or two and
- `node` is a 2-bit value that determines which of the four nodes in the above diagram the host is in at any particular time.

A state `(count,double,node)` is called *primary* if `node=#00`. The action taken when control reaches a node is represented in HOL by the functions `FETCH`, `LOAD`, `INC1` and `INC2`. These actions are:

1. `FETCH`: `func` determines the next state.
2. `INC1`: `count` is incremented; `double` determines the next state.
3. `INC2`: `count` is incremented; the next state is primary. (A state with `node=#10` is reached only if the counter is to be incremented twice, the first increment being done at the preceding node `#01`).

4. LOAD: count becomes loadin; the next state is primary. (Note that the value stored is the value input when node #11 is reached, *not* the value input during the preceding primary state.)

The HOL functions representing the four nodes are defined below. Some of these definitions make use of an auxiliary function ADD1 for incrementing a 6-bit word. This is defined by

```
ADD1(x) =
let xval = VAL6 x in
((xval=63) => WORD6 0 | WORD6(xval+1))
```

The HOL definitions that follow are based on those on page 6 of RSRE Memorandum 3832.

```
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))

LOAD(count,double,loadin,func) =
 let twice     = EL 0 (BITS2 func) in
 let fetchnode = WORD2 0           in
 (loadin,twice,fetchnode)

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) |
            (ADD1 count,twice,fetchnode))

INC2(count,double,loadin,func) =
 let twice     = EL 0 (BITS2 func) in
 let fetchnode = WORD2 0           in
 (ADD1 count,twice,fetchnode)
```

A state transition function NEXT for the host machine can now be defined.

```
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) | ARB)
```

If the host machine is in state (count,double,node) and the values being input are loadin and func, then the next state of the machine is the value of NEXT((count,double,node),loadin,func).

In the next section we outline what it means for the function NEXT to correctly implement the specification COUNTER.

4 Correctness of the Host Machine

We use the variable state to range over host machine states (*i.e.* triples of the form (count,double,node)). The functions COUNT, DOUBLE and NODE select the first, second and third components of a state respectively. These functions satisfy the equation:

```
!state. state = (COUNT(state),DOUBLE(state),NODE(state))
```

The host machine can take several transitions to implement a single transition of COUNTER. Suppose $\texttt{state}_0$, ... , $\texttt{state}_n$ is a sequence of states such that $\texttt{NODE(state}_0\texttt{)=\#00}$, $\texttt{NODE(state}_n\texttt{)=\#00}$, and $\texttt{NODE(state}_i\texttt{)}$ is not equal to #00 for $0<i<n$. Suppose also that $\texttt{loadin}_0$, ... , $\texttt{loadin}_n$ and $\texttt{func}_0$, ... , $\texttt{func}_n$ are sequences of inputs; then the function NEXT is a correct implementation of COUNTER if whenever

$\texttt{NEXT(state}_0\texttt{,loadin}_0\texttt{,func}_0\texttt{) = state}_1$

$\texttt{NEXT(state}_1\texttt{,loadin}_1\texttt{,func}_1\texttt{) = state}_2$

$\vdots$

$\texttt{NEXT(state}_{n-1}\texttt{,loadin}_{n-1}\texttt{,func}_{n-1}\texttt{) = state}_n$

then

$\texttt{COUNTER(COUNT(state}_0\texttt{),loadin}_1\texttt{,func}_0\texttt{) = COUNT(state}_n\texttt{)}$

Note that the value read in is the value input on the second cycle of the computation (*i.e.* $\texttt{loadin}_1$, not $\texttt{loadin}_0$). Primary states are those states whose NODE-component is equal to #00. The host machine is thus correct if the state transitions specified by COUNTER correspond to those sequences of state transitions specified by NEXT that start and end at primary states and have no primary states in between.

In HOL we can represent the sequence of states $\texttt{state}_0$, $\texttt{state}_1$, ... , $\texttt{state}_n$ by a function state_sig such that $\texttt{state\_sig(0)=state}_0$, $\texttt{state\_sig(1)=state}_1$, *etc.* The function state_sig is a state-valued *signal*; it has type num->(word6#bool#word2). In general, a ty-valued signal is a function of type num->ty where numbers represent time.

A state valued signal state_sig will represent a sequence of states of the implementation if

```
!t. state_sig(t+1) = NEXT(state_sig(t),loadin_sig(t),func_sig(t))
```

The implementation is correct if whenever a primary state occurs at time t (*i.e.* NODE(state_sig(t))=#00) and the *next* time a primary state occurs is at time t', then

```
COUNT(state_sig(t')) =
 COUNTER(COUNT(state_sig(t)),loadin(t+1),func(t))
```

To formalize this, we define the function NEXTTIME such that NEXTTIME(t,t')f is true if and only if t' is the next time after t that f is true. For example, the term NEXTTIME(t,t')(\x.NODE(state_sig x)=#00) is true if and only if t' is the next time after t that the NODE-value of the state is #00. NEXTTIME is defined by

```
NEXTTIME (x1,x2) f =
 (x1 < x2) /\ (f x2) /\ (!x. (x1 < x) /\ (x < x2) ==> ~f x)
```

Using this function, we can formulate the *partial correctness* of the host as

```
(!t. state_sig(t+1) = NEXT(state_sig(t),loadin(t),func(t))) /\
(NODE(state_sig(t))=#00)                                    /\
NEXTTIME(t,t')(\x.NODE(state_sig(x))=#00)
==>
(COUNT(state_sig(t')) =
  COUNTER(COUNT(state_sig(t)),loadin(t+1),func(t)))
```

In addition to this partial correctness we must also prove *termination*, namely that the implementation always returns to a primary state when started in one. This can be expressed as

```
(!t. (state_sig(t+1) = NEXT(state_sig(t),loadin(t),func(t))) /\
(NODE(state_sig(t))=#00)
==>
?t'.NEXTTIME(t,t')(\x.NODE(state_sig(x))=#00)
```

Partial correctness and termination can be combined into a single statement of total correctness.

```
(!t. (state_sig(t+1) = NEXT(state_sig(t),loadin(t),func(t))) /\
(NODE(state_sig(t))=#00)
==>
?t'.NEXTTIME(t,t')(\x.NODE(state_sig(x))=#00) /\
    (COUNT(state_sig(t')) =
      COUNTER(COUNT(state_sig(t)),loadin(t+1),func(t)))
```

In the next section we outline a formal proof of the total correctness of the host machine. Detailed HOL commands for generating this proof are given in the Appendix.

5 Verification of the Host Machine

The goal is to prove that

```
(!t. (state_sig(t+1) = NEXT(state_sig(t),loadin(t),func(t))) /\
(NODE(state_sig(t))=#00)
==>
?t'.NEXTTIME(t,t')(\x.NODE(state_sig(x))=#00) /\
    (COUNT(state_sig(t')) =
      COUNTER(COUNT(state_sig(t)),loadin(t+1),func(t)))
```

The first step is to expand out definitions (and simplify the results) to derive the four lemmas shown below. Each of these lemmas describes the possible state transitions from a node of the host machine; taken together they formalize the complete state transition diagram.

For node #00:

```
NEXT((count,double,#00),loadin,func) =
 count,EL 0(BITS2 func),
 ((VAL2 func = 0) => #00 | ((VAL2 func = 1) => #11 | #01))
```

For node #01:

```
NEXT((count,double,#01),loadin,func) =
 ((VAL6 count = 63) => #000000 | WORD6((VAL6 count)+1)),
 EL 0(BITS2 func),(double => #10 | #00)
```

For node #10:

```
NEXT((count,double,#10),loadin,func) =
 ((VAL6 count = 63) => #000000 | WORD6((VAL6 count)+1)),
 EL 0(BITS2 func),#00
```

For node #11:

```
NEXT((count,double,#11),loadin,func) = loadin,EL 0(BITS2 func),#00
```

Using these lemmas we next derive four theorems of the form

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==>
(NODE(state_sig t) = w) ==>
(COUNT(state_sig(t+1))  = ... )  /\
(DOUBLE(state_sig(t+1)) = ... )  /\
(NODE(state_sig(t+1))   = ... )
```

where "..." stands for terms giving the values of the three components of the state at time t+1 in terms of their values at time t and the values of the inputs loadin and func at time t.

The four theorems below have the form shown above and correspond to the four nodes of the state transition diagram.

For node #00:

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==>
(NODE(state_sig t) = #00) ==>
(COUNT(state_sig(t+1)) = COUNT(state_sig t)) /\
(DOUBLE(state_sig(t+1)) = EL 0(BITS2(func_sig t))) /\
(NODE(state_sig(t+1)) =
 ((VAL2(func_sig t) = 0) =>
  #00 |
  ((VAL2(func_sig t) = 1) => #11 | #01)))
```

For node #01:

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==>
(NODE(state_sig t) = #01) ==>
(COUNT(state_sig(t+1)) =
 ((VAL6(COUNT(state_sig t)) = 63) =>
  #000000 |
  WORD6((VAL6(COUNT(state_sig t)))+1))) /\
(DOUBLE(state_sig(t+1)) = EL 0(BITS2(func_sig t))) /\
(NODE(state_sig(t+1)) = (DOUBLE(state_sig t) => #10 | #00))
```

For node #10:

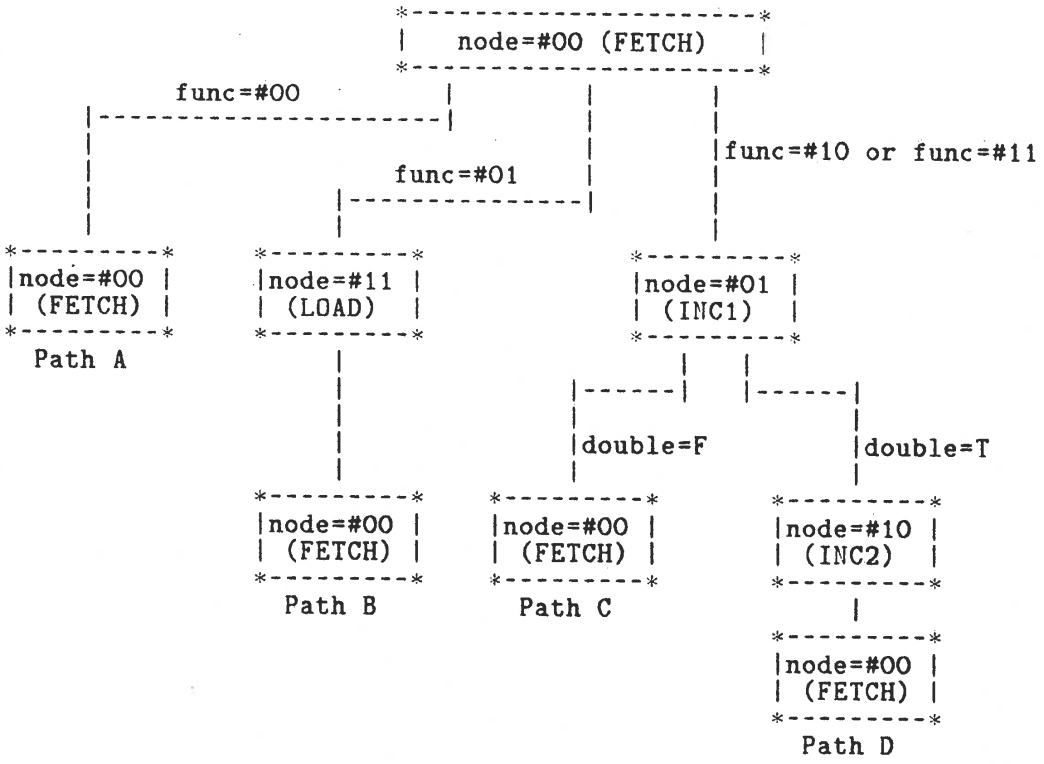
```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==>
(NODE(state_sig t) = #10) ==>
(COUNT(state_sig(t+1)) =
 ((VAL6(COUNT(state_sig t)) = 63) =>
  #000000 |
  WORD6((VAL6(COUNT(state_sig t)))+1))) /\
(DOUBLE(state_sig(t+1)) = EL 0(BITS2(func_sig t))) /\
(NODE(state_sig(t+1)) = #00)
```

For node #11:

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==>
(NODE(state_sig t) = #11) ==>
(COUNT(state_sig(t+1)) = loadin_sig t) /\
(DOUBLE(state_sig(t+1)) = EL 0(BITS2(func_sig t))) /\
(NODE(state_sig(t+1)) = #00)
```

These theorems describe single state transitions of the host. We use them to derive descriptions of the effect of each complete path through the the state transition diagram.

The "spanning tree" shown below is based on Figure 7 of RSRE Memorandum 3832 and shows the four complete paths through the state transition diagram.



The four paths through this tree are called (following RSRE Memorandum 3832) A, B, C and D. The following four theorems correspond to these paths.

For Path A:

```
(NODE(state_sig t) = #00) ==>
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==>
(func_sig t = #00) ==>
(COUNT(state_sig(t+1)) = COUNT(state_sig t)) /\
(DOUBLE(state_sig(t+1)) = F) /\
(NODE(state_sig(t+1)) = #00)
```

For Path B:

```
(NODE(state_sig t) = #00) ==>
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==>
(func_sig t = #01) ==>
((COUNT(state_sig(t+1)) = COUNT(state_sig t)) /\
 (DOUBLE(state_sig(t+1)) = T) /\
 (NODE(state_sig(t+1)) = #11)) /\
(COUNT(state_sig((t+1)+1)) = loadin_sig(t+1)) /\
(DOUBLE(state_sig((t+1)+1)) = EL 0(BITS2(func_sig(t+1)))) /\
(NODE(state_sig((t+1)+1)) = #00)
```

For Path C:

```
(NODE(state_sig t) = #00) ==>
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==>
(func_sig t = #10) ==>
((COUNT(state_sig(t+1)) = COUNT(state_sig t)) /\
 (DOUBLE(state_sig(t+1)) = F) /\
 (NODE(state_sig(t+1)) = #01)) /\
(COUNT(state_sig((t+1)+1)) =
 ((VAL6(COUNT(state_sig t)) = 63) =>
  #000000 |
  WORD6((VAL6(COUNT(state_sig t)))+1))) /\
(DOUBLE(state_sig((t+1)+1)) = EL 0(BITS2(func_sig(t+1)))) /\
(NODE(state_sig((t+1)+1)) = #00)
```

and for path D the rather complex theorem:

```
(NODE(state_sig t) = #00) ==>
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==>
(func_sig t = #11) ==>
((COUNT(state_sig(t+1)) = COUNT(state_sig t)) /\
 (DOUBLE(state_sig(t+1)) = T) /\
 (NODE(state_sig(t+1)) = #01)) /\
((COUNT(state_sig((t+1)+1)) =
  ((VAL6(COUNT(state_sig t)) = 63) =>
   #000000 |
   WORD6((VAL6(COUNT(state_sig t)))+1))) /\
 (DOUBLE(state_sig((t+1)+1)) = EL 0(BITS2(func_sig(t+1)))) /\
 (NODE(state_sig((t+1)+1)) = #10)) /\
(COUNT(state_sig(((t+1)+1)+1)) =
 ((VAL6
   ((VAL6(COUNT(state_sig t)) = 63) =>
    #000000 |
    WORD6((VAL6(COUNT(state_sig t)))+1)) =
   63) =>
  #000000 |
  WORD6
  ((VAL6
    ((VAL6(COUNT(state_sig t)) = 63) =>
     #000000 |
     WORD6((VAL6(COUNT(state_sig t)))+1))) +
   1))) /\
(DOUBLE(state_sig(((t+1)+1)+1)) =
 EL 0(BITS2(func_sig((t+1)+1)))) /\
(NODE(state_sig(((t+1)+1)+1)) = #00)
```

Using these theorems, along with the definition of NEXTTIME and some elementary theorems of arithmetic, we can derive four theorems that give the time taken to traverse the four spanning tree paths.

For Path A:

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t) = #00) /\
(func_sig t = #00) ==>
NEXTTIME(t,t+1)(\x. NODE(state_sig x) = #00)
```

For Path B:

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t) = #00) /\
(func_sig t = #01) ==>
NEXTTIME(t,(t+1)+1)(\x. NODE(state_sig x) = #00)
```

For Path C:

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t) = #00) /\
(func_sig t = #10) ==>
NEXTTIME(t,(t+1)+1)(\x. NODE(state_sig x) = #00)
```

For Path D:

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t) = #00) /\
(func_sig t = #11) ==>
NEXTTIME(t,((t+1)+1)+1)(\x. NODE(state_sig x) = #00)
```

The host machine terminates if

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t) = #00)
==>
(?t'. NEXTTIME(t,t')(\x. NODE(state_sig x) = #00))
```

This follows easily by case analysis on the term `func_sig t`, followed by rewriting using the four path theorems.

To prove correctness we need the (intuitively obvious) theorem that for any `f` and `x1` there is a unique `x2` such that `NEXTTIME(x1,x2)f`. Formally:

```
!f x1 x2. NEXTTIME(x1,x2)f /\ NEXTTIME(x1,x3)f ==> (x2 = x3)
```

From this fact and the path lemmas it easily follows that

```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t)=#00)                                          /\
NEXTTIME (t,t') (\x. NODE(state_sig x)=#00)
==>
(t' = ((func_sig t = #00) => t+1          |
       (func_sig t = #01) => (t+1)+1      |
       (func_sig t = #10) => (t+1)+1      |
       (func_sig t = #11) => ((t+1)+1)+1 | ARB))
```

Using this theorem, the definition of COUNTER, the path theorems, some case analysis and some arithmetic, we can deduce the partial correctness of the host machine, namely

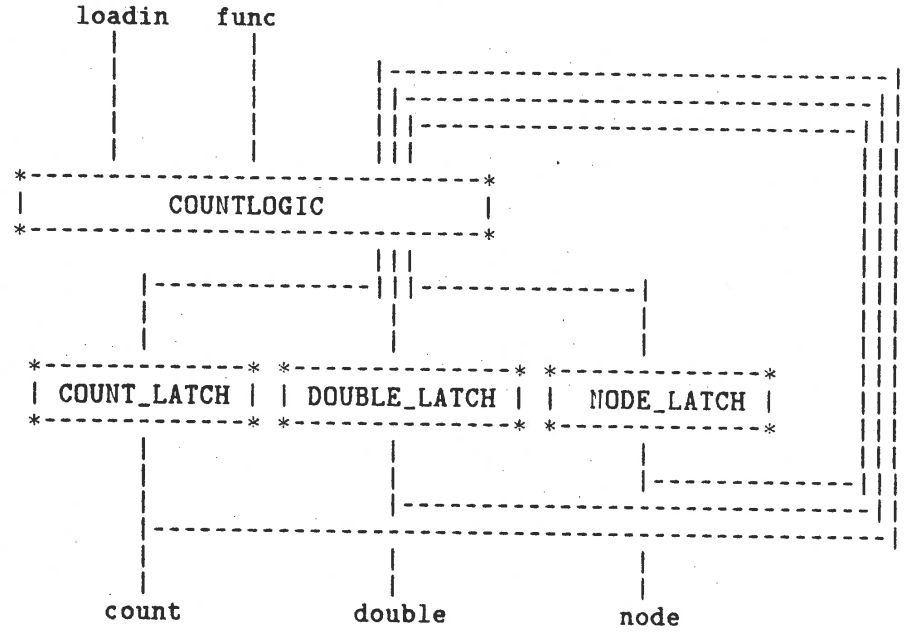
```
(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t)=#00)                                          /\
NEXTTIME (t,t') (\x. NODE(state_sig x)=#00)
==>
(COUNT(state_sig t') =
  COUNTER(COUNT(state_sig t),loadin_sig(t+1),func_sig t))
```

The total correctness of the host machine follows easily from its partial correctness and termination.

The large number of omitted details of this proof can be found in the Appendix.

6 Specification of the High Level Design

The high level design consists of the three registers COUNT_LATCH, DOUBLE_LATCH and NODE_LATCH together with some combinational control logic called COUNTLOGIC.



A state of the high level design is a triple (count,double,node) where count is the 6-bit word stored in the register COUNT_LATCH, double is the boolean value stored in DOUBLE_LATCH and node a 2-bit word stored in NODE_LATCH. If the high level design is in state (count,double,node) and loadin and func are being input, then the next state is (count',double',node') if

```
COUNTLOGIC((count,double,node),loadin,func) = (count',double',node')
```

The actual definition of COUNTLOGIC is

```
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))
```

where the functions MULTIPLEX, INCLOGIC, INCCON, MPLXCON and NEXTNODE are specified by the following HOL definitions:

```
MULTIPLEX(incout,loadin,mplxsel) =
 (mplxsel => incout | loadin)

INCLOGIC(count,noinc) =
 let countval = VAL6 count in
 (noinc => count | ((countval=63) => WORD6 0 | WORD6(countval+1)))

INCCON(node) = (VAL2 node = 0)

MPLXCON(node) = ~(VAL2 node = 3)

NEXTNODE(node,func,double) =
 let funcnum   = VAL2 func in
 let nodenum   = VAL2 node in
 let fetchnode = WORD2 0   in
 let inc1node  = WORD2 1   in
 let inc2node  = WORD2 2   in
 let loadnode  = WORD2 3   in
 ((nodenum=0) => ((funcnum=0) => fetchnode |
                  (funcnum=1) => loadnode  | inc1node) |
  (nodenum=1) => (double => inc2node | fetchnode)      |
                 fetchnode)
```

In the next section we outline the proof that the host machine is correctly implemented by the high level design. This can be expressed very simply by

```
NEXT = COUNTLOGIC
```

7 Verification of the High Level Design

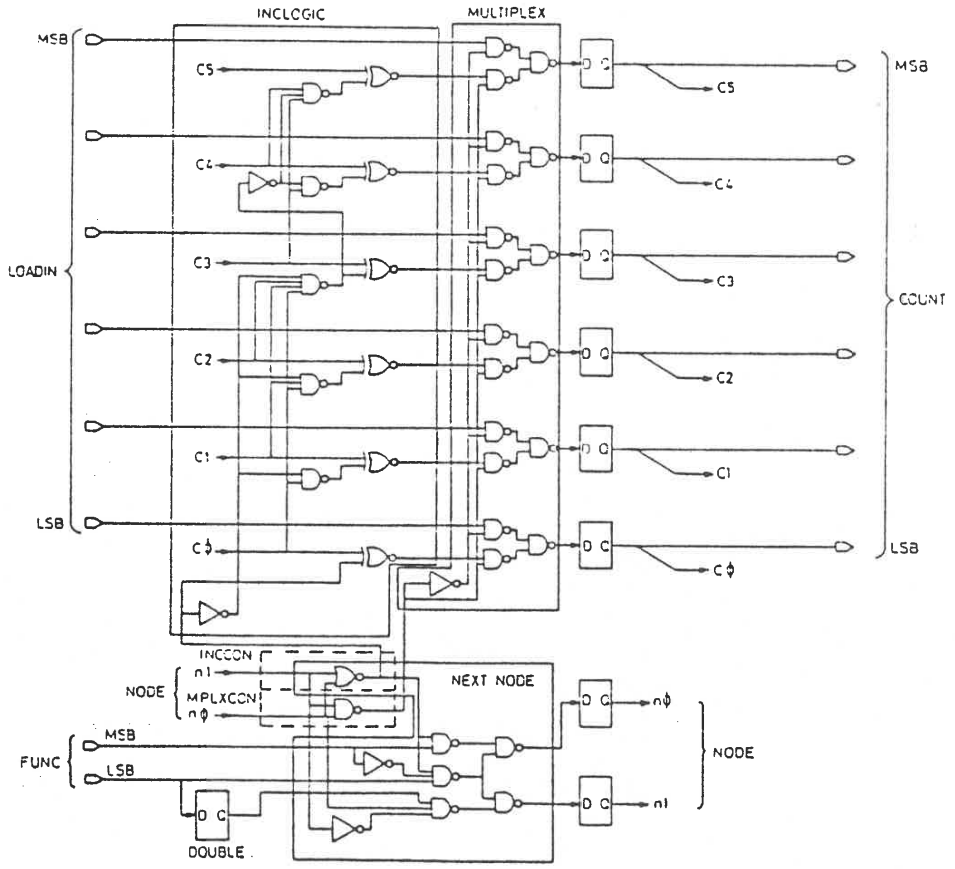
To prove NEXT = COUNTLOGIC it is sufficient (by extensionality) to establish

```
!count double node loadin func.
  NEXT((count,double,node),loadin,func) =
   COUNTLOGIC((count,double,node),loadin,func)
```

This is easily proved by expanding out the definitions and then doing case analysis on the terms node, func and double.

8 Specification of the Circuit

The counter circuit diagram below is reproduced from RSRE Memorandum 3832.



We prove correct two representations of this circuit. The first one (which we call the logical specification) is a direct translation into logic of the circuit diagram. The second one is a translation of the ELLA text given in Memorandum 3832.

8.1 The Logical Specification

The combinational behaviour of the circuit `COUNTLOGIC` is determined by the behaviour of its five component parts, `INCCON`, `MPLXCON`, `MULTIPLEX`, `INCLOGIC` and `NEXTNODE`, whose behavioural specifications are given in Section 6. For each of these parts there is an implementation in terms of gates as shown in the circuit diagram.

To describe the circuit, we first define gate constants `INV`, `NAND`, and so on, as logical relations between inputs and outputs:

```
NOR(in1,in2,out) = (out = ~(in1 \/ in2))
```

```
NAND(in1,in2,out) = (out = ~(in1 /\ in2))

NAND3(in1,in2,in3,out) = (out = ~(in1 /\ in2 /\ in3))

NAND4(in1,in2,in3,in4,out) = (out = ~(in1 /\ in2 /\ in3 /\ in4))

INV(in,out) = (out = ~in)

XNOR(in1,in2,out) = (out = (in1 = in2))
```

Then, for example, we can define the predicate INCLOGIC_IMP (the implementation of INCLOGIC) in terms of these gate constants. We let c0,...,c5 represent the boolean input values to the circuit, as in the circuit diagram, and b the boolean value sent to INCLOGIC by INCCON; c0',...,c5' represent the boolean values output by INCLOGIC_IMP (and sent on to MULTIPLEX); c0 and c0' are the least significant bits. INCLOGIC_IMP holds between c0,...,c5, b and c0',...,c5'. We then introduce variables b', x1, x2, x3, x3', x4 and x5 that, respectively and from bottom to top in the circuit, name the internal lines emanating from the inverter, the first NAND gate, the second NAND gate, the third NAND gate, the second inverter, the fourth NAND gate and the fifth NAND gate. In the description, we existentially quantify over these new variables. (A full explanation of this technique of description is given in another paper [Gordon85(b)].)

```
INCLOGIC_IMP(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
 ?b' x1 x2 x3 x3' x4 x5.
  INV(b,b')                /\
  INV(x3,x3')              /\
  NAND(b',c0,x1)           /\
  NAND3(b',c0,c1,x2)       /\
  NAND4(b',c0,c1,c2,x3)    /\
  NAND(x3',c3,x4)          /\
  NAND3(x3', c3,c4,x5)     /\
  XNOR(c0,b,c0')           /\
  XNOR(c1,x1,c1')          /\
  XNOR(c2,x2,c2')          /\
  XNOR(c3,x3,c3')          /\
  XNOR(c4,x4,c4')          /\
  XNOR(c5,x5,c5')
```

The other four components are treated similarly; INCLOGIC is the most complex of the five. The analogous descriptions are called INCCON_IMP, MPLXCON_IMP, MULTIPLEX_IMP and NEXTNODE_IMP. Further details of the others appear in the Appendix.

Finally, we relate the definition of the whole combinational part of the counter to the whole implementation. In Section 6 the high level design was specified with the function COUNTLOGIC, defined by

```
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))
```

The predicate COUNTLOGIC_IMP defined below formalizes the implementation of the combinational part of the high level design. It takes as arguments all of the boolean inputs to the circuit: c0,...,c5, and l0,...,l5, and n0 and n1 as in the circuit diagram, as well as f0 and f1, jointly called FUNC in the circuit, and double, extracted from from FUNC in the circuit. It also takes the various outputs as arguments: c0',...,c5', corresponding to c0',...,c5'; n0' and n1' corresponding to n0 and n1; and double' corresponding to double.

As we have done for the components, we describe the whole circuit with a predicate COUNTLOGIC_IMP by giving names to the internal lines, and existentially quantifying over these names; in this case, we need b1 for the boolean output of MPLXCON, b2 for the boolean output of INCCON, and d0,...,d5 for the boolean outputs of INCLOGIC (with d0 the least significant bit). The description of the circuit is

```
COUNTLOGIC_IMP(c0,c1,c2,c3,c4,c5,n0,n1,l0,l1,l2,l3,l4,l5,f0,f1,double,
               c0',c1',c2',c3',c4',c5',n0',n1',double') =
 ?b1 b2 d0 d1 d2 d3 d4 d5.
  INCCON_IMP(n0,n1,b2)                                        /\
  MPLXCON_IMP(n0,n1,b1)                                       /\
  NEXTNODE_IMP(n0,n1,f0,f1,double,n0',n1')                    /\
  INCLOGIC_IMP(c0,c1,c2,c3,c4,c5,b2,d0,d1,d2,d3,d4,d5)        /\
  MULTIPLEX_IMP(d0,d1,d2,d3,d4,d5,l0,l1,l2,l3,l4,l5,b1,
                c0',c1',c2',c3',c4',c5')                      /\
  (double' = f0)
```

where INCLOGIC_IMP and the four other "_IMP" predicates are defined in terms of gate constants, as explained above.

8.2 The ELLA Specification

In RSRE Memorandum 3832 the circuit is specified in ELLA. This specification is easily translated into higher-order logic (and gives an alternative formalization to the specification INCLOGIC_IMP in the previous section). For example, the ELLA definition of INCLOGIC is called INCCIRC in the RSRE Memorandum.

```
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], carrybar).
      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.
```

To produce an HOL equivalent, we define new gate constants which are functional rather than relational as in Section 8.1 (in lower case, to distinguish)

```
nor(in1,in2) =  ~(in1 \/ in2)

nand(in1,in2) =  ~(in1 /\ in2)

nand3(in1,in2,in3) =  ~(in1 /\ in2 /\ in3)

nand4(in1,in2,in3,in4) =  ~(in1 /\ in2 /\ in3 /\ in4)

inv in = ~in

xnor(in1,in2) = (in1 = in2)
```

(we use nand rather than nand2 to be consistent with NAND).

In the translation of the definition of INCCIRC to HOL, the ELLA phrase count[5], for example, becomes the HOL term BIT6 count 5. We use the HOL let...in construct for ELLA's repeated LET construction, so that the ELLA keyword OUTPUT is not necessary. A coercion, V6, is added to correct the type of the result, and the normal bit order is used. The HOL t:ty notation replaces ELLA's ty:t. The HOL definition of INCCIRC is thus

```
INCCIRC count noinc =
 let noincbar = inv noinc in
 let ic1 = xnor(BIT6 count 1,noinc) in
 let ic2 = xnor(BIT6 count 2,nand(noincbar,BIT6 count 1)) in
 let ic3 = xnor(BIT6 count 3,nand3(noincbar,BIT6 count 1,BIT6 count 2)) in
 let carry4bar = nand4(noincbar,BIT6 count 1,BIT6 count 2,BIT6 count 3) in
 let ic4 = xnor(BIT6 count 4,carry4bar) in
 let carry4 = inv carry4bar in
 let ic5 = xnor(BIT6 count 5,nand(carry4,BIT6 count 4)) in
 let ic6 = xnor(BIT6 count 6,nand3(carry4,BIT6 count 4,BIT6 count 5)) in
 V6(ic6,ic5,ic4,ic3,ic2,ic1)
```

The other four components have ELLA definitions which can be translated in a similar way; further details of the translation process and the other components can be found in the Appendix.

9 Verification of the Circuit

We now verify that the circuit meets the two specifications given in the previous section.

9.1 Verification of the Logical Specification

Our first goal is to relate the predicates such as INCLOGIC_IMP from Section 8.1, which together represent the circuit, to the functions such as INCLOGIC, which specify the components of the high level design. The definition of INCLOGIC was

```
INCLOGIC(count,noinc) =
 let countval = VAL6 count
 in (noinc => count | ((countval=63)=>WORD6 0|WORD6(countval+1)))
```

INCLOGIC takes a 6-bit word, count, represented for INCLOGIC_IMP by the booleans c0,...,c5, and a boolean, noinc, represented for INCLOGIC_IMP by b, and returns a 6-bit word, represented for INCLOGIC_IMP by the booleans c0',...,c5'. To relate these two rather different representations, it is convenient to introduce an intermediate level, a *relational specification*, which is a predicate we shall call INCLOGIC_SPEC.

```
INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
 (c5',c4',c3',c2',c1',c0') =
  (b                                    => (c5,c4,c3,c2,c1,c0) |
   ((c5,c4,c3,c2,c1,c0)=(T,T,T,T,T,T)) => (F,F,F,F,F,F)       |
   TUPLE6(BITS6(V6(c5,c4,c3,c2,c1,c0)+1)))
```

INCLOGIC_SPEC, with the same type as INCLOGIC_IMP, holds of its (thirteen) arguments if the outputs, c0',...,c5', are equal to a certain function of the inputs, b and c0,...,c5. That function, on inspection, is really just INCLOGIC phrased in terms of separate boolean values. We then show that

```
INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
 (V6(c5',c4',c3',c2',c1',c0') =
  INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b))
```

and

```
INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
 INCLOGIC_IMP (c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5')
```

and thereby relate INCLOGIC to INCLOGIC_IMP. The first goal is proved by unwinding the definitions, doing case analysis on the boolean b, and then case analysis on whether (c5,c4,c3,c2,c1,c0) = (T,T,T,T,T,T). The second goal is proved similarly, but with a further analysis of the sixty-four possible combinations of the booleans c0, c1, c2, c3, c4 and c5. Analogous results are proved for INCCON, MPLXCON, MULTIPLEX and NEXTNODE.

These proofs require some simple facts about bit strings and boolean lists, such as

```
~(V[b0;b1;b2;b3;b4;b5]=63) ==>
  (V(BITS6(WORD6(V[b0;b1;b2;b3;b4;b5]+1))) =
  (V[b0;b1;b2;b3;b4;b5]+1))
```

To tie these results together, we make an intermediate level relational specification of COUNTLOGIC, just as we have done for each of its components.

```
COUNTLOGIC_SPEC(c0,c1,c2,c3,c4,c5,n0,n1,l0,l1,l2,l3,l4,l5,f0,f1,double,
                c0',c1',c2',c3',c4',c5',n0',n1',double') =
?b1 b2 d0 d1 d2 d3 d4 d5.
  INCCON_SPEC(n0,n1,b2)                                        /\
  MPLXCON_SPEC(n0,n1,b1)                                       /\
  NEXTNODE_SPEC(n0,n1,f0,f1,double,n0',n1')                    /\
  INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b2,d0,d1,d2,d3,d4,d5)        /\
  MULTIPLEX_SPEC(d0,d1,d2,d3,d4,d5,l0,l1,l2,l3,l4,l5,b1,
                 c0',c1',c2',c3',c4',c5')                      /\
  (double' = f0)
```

This definition of COUNTLOGIC_SPEC is very like that of COUNTLOGIC_IMP (in Section 8.1) except that it is in terms of the intermediate level specifications of the components rather than the circuit level definitions. We show that

```
COUNTLOGIC_SPEC(c0,c1,c2,c3,c4,c5,n0,n1,l0,l1,l2,l3,l4,l5,f0,f1,double,
                c0',c1',c2',c3',c4',c5',n0',n1',double') =
((V6(c5',c4',c3',c2',c1',c0'),double',V2(n1',n0')) =
 COUNTLOGIC((V6(c5,c4,c3,c2,c1,c0),double,V2(n1,n0)),
            V6(l5,l4,l3,l2,l1,l0),
            V2(f1,f0)))
```

and

```
COUNTLOGIC_SPEC(c0,c1,c2,c3,c4,c5,n0,n1,l0,l1,l2,l3,l4,l5,f0,f1,double,
                c0',c1',c2',c3',c4',c5',n0',n1',double') =
 COUNTLOGIC_IMP(c0,c1,c2,c3,c4,c5,n0,n1,l0,l1,l2,l3,l4,l5,f0,f1,double,
                c0',c1',c2',c3',c4',c5',n0',n1',double')
```

so that by transitivity

```
COUNTLOGIC_IMP (c0,c1,c2,c3,c4,c5,n0,n1,l0,l1,l2,l3,l4,l5,f0,f1,double,
                c0',c1',c2',c3',c4',c5',n0',n1',double') =
 ((V6(c5',c4',c3',c2',c1',c0'),double',V2(n1',n0')) =
  COUNTLOGIC((V6(c5,c4,c3,c2,c1,c0),double,V2(n1,n0)),
             V6(l5,l4,l3,l2,l1,l0),
             V2(f1,f0)))
```

The first goal is proved by unwinding definitions, and using the five theorems (for the five component parts) stating that their definitions are related to their specifications. The second goal is proved similarly, using the theorems equating the specifications and implementations.

9.2 Verification of the ELLA Specification

Given the HOL version of the ELLA definition, we wish to prove, for example, that

```
INCLOGIC_IMP(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
 (V6(c5',c4',c3',c2',c1',c0') = INCCIRC(V6(c5,c4,c3,c2,c1,c0)))
```

The proof is by unwinding definitions and doing a case analysis on the boolean term `b`. Some simple lemmas about bit strings and boolean lists are used again. The statements and proofs for the other four component are similar, with case analysis on appropriate boolean terms, such as `f0`, `f1`, `n0`, and `n1` as necessary. The ELLA definitions given in RSRE Memorandum 3832 for `MPLXCON`, `INCCON` and `NEXTNODE` are combined into a single definition (as these three components overlap in the circuit diagram), so the equivalence statements must select out the appropriate parts. Details are given in the Appendix.

10 Correctness of the Counter Implementation

To complete the proof of the counter, we must tie together the results about the combinational behaviour of `COUNTLOGIC` with the results about the temporal behaviour (correctness and termination) of `NEXT`. We have already shown that `COUNTLOGIC` and `NEXT` are the same, in Section 7. We have also shown, in Section 5, that

```
(!t.state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t)=#00)                                         /\
NEXTTIME (t,t') (\x. NODE(state_sig x)=#00)
==>
(COUNT(state_sig t') =
  COUNTER(COUNT(state_sig t),loadin_sig(t+1),func_sig t))
```

and

```
(!t.state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t) = #00)
==>
?t'.NEXTTIME(t,t')(\x.NODE(state_sig x)=#00)
```

so we simply substitute to get statements of the partial correctness and termination of COUNTLOGIC:

```
(!t.state_sig(t+1) = COUNTLOGIC(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t)=#00)                                                /\
NEXTTIME (t,t') (\x. NODE(state_sig x)=#00)
==>
(COUNT(state_sig t') =
  COUNTER(COUNT(state_sig t),loadin_sig(t+1),func_sig t))
```

and

```
(!t.state_sig(t+1) = COUNTLOGIC(state_sig t,loadin_sig t,func_sig t)) /\
(NODE(state_sig t) = #00)
==>
?t'.NEXTTIME(t,t')(\x.NODE(state_sig x)=#00)
```

To relate COUNTLOGIC to the temporal behaviour of the circuit, we first define COUNTLOGIC_DEV (for device) which is like COUNTLOGIC but takes signals as arguments

```
COUNTLOGIC_DEV
(c0_sig,c1_sig,c2_sig,c3_sig,c4_sig,c5_sig,n0_sig,n1_sig,
 l0_sig,l1_sig,l2_sig,l3_sig,l4_sig,l5_sig,f0_sig,f1_sig,double_sig,
 c0'_sig,c1'_sig,c2'_sig,c3'_sig,c4'_sig,c5'_sig,
 n0'_sig,n1'_sig,double'_sig) =
!t:num. COUNTLOGIC_IMP
 (c0_sig t,c1_sig t,c2_sig t,c3_sig t,c4_sig t,c5_sig t,
  n0_sig t,n1_sig t,
  l0_sig t,l1_sig t,l2_sig t,l3_sig t,l4_sig t,l5_sig t,
  f0_sig t,f1_sig t,double_sig t,
  c0'_sig t,c1'_sig t,c2'_sig t,c3'_sig t,c4'_sig t,c5'_sig t,
  n0'_sig t,n1'_sig t,double'_sig t)
```

We then introduce the definition of a LATCH

```
LATCH(in,out)  =  !t. out(t+1) = in t
```

which enables us to give a temporal description of the entire circuit. To do this, we define a predicate CIRCUIT_IMP to represent the entire circuit diagram in HOL.

```
CIRCUIT_IMP
(((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),
 double_sig,
 (n1_sig,n0_sig)),
(l5_sig,l4_sig,l3_sig,l2_sig,l1_sig,l0_sig),
(f1_sig,f0_sig)) =
?c0'_sig  c1'_sig  c2'_sig  c3'_sig  c4'_sig  c5'_sig double'_sig
 n0'_sig  n1'_sig.
 (COUNTLOGIC_DEV
  (c0_sig,c1_sig,c2_sig,c3_sig,c4_sig,c5_sig,n0_sig,n1_sig,
   l0_sig,l1_sig,l2_sig,l3_sig,l4_sig,l5_sig,f0_sig,f1_sig,double_sig,
   c0'_sig,c1'_sig,c2'_sig,c3'_sig,c4'_sig,c5'_sig,
   n0'_sig,n1'_sig,double'_sig)          /\
  LATCH(c5'_sig,c5_sig)                  /\
  LATCH(c4'_sig,c4_sig)                  /\
  LATCH(c3'_sig,c3_sig)                  /\
  LATCH(c2'_sig,c2_sig)                  /\
  LATCH(c1'_sig,c1_sig)                  /\
  LATCH(c0'_sig,c0_sig)                  /\
  LATCH(n1'_sig,n1_sig)                  /\
  LATCH(n0'_sig,n0_sig)                  /\
  LATCH(double'_sig,double_sig))
```

Using the definition of COUNTLOGIC_DEV and the result from Section 9 relating COUNTLOGIC_IMP to COUNTLOGIC, we can prove the following lemma relating CIRCUIT_IMP to COUNTLOGIC:

```
CIRCUIT_IMP(((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),
             double_sig,
             (n1_sig,n0_sig)),
            (l5_sig,l4_sig,l3_sig,l2_sig,l1_sig,l0_sig),
            (f1_sig,f0_sig)) =
 !t.(V6(c5_sig(t+1),c4_sig(t+1),c3_sig(t+1),
        c2_sig(t+1),c1_sig(t+1),c0_sig(t+1)),
     double_sig (t+1),
     V2(n1_sig(t+1),n0_sig(t+1))) =
     COUNTLOGIC
      ((V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t_),
        double_sig t,
        V2(n1_sig t,n0_sig t)),
       V6(l5_sig t,l4_sig t,l3_sig t,l2_sig t,l1_sig t,l0_sig t),
       V2(f1_sig t,f0_sig t))
```

We then use this lemma in combination with the initial results of this section, the partial correctness and termination of COUNTLOGIC, to prove the partial correctness and the termination of the circuit:

```
CIRCUIT_IMP
 (((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),
    double_sig,
    (n1_sig,n0_sig)),
  (l5_sig,l4_sig,l3_sig,l2_sig,l1_sig,l0_sig),
  (f1_sig,f0_sig))                          /\
(V2(n1_sig t,n0_sig t) = #00)               /\
(NEXTTIME (t,t') (\x. V2(n1_sig x,n0_sig x)  = #00))
==>
(V6(c5_sig t',c4_sig t',c3_sig t',c2_sig t',c1_sig t',c0_sig t') =
  COUNTER(V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),
          V6(l5_sig(t+1),l4_sig(t+1),l3_sig(t+1),
             l2_sig(t+1),l1_sig(t+1),l0_sig(t+1)),
          V2(f1_sig t,f0_sig t)))
```

and

```
CIRCUIT_IMP(((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),
             double_sig,
             n1_sig,n0_sig),
            (l5_sig,l4_sig,l3_sig,l2_sig,l1_sig,l0_sig),
            f1_sig,f0_sig) /\
(V2(n1_sig t,n0_sig t) = #00)
==>
(?t'. NEXTTIME(t,t')(\x. V2(n1_sig x,n0_sig x) = #00))
```

If the function VS2 satisfies `VS2(s1,s0)t = V2(s1 t,s0 t)` and the function VS6 satisfies `VS6(s5,s4,s3,s2,s1,s0)t = V6(s5 t,s4 t,s3 t,s2 t,s1 t,s0 t)`, then the two theorems above can be simplified to

```
CIRCUIT_IMP((count_sigs,double_sig,node_sigs),loadin_sigs,func_sigs) /\
(VS2 node_sigs t = #00)                                              /\
(NEXTTIME (t,t') (\x. VS2 node_sigs x  = #00))
==>
(VS6 count_sigs t' =
 COUNTER(VS6 count_sigs t,
         VS6 loadin_sigs (t+1),
         VS2 func_sigs t))
```

and

```
CIRCUIT_IMP((count_sigs,double_sig,node_sigs),loadin_sigs,func_sigs) /\
(VS2 node_sigs t = #00)
==>
(?t'. NEXTTIME(t,t')(\x. VS2 node_sigs x = #00))
```

These theorems are easily combined to prove the total correctness of the circuit, namely

```
CIRCUIT_IMP((count_sigs,double_sig,node_sigs),loadin_sigs,func_sigs) /\
(VS2 node_sigs t = #00)
==>
(?t'. NEXTTIME(t,t')(\x. VS2 node_sigs x = #00) /\
      (VS6 count_sigs t' =
       COUNTER(VS6 count_sigs t,
               VS6 loadin_sigs (t+1),
               VS2 func_sigs t)))
```

This completes the verification of the counter.

11 Conclusions

Although the counter described in this report is simple, its formal verification may appear to the reader to be depressingly large and intricate, and it might seem that the verification of much more complex devices would be impractical. We believe, however, that real hardware *can* be formally verified using existing techniques.

Our optimism is based on the success of the LCF methodology in partially automating the production of long proofs; succinct and general meta-language procedures can generate long chains of inferences. Proofs of the sort described in this paper consist largely in unwinding definitions, obvious case analyses, and simple rewritings; they require little cleverness and producing them is really simpler than it probably seems. With some experience, HOL proofs can be done quite quickly. More of the time and effort go into writing, correcting and typing in the definitions than producing the proofs, even though in number of inference steps or cpu-time used the proofs are large objects. In this proof, for example, various additional formal specifications had to be invented and added to the RSRE descriptions:

- the apparatus for relating different timescales, (`NEXTTIME` *etc.*),
- the relational specifications of the circuit (`COUNTLOGIC_SPEC`),
- the structural description of the circuit (`COUNTLOGIC_IMP`),
- the temporal description of the circuit (`CIRCUIT_IMP`).

Furthermore, the correctness criteria had to be formulated for the host machine, the high level design, the circuit, and the complete counter implementation. Given all of that, generating the proofs was straightforward.

References

[Cullyer *et al.*] Cullyer, W.J., and Pygott, C.H., "Hardware Proofs using LCF_LSM and ELLA", RSRE Memorandum 3832, September 1985.

[Gordon83] Gordon, M., "LCF_LSM: A System for Specifying and Verifying Hardware", University of Cambridge Computer Laboratory Technical Report no. 41, 1983.

[Gordon85(a)] Gordon, M., "HOL: A Machine Oriented Formulation of Higher-Order Logic", University of Cambridge Computer Laboratory Technical Report no. 68, 1985.

[Gordon85(b)] Gordon, M., "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware", University of Cambridge Computer Laboratory Technical Report no. 77; also to appear in *Formal Aspects of* VLSI *Design*, Proceedings of the workshop on VLSI, Edinburgh. North Holland, 1986.

[Gordon *et al.*] Gordon, M., Milner, R. and Wadsworth, C., "Edinburgh LCF: a mechanized logic of computation", Lecture Notes in Computer Science Number 78, Springer-verlag, 1978.

[Hunt] Hunt, W. A. Jr., "FM8501: A Verified Microprocessor", Technical Report 47, Institute for Computing Science, The University of Texas at Austin, 1985.

[Joyce *et al.*] Joyce, J., Birtwistle, G. and Gordon, M., "Proving a Computer Correct in Higher Order Logic", Technical Report, University of Calgary, 1985.

[Kershaw] Kershaw, J., "VIPER: a microprocessor for safety-critical applications", RSRE memorandum 3754, September, 1984.

[Morison *et al.*] "ELLA: Hardware description or specification?", proceedings IEEE International Conference, CAD-84, Santa Clara, November 1984.

[Pygott] Pygott, C.H., "Formal proof of correspondence between a hardware module and its gate level implementation", RSRE Memorandum 85012, June, 1985.

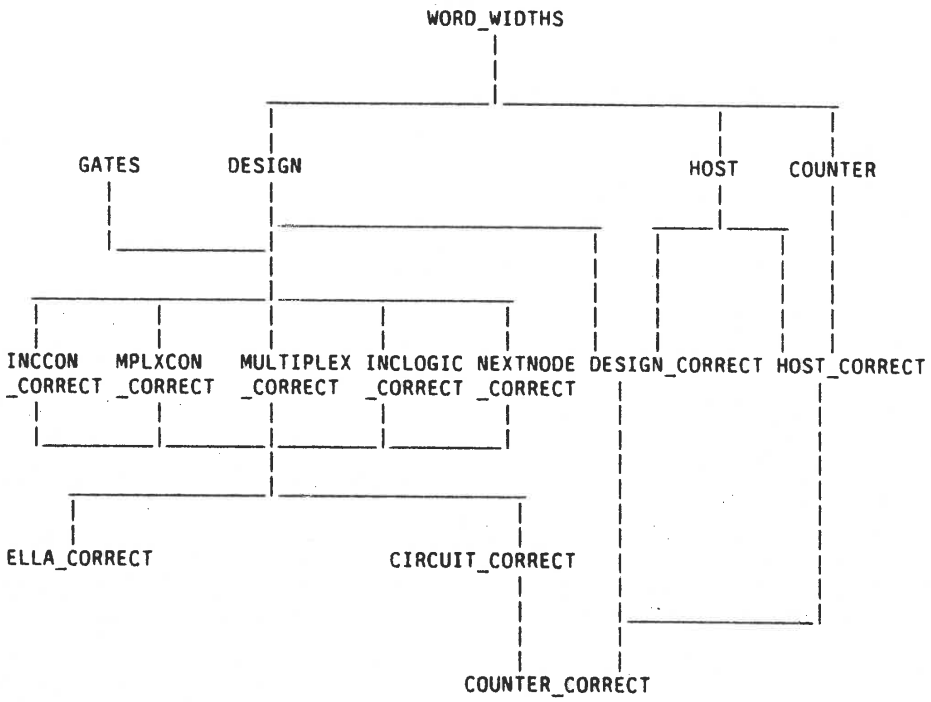
A Proof of the Counter in the HOL System

This appendix contains a complete description of how to specify and verify the counter using the HOL system. The ML code that follows (rules, tactics *etc.*) has not been polished; there are many inelegancies (and a few downright hacks). We felt it better to show some 'real' code rather than the smooth and polished examples that one is tempted to put on public display. Note, however, that ML's type discipline ensures that no amount of wild hacking can result in non-theorems being proved ([Gordon *et al.*]).

```
%============================================================================%
% This Appendix contains all of the ML files required to perform the counter %
% proof. The file you are reading now is a text file, not a source file. In %
% the source files, comments and explanations appear in boxes like this one. %
%                                                                            %
% The ML files that follow are all documented so that the interested reader  %
% should be able to follow the details of the machine proofs. Figures        %
% mentioned refer to the RSRE report.                                        %
%                                                                            %
% In this file, we describe the theory structure in which the counter's      %
% correctness is stated and proved. We also summarize the main results, for %
% easy reference.                                                            %
%============================================================================%
```

THE STRUCTURE OF THEORIES

This graph shows the structure of theories in which the correctness of the counter is stated and proved. To build the structure of theories, the ml files corresponding to each theory should be loaded into HOL in the appropriate order -- that is, horizontal layers of the graph, from top to bottom, in any order within one layer.



FILES FOR BUILDING THEORIES

The correspondence of ML files to construct theories, and the theories, is shown in this table:

ML file	Theory
word_widths.ml	WORD_WIDTHS.th
gates.ml	GATES.th
design.ml	DESIGN.th
host.ml	HOST.th
counter.ml	COUNTER.th
inccon_correct.ml	INCCON_CORRECT.th
mplxcon_correct.ml	MPLXCON_CORRECT.th
multiplex_correct.ml	MULTIPLEX_CORRECT.th
inclogic_correct.ml	INCLOGIC_CORRECT.th
nextnode_correct.ml	NEXTNODE_CORRECT.th
design_correct.ml	DESIGN_CORRECT.th
host_correct.ml	HOST_CORRECT.th
circuit_correct.ml	CIRCUIT_CORRECT.th
ella_correct.ml	ELLA_CORRECT.th
counter_correct.ml	COUNTER_CORRECT.th

To build the theories, the file utilities.ml, of useful ML functions, is also needed.

MAIN RESULTS

This table is a brief summary of the main results:

Theory	Main Theorems	Purpose
COUNTER_CORRECT	COUNTER_PCORRECT COUNTER_TERMINATES COUNTER_CORRECT	The top level partial correctness & termination of the counter, & total correctness, with time.
ELLA_CORRECT	MULTIPLEX_ELLA_IMP_EQUIV INCLOGIC_ELLA_IMP_EQUIV INCCON_ELLA_IMP_EQUIV MPLXCON_ELLA_IMP_EQUIV NEXTNODE_ELLA_IMP_EQUIV	The equivalence of the ELLA definitions to the implementation level definitions: MULTIPLEX_IMP_DEF on MULTIPLEX_CORRECT.th, and so on.
CIRCUIT_CORRECT	COUNTLOGIC_DEF_IMP_EQUIV	The equivalence of the definition and implementation of COUNTLOGIC, i.e. the combinatorial behaviour of the circuit.
HOST_CORRECT	HOST_TERMINATES HOST_PCORRECT HOST_CORRECT	The partial correctness and termination of the host machine, & total correctness, with time.
DESIGN_CORRECT	NEXT_COUNTLOGIC_EQUIV	The equivalence of host machine and COUNTLOGIC (the behaviour of the circuit, without time).
NEXTNODE_ CORRECT	NEXTNODE_DEF_SPEC_EQUIV NEXTNODE_SPEC_IMP_EQUIV	For this and the next four: Equivalence of the definition and intermediate-level specification; and equivalence of the specification and implementation (combinatorially).
INCLOGIC_ CORRECT	INCLOGIC_DEF_SPEC_EQUIV INCLOGIC_SPEC_IMP_EQUIV	
MULTIPLEX_ CORRECT	MULTIPLEX_DEF_SPEC_EQUIV MULTIPLEX_SPEC_IMP_EQUIV	
MPLXCON_ CORRECT	MPLXCON_DEF_SPEC_EQUIV MPLXCON_SPEC_IMP_EQUIV	
INCCON_ CORRECT	INCCON_DEF_SPEC_EQUIV INCCON_SPEC_IMP_EQUIV	
COUNTER		Definition of the counter.
DESIGN		Definition of COUNTLOGIC, etc.
HOST		Definition of host machine.
GATES		Definitions of NAND, NOR, etc.
WORD_WIDTHS		The type definitions, axioms and theorems.

THE MAIN RESULTS IN DETAIL

COUNTER_PCORRECT:

```
   "CIRCUIT_IMP((count_sigs,double_sig,node_sigs),loadin_sigs,func_sigs) /\
    (VS2 node_sigs t = #00)                                               /\
    (NEXTTIME (t,t') (\x. VS2 node_sigs x  = #00))
    ==>
    (VS6 count_sigs t' =
     COUNTER(VS6 count_sigs t,
             VS6 loadin_sigs (t + 1),
             VS2 func_sigs t))"
```

COUNTER_TERMINATES:

```
   "CIRCUIT_IMP((count_sigs,double_sig,node_sigs),loadin_sigs,func_sigs) /\
    (VS2 node_sigs t = #00)
```

```
    ==>
    (?t'. NEXTTIME(t,t')(\x. VS2 node_sigs x = #00))"

COUNTER_CORRECT:

   "CIRCUIT_IMP((count_sigs,double_sig,node_sigs),loadin_sigs,func_sigs) /\
    (VS2 node_sigs t = #00)
    ==>
    ?t'. NEXTTIME (t,t') (\x. VS2 node_sigs x =#00) /\
         (VS6 count_sigs t' =
          COUNTER(VS6 count_sigs t,
                  VS6 loadin_sigs (t + 1),
                  VS2 func_sigs t))"

COUNTLOGIC_DEF_IMP_EQUIV:

   "COUNTLOGIC_IMP(c0,c1,c2,c3,c4,c5,n0,n1,
                   l0,l1,l2,l3,l4,l5,f0,f1,double,
                   c0',c1',c2',c3',c4',c5',n0',n1',double') =
    ((V6(c5',c4',c3',c2',c1',c0'),double',V2(n1',n0')) =
    COUNTLOGIC((V6(c5,c4,c3,c2,c1,c0),double,V2(n1,n0)),
               V6(l5,l4,l3,l2,l1,l0),
               V2(f1,f0)) )"

MULTIPLEX_ELLA_IMP_EQUIV:

   "MULTIPLEX_IMP(c0,c1,c2,c3,c4,c5,l0,l1,l2,l3,l4,l5,b,
                  c0',c1',c2',c3',c4',c5') =
    (V6(c5',c4',c3',c2',c1',c0') =
    MPLEXCIRC (V6(c5,c4,c3,c2,c1,c0))(V6(l5,l4,l3,l2,l1,l0)) b)"

INCLOGIC_ELLA_IMP_EQUIV:

   "INCLOGIC_IMP(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
    (V6(c5',c4',c3',c2',c1',c0') =
    INCCIRC (V6(c5,c4,c3,c2,c1,c0)) b )"

NEXTNODE_ELLA_IMP_EQUIV:

   "NEXTNODE_IMP(n0,n1,f0,f1,double,n0',n1') =
    (V2(n1',n0') = SND(SND(CONTROLCIR(V2(n1,n0))(V2(f1,f0)) double)))"

MPLXCON_ELLA_IMP_EQUIV:

   "MPLXCON_IMP(n0,n1,b) =
    (b = FST(SND(CONTROLCIR (V2(n1,n0)) (V2(f1,f0)) double)))"

INCCON_ELLA_IMP_EQUIV:

   "INCCON_IMP(n0,n1,b) =
    (b = FST(CONTROLCIR (V2(n1,n0)) (V2(f1,f0)) double))"

HOST_PCORRECT:

   "(!t. state_sig(t+1) =
    NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t)=#00)                                  */\
    NEXTTIME (t,t') (\x. NODE(state_sig x)=#00)
    ==>
    (COUNT(state_sig t') =
    COUNTER(COUNT(state_sig t),loadin_sig(t+1),func_sig t))"

HOST_TERMINATES:

   "(!t.state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t) = #00)
    ==
    ?t'.NEXTTIME(t,t')(\x.NODE(state_sig x)=#00)"

HOST_CORRECT:

   "(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t)=#00)
    ==>
    ?t'. NEXTTIME (t,t') (\x. NODE(state_sig x)=#00) /\
         (COUNT(state_sig t') =
           COUNTER(COUNT(state_sig t),loadin_sig(t+1),func_sig t))"

NEXT_COUNTLOGIC_EQUIV:
```

```
   "NEXT = COUNTLOGIC"

INCLOGIC_DEF_SPEC_EQUIV:

   "INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
    (V6(c5',c4',c3',c2',c1',c0') =
    INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b))"

INCLOGIC_SPEC_IMP_EQUIV:

   "INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
    INCLOGIC_IMP (c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5')"

where INCLOGIC is:

   "INCLOGIC(count:word6,noinc:bool) =
    let countval = VAL6 count in
    (noinc => count | ((countval=63) => WORD6 0 | WORD6(countval+1)))"

and INCLOGIC_SPEC is:

   "INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,
                  c0',c1',c2',c3',c4',c5') =
    (c5',c4',c3',c2',c1',c0') =
    (b => (c5,c4,c3,c2,c1,c0) |
          ((c5,c4,c3,c2,c1,c0) = (T,T,T,T,T,T)) => (F,F,F,F,F,F) |
          TUPLE6(BITS6(WORD6(V[c5;c4;c3;c2;c1;c0] + 1))))"

and INCLOGIC_IMP is:

   "INCLOGIC_IMP(c0,c1,c2,c3,c4,c5,b,
                 c0',c1',c2',c3',c4',c5') =
    ?b' x1 x2 x3 x3' x4 x5.
     INV(b,b')                 /\
     INV(x3,x3')               /\
     NAND(b',c0,x1)            /\
     NAND3(b',c0,c1,x2)        /\
     NAND4(b',c0,c1,c2,x3)     /\
     NAND(x3',c3,x4)           /\
     NAND3(x3', c3,c4,x5)      /\
     XNOR(c0,b,c0')            /\
     XNOR(c1,x1,c1')           /\
     XNOR(c2,x2,c2')           /\
     XNOR(c3,x3,c3')           /\
     XNOR(c4,x4,c4')           /\
     XNOR(c5,x5,c5')"
```

The results for NEXTNODE, MPLXCON, MULTIPLEX and INCCON are similiar.

Finally, COUNTER is defined as:

```
   "COUNTER(count:word6,loadin:word6,func:word2) =
    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))       |
                    ARB)"
```

THE TIME TO DO THE PROOFS

The relative times for loading the various files are given below, in abstract units. Almost all of the time is spent doing the proofs.

word_widths.ml	13 units
gates.ml	0
design.ml	0
host.ml	0
counter.ml	0
inccon_correct.ml	0.5
mplxcon_correct.ml	0.5
multiplex_correct.ml	3
inclogic_correct.ml	50
nextnode_correct.ml	27

design_correct.ml	4
host_correct.ml	16
ella_correct.ml	48
circuit_correct.ml	8
counter_correct.ml	10

ABOUT THE PROOF

The logically difficult part of the proof is contained in the theory HOST_CORRECT. The computationally heavy parts are the large case analyses of the behaviours at the gate level, contained in the theories INCLOGIC_CORRECT, NEXTNODE_CORRECT and ELLA_CORRECT.

```
%============================================================================%
% This file contains a miscellaneous collection of theorems and rules of     %
% inference.                                                                 %
%============================================================================%

%----------------------------------------------------------------------------%
% The following lemma is |- !x y x' y'. ((x,y)=(x',y')) ==> ((x=x')/\(y=y')) %
% which is useful  in proving PAIR_SPLIT below.                              %
%----------------------------------------------------------------------------%

let PAIR_IMP_LEMMA =
 DISCH_ALL
 (CONJ
  (REWRITE_RULE[FST](AP_TERM "FST:*#**->*"(ASSUME "(x:*,y:**)=(x',y')")))
  (REWRITE_RULE[SND](AP_TERM "SND:*#**->**"(ASSUME "(x:*,y:**)=(x',y')")))
 );;

let PAIR_SPLIT =
 TAC_PROOF
  (([], "!(x:*) (y:**) x' y'. ((x,y)=(x',y')) = ((x=x') /\ (y=y'))"),
   REPEAT GEN_TAC
   THEN EQ_TAC
   THEN ASM_REWRITE_TAC[PAIR_IMP_LEMMA]
   THEN REPEAT STRIP_TAC
   THEN ASM_REWRITE_TAC[]);;

%----------------------------------------------------------------------------%
% The tactic CONJ_SET_TAC solves goals of the form                           %
%                                                                            %
%    u1 /\ ... /\ un = v1 /\ ... /\ vm                                       %
%                                                                            %
% when the sets of terms {u1,...,un} and {v1,...,vm} are equal.              %
%----------------------------------------------------------------------------%

let CONJ_SET_TAC:tactic(asl,tm) =
 (let l,r = dest_eq tm
  in
  let ll = conjuncts l
  and rl = conjuncts r
  in
  if set_equal ll rl
   then ([],\thl. CONJ_SET_CONV ll rl)
   else fail)
 ? failwith `CONJ_SET_TAC`;;

%----------------------------------------------------------------------------%
% GDISJ_CASES_TAC generalizes DISJ_CASES_TAC by:                             %
%                                                                            %
%    1.  dealing with n-ary disjunctions, and                                %
%                                                                            %
%    2.  substituting the case assumptions into the goal.                    %
%                                                                            %
%                           t(a)                                             %
%                  ==========================     |- (a=a1)\/...\/(a=an)     %
%                  [a=a1]t(a1) ... [a=an]t(an)                               %
%                                                                            %
%----------------------------------------------------------------------------%

let GDISJ_CASES_TAC =
 REPEAT_TCL DISJ_CASES_THEN (\th. SUBST1_TAC th THEN ASSUME_TAC th);;

%----------------------------------------------------------------------------%
% LIST_REWRITE_CONV is a conversion for rewriting all subterms of a given    %
% using a list of theorems.                                                  %
%----------------------------------------------------------------------------%

let LIST_REWRITE_CONV =
 let basic_net = mk_conv_net basic_rewrites
 in
 \thl. TOP_DEPTH_CONV
       (REWRITES_CONV(merge_term_nets (mk_conv_net thl) basic_net));;

%----------------------------------------------------------------------------%
% The ordinary BOOL_CASES_TAC does not put case assumption in assumption     %
% list, which may be necessary for resolution (e.g. in INCLOGIC proof).      %
%----------------------------------------------------------------------------%

let ASM_BOOL_CASES_TAC t = DISJ_CASES_TAC(SPEC t EXCLUDED_MIDDLE);;
```

```
%============================================================================%
% This theory sets up types ":word2" and ":word6", etc., and some useful     %
% definitions and axioms.                                                    %
%                                                                            %
% All of the reasoning about words, at the moment, is done by reasoning      %
% about SPECIFIC words, such as "#111110" or "#00". This means that some of  %
% the facts we would like to prove must be asserted as axioms, because the   %
% proofs would involve too many cases (e.g. 2 EXP 12, where two variables    %
% of type ":word6" are involved). A better theory, which we plan to build    %
% in the future, would have abstract definitions of words, and hence fewer   %
% axioms and more theorems. In this theory, facts are proved where possible, %
% and it is noted where proofs are impractical.                              %
%============================================================================%
%----------------------------------------------------------------------------%
% Examples of useful definitions:                                            %
%                                                                            %
%             "WORD2 2 = #10"                                                %
%             "V[T;F]  = 2"                                                  %
%             "BITS2 #10 = [T;F]"                                            %
%             "VAL2 #10  = 2"                                                %
%             "TUPLE6[T;F;F;F;F;F] = (T,F,F,F,F,F)"                          %
%             "V2(T,F) = #10"                                                %
%----------------------------------------------------------------------------%

new_theory `WORD_WIDTHS`;;

%----------------------------------------------------------------------------%
% The following sets up the types ":word2" and ":word6", the constants "V",  %
% "BITS2", "BITS6", "EL", "VAL2", "VAL6", "WORD2" and "WORD6",               %
% and the facilities (conversions) for                                       %
% rewriting expressions containing particular instances of the two types.    %
%----------------------------------------------------------------------------%

declare_word_widths[2;6];;

%----------------------------------------------------------------------------%
% The following file contains useful general ML functions, such as:          %
%    GDISJ_CASES_TAC: a tactic for case analysis according to a disjunction. %
%----------------------------------------------------------------------------%

loadt `~/counter/utilities`;;

%----------------------------------------------------------------------------%
% The new constant                                                           %
% "ARB:*" is an arbitrary value used as the final arm of conditionals when   %
%  RSRE don't give one.                                                      %
%  For example, it is used in formulating HOST_NEXT_THM on HOST_CORRECT.th.  %
%----------------------------------------------------------------------------%

new_definition(`ARB_DEF`, "ARB = @x:*.T");;

%----------------------------------------------------------------------------%
% The constant "TUPLE6" is used in the definition INCLOGIC_SPEC_DEF, on      %
% INCLOGIC.th.                                                               %
%    "TUPLE6[F;T;T;T;T;T] = (F,T,T,T,T,T)"                                   %
% Axioms for it are found below.                                             %
%----------------------------------------------------------------------------%

new_constant
 (`TUPLE6`,":bool list -> (bool # bool # bool # bool # bool # bool)");;

%----------------------------------------------------------------------------%
% The definitions below are useful abbreviations.                            %
% "V2:bool # bool -> word2"  and                                             %
% "V6:bool # bool # bool # bool # bool # bool -> word6"                       %
%----------------------------------------------------------------------------%

let V2_DEF = new_definition(`V2_DEF`, "V2(b1,b0) = WORD2(V[b1;b0])");;

let V6_DEF = new_definition(`V6_DEF`,"V6(b5,b4,b3,b2,b1,b0) =
                                      WORD6(V[b5;b4;b3;b2;b1;b0])");;

%============================================================================%
% The following axiom is for reasoning about objects of type ":word2".       %
%============================================================================%

let WORD2_CASES_AX = new_axiom (`WORD2_CASES_AX`,
                     "!w:word2. (w = #00) \/ (w = #01) \/
```

```
                                (w = #10) \/ (w = #11)");;

%============================================================================%
% To state the corresponding axiom for ":word6", we need some ML apparatus:  %
%============================================================================%
%----------------------------------------------------------------------------%
% #mk_word6 45;;                                                             %
% "#101101" : term                                                           %
%----------------------------------------------------------------------------%

let mk_word6 n =
  mk_const(implode(`#`.mk_bin_rep(6,n)),":word6");;

%----------------------------------------------------------------------------%
% #all_word6_list 2;;                                                        %
% ["#000001"; "#000000"] : term list                                         %
%----------------------------------------------------------------------------%

letrec all_word6_list n = (n = 0 => nil |
                           (mk_word6 (n - 1)).(all_word6_list (n - 1)));;

%----------------------------------------------------------------------------%
% #word6_disj "w:word6";;                                                    %
% "(w = #111111) \/                                                          %
%  (w = #111110) \/                                                          %
%  (w = #111101) \/                                                          %
%  (w = #111100) \/                                                          %
%       .                                                                    %
%       .                                                                    %
%       .                                                                    %
% (w = #000001) \/                                                           %
% (w = #000000)"                                                             %
% : term                                                                     %
%----------------------------------------------------------------------------%

let word6_disj w =
list_mk_disj(map(\x.mk_eq(w,x)) (all_word6_list 64));;

%----------------------------------------------------------------------------%
% The following is the axiom required for reasoning about the type ":word6": %
%----------------------------------------------------------------------------%

let WORD6_CASES_AX = new_axiom(`WORD6_CASES_AX`,word6_disj "w:word6");;

%----------------------------------------------------------------------------%
% The axioms VAL6_LESS_AX and VAL6_WORD6_AX are both used to prove           %
% ADD2_LEMMA, which is used to prove HOST_CORRECT, on HOST_CORRECT.th.       %
% VAL6_LESS_AX could easily be proved using GDISJ_CASES_TAC on an instance   %
% of WORD6_CASES_AX. One would first have to prove 64 lemmas of the form     %
% "0 < 64", "1 < 64" and so on. To prove VAL6_WORD_AX, one would need the    %
% same 64 lemmas; then one could either use induction, or another lemma, of  %
% the form "n < 64 => (n=0) \/ (n=1) ... (n=63)".                            %
%----------------------------------------------------------------------------%

let VAL6_LESS_AX  =
 new_axiom(`VAL6_LESS_AX`,"!w:word6. VAL6 w < 64");;

let VAL6_WORD6_AX =
 new_axiom(`VAL6_WORD6_AX`, "!n:num. (n<64) ==> (VAL6(WORD6 n) = n)");;

%----------------------------------------------------------------------------%
%  The following nine axioms/theorems are required for the                   %
%  proofs of INCLOGIC_DEF_SPEC_EQUIV (on INCLOGIC_CORRECT.th) and            %
%  MULTIPLEX_DEF_SPEC_EQUIV (on MULTIPLEX_CORRECT.th).                       %
%  The first two axioms are for reasoning about 6-tuples.                    %
%  The next five  are required to cope with expressions of the form          %
%  "...WORD6(...)... = ...WORD6(...)...", "...V(...)... = ...V(...)...",     %
%  and "...[...]... = ...[...]...".                                          %
%  The last two show certain inverses.                                       %
%     MULTIPLEX_CORRECT.th uses LIST_AX_1, WORD6_AX2 and V_AX1.              %
%     INCLOGIC_CORRECT.th  uses all nine.                                    %
%     INCLOGIC_CORRECT.th also uses LEMMA6 (on this theory) which uses       %
%  V_AX1 and LIST_AX1.                                                       %
%----------------------------------------------------------------------------%
%----------------------------------------------------------------------------%
% TUPLE6_AX is definitional, so not proved.                                  %
%----------------------------------------------------------------------------%

let TUPLE6_AX =       new_axiom(`TUPLE6_AX`,
```

```
                                "TUPLE6[x1;x2;x3;x4;x5;x6] =
                                 (x1,x2,x3,x4,x5,x6)");;

%----------------------------------------------------------------------------%
% Possibly we could prove TUPLE6_AX2 by induction on n, given TUPLE6_AX.     %
% However, we would first have to say what "WORD6 n" is for n>63.             %
%----------------------------------------------------------------------------%

let TUPLE6_AX2 =      new_axiom(`TUPLE6_AX2`,
                                "((b0,b1,b2,b3,b4,b5) =
                                 TUPLE6(BITS6(WORD6 n))) =
                                 ([b0;b1;b2;b3;b4;b5] =
                                 (BITS6(WORD6 n)))");;

%----------------------------------------------------------------------------%
% LIST_AX1 could be proved in an ordinary list theory, if there were one.    %
%----------------------------------------------------------------------------%

let LIST_AX1 =        new_axiom(`LIST_AX1`,
                                "((CONS (x:*) l) = (CONS x' l')) =
                                 ((x = x') /\ (l = l'))");;


%----------------------------------------------------------------------------%
% The proofs of WORD6_AX1, WORD6_AX2 and V_AX1                               %
% would have 2 EXP 12 cases, so are not feasible in the current framework.   %
%----------------------------------------------------------------------------%

let WORD6_AX1 =       new_axiom(`WORD6_AX1`,
                                "~(V[c0;c1;c2;c3;c4;c5] = 63) ==>
                                (! b0 b1 b2 b3 b4 b5.
                                ((WORD6(V[b0;b1;b2;b3;b4;b5]) =
                                  WORD6(V([c0;c1;c2;c3;c4;c5]) + 1)) =
                                 (V[b0;b1;b2;b3;b4;b5] =
                                 ((V[c0;c1;c2;c3;c4;c5]) + 1) )))");;

let WORD6_AX2 =       new_axiom(`WORD6_AX2`,
                                 "(WORD6(V[b0;b1;b2;b3;b4;b5]) =
                                   WORD6(V[c0;c1;c2;c3;c4;c5])) =
                                  (V[b0;b1;b2;b3;b4;b5] =
                                   V[c0;c1;c2;c3;c4;c5])");;

let V_AX1 =           new_axiom(`V_AX1`, "(V[b0;b1;b2;b3;b4;b5] =
                                          V[c0;c1;c2;c3;c4;c5]) =
                                         ([b0;b1;b2;b3;b4;b5] =
                                          [c0;c1;c2;c3;c4;c5])");;

%----------------------------------------------------------------------------%
% V_AX2 would follow from V_AX1 and 64 lemmas of the form:                   %
% "63 = V[T;T;T;T;T;T]", with a 64 case analysis, if one wished.             %
%----------------------------------------------------------------------------%

let V_AX2 =           new_axiom(`V_AX2`, "([b0;b1;b2;b3;b4;b5] =
                                          BITS6 w) =
                                         (V[b0;b1;b2;b3;b4;b5] =
                                          V(BITS6 w))");;

%----------------------------------------------------------------------------%
% To prove VAL6_WORD6_AX2 and BITS6_WORD6_AX,  we use a simple tactic as     %
% follows (which could easily be generalised into an n-ary tactic):          %
%----------------------------------------------------------------------------%

let SIX_BOOL_CASES_TAC [b0;b1;b2;b3;b4;b5] =
     BOOL_CASES_TAC b0  THEN
     BOOL_CASES_TAC b1  THEN
     BOOL_CASES_TAC b2  THEN
     BOOL_CASES_TAC b3  THEN
     BOOL_CASES_TAC b4  THEN
     BOOL_CASES_TAC b5;;

let VAL6_WORD6_AX2 =
 prove_thm(`VAL6_WORD6_AX2`,
           "VAL6(WORD6(V[b0;b1;b2;b3;b4;b5])) =
            V[b0;b1;b2;b3;b4;b5]",
      SIX_BOOL_CASES_TAC ["b0";"b1";"b2";"b3";"b4";"b5"]
      THEN CONV_TAC(V_CONV THENC WORD_CONV THENC VAL_CONV THENC EQ_CONV));;


let BITS6_WORD6_AX =
```

```
 prove_thm(`BITS6_WORD6_AX`,
           "~(V[b0;b1;b2;b3;b4;b5] = 63) ==>
            (V(BITS6(WORD6(V[b0;b1;b2;b3;b4;b5] + 1))) =
            (V[b0;b1;b2;b3;b4;b5] + 1))",
      SIX_BOOL_CASES_TAC ["b0";"b1";"b2";"b3";"b4";"b5"]
      THEN CONV_TAC(V_CONV THENC ADD_CONV THENC
                  (WORD_CONV ORELSEC ALL_CONV) THENC BITS_CONV
                   THENC V_CONV THENC EQ_CONV)
      THEN REWRITE_TAC[]);;

%----------------------------------------------------------------------------%
% The following two facts are needed in order to apply the rule EXPAND, in   %
% the proof of COUNTLOGIC.  They replace a compound left hand side with a    %
% simple one. (See CIRCUIT_CORRECT.th.)  BITS6_AX would have 2 EXP 12 cases, %
% so is asserted in this theory.                                             %
%----------------------------------------------------------------------------%

let BITS2_AX = prove_thm(`BITS2_AX`,
       "(V2(d0,d1) = x) = ((d0 = EL 1 (BITS2 x)) /\
                           (d1 = EL 0 (BITS2 x)))",
       GDISJ_CASES_TAC (SPEC "x:word2" WORD2_CASES_AX)
       THEN ASM_REWRITE_TAC[V2_DEF]
       THEN CONV_TAC(BITS_CONV THENC EL_CONV)
       THEN BOOL_CASES_TAC "d0:bool"
       THEN BOOL_CASES_TAC "d1:bool"
       THEN CONV_TAC(V_CONV THENC WORD_CONV THENC EQ_CONV)
       THEN REWRITE_TAC[]);;

new_axiom(`BITS6_AX`,
          "(V6(d0,d1,d2,d3,d4,d5) = x) = ((d0 = EL 5 (BITS6 x)) /\
                                          (d1 = EL 4 (BITS6 x)) /\
                                          (d2 = EL 3 (BITS6 x)) /\
                                          (d3 = EL 2 (BITS6 x)) /\
                                          (d4 = EL 1 (BITS6 x)) /\
                                          (d5 = EL 0 (BITS6 x)))");;

%----------------------------------------------------------------------------%
% The following 3 theorems are necessary for the proof of COUNTLOGIC, on     %
% CIRCUIT_CORRECT.th.                                                        %
%----------------------------------------------------------------------------%

let EL_BITS2_AX =
  prove_thm(`EL_BITS2_AX`,
            "EL 0(BITS2(V2(f1,f0))) = f0",
            BOOL_CASES_TAC "f0:bool"
            THEN BOOL_CASES_TAC "f1:bool"
            THEN REWRITE_TAC[V2_DEF]
            THEN CONV_TAC (V_CONV THENC WORD_CONV THENC BITS_CONV
                           THENC EL_CONV)
            THEN REWRITE_TAC[]);;


let BITS6_WORD6_AX2 = prove_thm(`BITS6_WORD6_AX2`,
                "WORD6(V [EL 5 (BITS6 (x:word6));
                          EL 4 (BITS6 x);EL 3 (BITS6 x);EL 2 (BITS6 x);
                          EL 1 (BITS6 x);EL 0 (BITS6 x)]) =  x",
                GDISJ_CASES_TAC(SPEC "x:word6" WORD6_CASES_AX)
                THEN ASM_REWRITE_TAC[]
                THEN CONV_TAC(BITS_CONV THENC
                              EL_CONV THENC V_CONV THENC WORD_CONV)
                THEN REWRITE_TAC[]);;

let V6_THM1 = prove_thm(`V6_THM1`,
                        "V6(EL 5 (BITS6 (x:word6)),
                            EL 4 (BITS6 x),EL 3 (BITS6 x),EL 2 (BITS6 x),
                            EL 1 (BITS6 x),EL 0 (BITS6 x)) =  x",
                       REWRITE_TAC[V6_DEF;BITS6_WORD6_AX2]);;

%----------------------------------------------------------------------------%
% The following three lemmas are for INCLOGIC, on INCLOGIC_CORRECT.th.       %
% LEMMA4, to be used as a rewrite rule, says that                            %
% "#000000 = WORD6(V[F;F;F;F;F;F])".  Similarly, LEMMA5 says that            %
% "63 = V[T;T;T;T;T;T]".                                                     %
% LEMMA6, to be used by resolution, says that "~((b0,b1,b2,b3,b4,b5) =       %
% (T,T,T,T,T,T)) = ~(V[b0;b1;b2;b3;b4;b5]=63)".                              %
%----------------------------------------------------------------------------%

let LEMMA4 = prove_thm(`LEMMA4`,
                       "#000000 = WORD6(V[F;F;F;F;F;F])",
```

```
                        CONV_TAC(V_CONV THENC WORD_CONV)
                        THEN REWRITE_TAC[]);;

let LEMMA5 = prove_thm(`LEMMA5`,
                        "63 = V[T;T;T;T;T;T]",
                         CONV_TAC V_CONV THEN REWRITE_TAC[]);;

let LEMMA6 = save_thm(`LEMMA6`,
                      fst(EQ_IMP_RULE(AP_TERM "$~"
                      (TAC_PROOF(([],
    "((b0,b1,b2,b3,b4,b5) = (T,T,T,T,T,T)) = ((V[b0;b1;b2;b3;b4;b5]) = 63)"),
            REWRITE_TAC[LEMMA5;V_AX1;LIST_AX1;PAIR_SPLIT])))));;

%----------------------------------------------------------------------------%
% The following four theorems are for the proofs of the ELLA definitions,    %
% on ELLA_CORRECT.th.                                                        %
% First, we need 6 theorems of the form "EL 5(BITS6(V6(15,14,...,10))) = 15".%
% These could be stated  like EL_BITS2_AX, above, but to save having to      %
% prove 6 theorems, we prove BITS6_V6_THM and BITS2_V2_THM, and rewrite "EL" %
% in the main proof. Second, WORD2_TH2 is the analogue of WORD6_AX2, and     %
% V_TH3 is the analogue of V_AX1.                                            %
%----------------------------------------------------------------------------%

let BITS6_V6_THM = prove_thm(`BITS6_V6_THM`,
                             "BITS6(V6(b0,b1,b2,b3,b4,b5)) =
                              [b0;b1;b2;b3;b4;b5]",
                             SIX_BOOL_CASES_TAC["b0";"b1";"b2";"b3";"b4";"b5"]
                             THEN REWRITE_TAC[V6_DEF]
                             THEN CONV_TAC(V_CONV THENC WORD_CONV
                                           THENC BITS_CONV)
                             THEN REWRITE_TAC[]   );;

let BITS2_V2_THM = prove_thm(`BITS2_V2_THM`,
                             "BITS2(V2(b0,b1)) =
                              [b0;b1]",
                             BOOL_CASES_TAC "b0:bool"
                             THEN BOOL_CASES_TAC "b1:bool"
                             THEN REWRITE_TAC[V2_DEF]
                             THEN CONV_TAC(V_CONV THENC WORD_CONV
                                           THENC BITS_CONV)
                             THEN REWRITE_TAC[]   );;

let WORD2_TH2 =
  prove_thm(`WORD2_TH2`,
            "(WORD2(V[b0;b1]) = WORD2(V[c0;c1])) =
             (V[b0;b1] = V[c0;c1])",
            BOOL_CASES_TAC "b0:bool"
            THEN BOOL_CASES_TAC "b1:bool"
            THEN BOOL_CASES_TAC "c0:bool"
            THEN BOOL_CASES_TAC "c1:bool"
            THEN CONV_TAC(V_CONV THENC WORD_CONV THENC EQ_CONV)
            THEN REWRITE_TAC[] );;

let V_TH3 =
  prove_thm(`V_TH3`,
            "(V[b0;b1] = V[c0;c1]) =
             ([b0;b1] = [c0;c1])",
            BOOL_CASES_TAC "b0:bool"
            THEN BOOL_CASES_TAC "b1:bool"
            THEN BOOL_CASES_TAC "c0:bool"
            THEN BOOL_CASES_TAC "c1:bool"
            THEN CONV_TAC V_CONV
            THEN REWRITE_TAC[LIST_AX1]
            THEN CONV_TAC EQ_CONV
            THEN REWRITE_TAC[]);;

close_theory();;
```

```
%============================================================================%
% This theory contains the definitions of the gate relations in terms of     %
% HOL constants.                                                             %
%============================================================================%

new_theory`GATES`;;

new_definition
 (`NOR_DEF`,
  "NOR(in1,in2,out) = (out = ~(in1 \/ in2))");;

new_definition
 (`NAND_DEF`,
  "NAND(in1,in2,out) = (out = ~(in1 /\ in2))");;

new_definition
 (`NAND3_DEF`,
  "NAND3(in1,in2,in3,out) = (out = ~(in1 /\ in2 /\ in3))");;

new_definition
 (`NAND4_DEF`,
  "NAND4(in1,in2,in3,in4,out) = (out = ~(in1 /\ in2 /\ in3 /\ in4))");;

new_definition
 (`INV_DEF`,
  "INV(in,out) = (out = ~in)");;

new_definition
  (`XNOR_DEF`,
   "XNOR(in1:bool,in2:bool,out) = (out = (in1 = in2))");;

close_theory();;
```

```
%============================================================================%
% File for setting up the theory `DESIGN,` which contains the definitions    %
% for COUNTLOGIC.  (See p. 8 of report.)                                     %
%============================================================================%

new_theory `DESIGN`;;

new_parent `WORD_WIDTHS`;;

let major = ":word6#bool#word2";;

let INCCON_DEF =
 new_definition
  (`INCCON_DEF`,
   "INCCON(node:word2) = (VAL2 node = 0)");;

let MPLXCON_DEF =
 new_definition
  (`MPLXCON_DEF`,
   "MPLXCON(node:word2) = ~(VAL2 node = 3)");;

let MULTIPLEX_DEF =
 new_definition
  (`MULTIPLEX_DEF`,
   "MULTIPLEX(incout:word6,loadin:word6,mplxsel:bool) =
     (mplxsel => incout | loadin)");;

let INCLOGIC_DEF =
 new_definition
  (`INCLOGIC_DEF`,
   "INCLOGIC(count:word6,noinc:bool) =
     let countval = VAL6 count in
     (noinc => count |
               ((countval=63) => WORD6 0 | WORD6(countval+1)))");;

let NEXTNODE_DEF =
 new_definition
  (`NEXTNODE_DEF`,
   "NEXTNODE(node:word2,func:word2,double:bool) =
     let funcnum   = VAL2 func in
     let nodenum   = VAL2 node in
     let fetchnode = WORD2 0   in
     let inc1node  = WORD2 1   in
     let inc2node  = WORD2 2   in
     let loadnode  = WORD2 3   in
     ((nodenum=0) => ((funcnum=0) => fetchnode |
                      (funcnum=1) => loadnode  | inc1node) |
      (nodenum=1) => (double => inc2node | fetchnode)      |
                     fetchnode)");;

let COUNTLOGIC_DEF =
 new_definition
  (`COUNTLOGIC_DEF`,
   "COUNTLOGIC((count,double,node):`major,loadin:word6,func:word2) =
     let twice = EL 0 (BITS2 func) in
     (MULTIPLEX(INCLOGIC(count,INCCON node),loadin,MPLXCON node),
      twice,
      NEXTNODE(node,func,double))");;

close_theory();;
```

```
%============================================================================%
% File for setting up the theory `HOST` which contains the functions         %
% defining the host machine.                                                 %
%============================================================================%

new_theory `HOST`;;

%----------------------------------------------------------------------------%
% The theory `WORD_WIDTHS` contains the definition of "ARB:*" which is an    %
% arbitrary value used as the final arm of conditionals when RSRE don't give %
% any. It also sets up the word widths that are used.                        %
%----------------------------------------------------------------------------%

new_parent `WORD_WIDTHS`;;

%----------------------------------------------------------------------------%
%            Specification of the host machine (pages 6 - 8)                 %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% The host machine is a level of description devised by RSRE. It is          %
% specified as a state-transition function for a low level machine. A single %
% step of the behavioural specifications is implemented by several steps of  %
% the host machine as shown by fig. 3.                                       %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% A state of the host machine consists of the contents of the `latch for     %
% count' (of type ":word6"), the `latch for double' (of type ":bool") and    %
% the `latch for node' (of type ":word2"). Such a state is a triple          %
% "(count,double,node):`major", where major is defined below.                %
%----------------------------------------------------------------------------%

let major = ":word6#bool#word2";;

%----------------------------------------------------------------------------%
% We now give HOL definitions corresponding to the functions defined on      %
% pages 6-8.                                                                 %
%----------------------------------------------------------------------------%

let ADD1_DEF =
 new_definition
  (`ADD1_DEF`,
   "ADD1(x:word6) =
    let xval = VAL6 x in
    ((xval=63) => WORD6 0 | WORD6(xval+1))");;

let FETCH_DEF =
 new_definition
  (`FETCH_DEF`,
   "FETCH(count:word6,double:bool,loadin:word6,func:word2) =
     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))");;

let LOAD_DEF =
 new_definition
  (`LOAD_DEF`,
   "LOAD(count,double,loadin,func) =
     let twice     = EL 0 (BITS2 func) in
     let fetchnode = WORD2 0           in
     (loadin,twice,fetchnode)");;

let INC1_DEF =
 new_definition
  (`INC1_DEF`,
   "INC1(count:word6,double:bool,loadin:word6,func:word2) =
     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))");;
```

```
let INC2_DEF =
 new_definition
  (`INC2_DEF`,
   "INC2(count:word6,double:bool,loadin:word6,func:word2) =
     let twice     = EL 0 (BITS2 func) in
     let fetchnode = WORD2 0           in
     (ADD1 count,twice,fetchnode)");;

let NEXT_DEF =
 new_definition
  (`NEXT_DEF`,
   "NEXT((count,double,node):`major,loadin:word6,func:word2) =
     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) |
                     ARB)");;

close_theory();;
```

```
%============================================================================%
% File for setting up the theory `COUNTER` which contains the behavioural    %
% specification of the counter.                                              %
%============================================================================%

new_theory `COUNTER`;;

%----------------------------------------------------------------------------%
% The theory `WORD_WIDTHS` contains the definition of "ARB:*" which is an    %
% arbitrary value used as the final arm of conditionals when RSRE don't give %
% any. It also declares that word widths of 2 and 6 will be used.            %
%----------------------------------------------------------------------------%

new_parent `WORD_WIDTHS`;;

%----------------------------------------------------------------------------%
%          Behavioural specification of the counter (page 3)                 %
%----------------------------------------------------------------------------%

let COUNTER_DEF =
 new_definition
  (`COUNTER_DEF`,
   "COUNTER(count:word6,loadin:word6,func:word2) =
     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))       |
                     ARB)");;

close_theory();;
```

```
%============================================================================%
%  Statement and equivalence of the definition of INCCON (INCCON_DEF),       %
%  the specification of INCCON (INCCON_SPEC), and the implementation         %
%  of INCCON (INCCON_IMP).                                                   %
%============================================================================%

new_theory `INCCON_CORRECT`;;

new_parent `GATES`;;

new_parent `DESIGN`;;

let V2_DEF = definition `WORD_WIDTHS` `V2_DEF`;;

%----------------------------------------------------------------------------%
%                    Implementation of INCCON                                %
%----------------------------------------------------------------------------%

new_definition
    (`INCCON_IMP_DEF`,
     "INCCON_IMP((n0:bool),(n1:bool),(b:bool)) = NOR(n0,n1,b)");;

%----------------------------------------------------------------------------%
%                    Specification of INCCON                                 %
%----------------------------------------------------------------------------%

new_definition
    (`INCCON_SPEC_DEF`,
     "INCCON_SPEC((n0:bool),(n1:bool),(b:bool)) =
      ((n0 = F) /\ (n1 = F)) => (b = T) | (b = F)");;

%----------------------------------------------------------------------------%
%                     Definition of INCCON                                   %
%                                                                            %
%     "INCCON(node:word2) = (VAL2 node = 0)", on DESIGN theory               %
%----------------------------------------------------------------------------%

let INCCON_DEF = definition `DESIGN` `INCCON_DEF`;;

let NOR_DEF = definition `GATES` `NOR_DEF`;;

let INCCON_IMP_DEF  = definition `-` `INCCON_IMP_DEF`
and INCCON_SPEC_DEF = definition `-` `INCCON_SPEC_DEF`;;

%----------------------------------------------------------------------------%
%               Equivalence of Specification and Implementation              %
%----------------------------------------------------------------------------%

let INCCON_SPEC_IMP_EQUIV =
    prove_thm
    (`INCCON_SPEC_IMP_EQUIV`,
     "INCCON_SPEC(n0,n1,b) = INCCON_IMP(n0,n1,b)",
        BOOL_CASES_TAC "n0:bool"
        THEN BOOL_CASES_TAC "n1:bool"
        THEN REWRITE_TAC [INCCON_SPEC_DEF;INCCON_IMP_DEF;NOR_DEF]);;

%----------------------------------------------------------------------------%
%               Relation between Definition and Specification                %
%----------------------------------------------------------------------------%

let INCCON_DEF_SPEC_EQUIV =
    prove_thm
    (`INCCON_DEF_SPEC_EQUIV`,
     "INCCON_SPEC(n0,n1,b) =
      (b = INCCON(V2(n1,n0)))",
        BOOL_CASES_TAC "n0:bool"
        THEN BOOL_CASES_TAC "n1:bool"
        THEN REWRITE_TAC [INCCON_SPEC_DEF;INCCON_DEF;V2_DEF]
        THEN CONV_TAC(V_CONV THENC WORD_CONV THENC VAL_CONV
                      THENC EQ_CONV)
        THEN REWRITE_TAC[]);;


close_theory();;
```

```
%============================================================================%
%  Statement and equivalence of the definition of MPLXCON (MPLXCON_DEF),     %
%  the specification of MPLXCON (MPLXCON_SPEC), and the implementation       %
%  of MPLXCON (MPLXCON_IMP).                                                 %
%============================================================================%

new_theory `MPLXCON_CORRECT`;;

new_parent `GATES`;;

new_parent `DESIGN`;;

let V2_DEF = definition `WORD_WIDTHS` `V2_DEF`;;

%----------------------------------------------------------------------------%
%                      Implementation of MPLXCON                             %
%----------------------------------------------------------------------------%

new_definition
    (`MPLXCON_IMP_DEF`,
     "MPLXCON_IMP((n0:bool),(n1:bool),(b:bool)) = NAND(n0,n1,b)");;

%----------------------------------------------------------------------------%
%                      Specification of MPLXCON                              %
%----------------------------------------------------------------------------%

new_definition
    (`MPLXCON_SPEC_DEF`,
     "MPLXCON_SPEC((n0:bool),(n1:bool),(b:bool)) =
      ((n0 = T) /\ (n1 = T)) => (b = F) | (b = T)");;

%----------------------------------------------------------------------------%
%                        Definition of MPLXCON                               %
%                                                                            %
%             "MPLXCON(node:word2) = ~(VAL2 node = 3)" on DESIGN theory      %
%----------------------------------------------------------------------------%

let NAND_DEF = definition `GATES` `NAND_DEF`;;

let MPLXCON_IMP_DEF  = definition `-` `MPLXCON_IMP_DEF`
and MPLXCON_SPEC_DEF = definition `-` `MPLXCON_SPEC_DEF`
and MPLXCON_DEF      = definition `DESIGN` `MPLXCON_DEF`;;

%----------------------------------------------------------------------------%
%              Equivalence of Specification and Implementation               %
%----------------------------------------------------------------------------%

let MPLXCON_SPEC_IMP_EQUIV =
    prove_thm
    (`MPLXCON_SPEC_IMP_EQUIV`,
     "MPLXCON_SPEC(n0,n1,b) = MPLXCON_IMP(n0,n1,b)",
         BOOL_CASES_TAC "n0:bool"
         THEN BOOL_CASES_TAC "n1:bool"
         THEN REWRITE_TAC [MPLXCON_SPEC_DEF;MPLXCON_IMP_DEF;NAND_DEF]);;

%----------------------------------------------------------------------------%
%               Relation between Definition and Specification                %
%----------------------------------------------------------------------------%

let MPLXCON_DEF_SPEC_EQUIV =
    prove_thm
    (`MPLXCON_DEF_SPEC_EQUIV`,
     "MPLXCON_SPEC(n0,n1,b) =
      (b = MPLXCON(V2(n1,n0)))",
         BOOL_CASES_TAC "n0:bool"
         THEN BOOL_CASES_TAC "n1:bool"
         THEN REWRITE_TAC [MPLXCON_SPEC_DEF;MPLXCON_DEF;V2_DEF]
         THEN CONV_TAC(V_CONV THENC WORD_CONV THENC VAL_CONV
                       THENC EQ_CONV)
         THEN REWRITE_TAC[]);;

close_theory();;
```

```
%============================================================================%
%  Statement and equivalence of the definition of MULTIPLEX (MULTIPLEX_DEF), %
%  the specification of MULTIPLEX (MULTIPLEX_SPEC), and the implementation   %
%  of MULTIPLEX (MULTIPLEX_IMP).                                             %
%============================================================================%


new_theory`MULTIPLEX_CORRECT`;;

new_parent`GATES`;;

new_parent `DESIGN`;;

let V6_DEF = definition `WORD_WIDTHS` `V6_DEF`;;

loadt `~/counter/utilities`;;

%----------------------------------------------------------------------------%
%  The following three facts are required to cope with subgoals of the       %
%  form "((b0,b1,b2,b3,b4,b5) = (b0',b1',b2',b3',b4',b5')) =                 %
%        (WORD6(V(b0,b1,b2,b3,b4,b5)) = WORD6(V(b0',b1',b2',b3',b4',b5')))". %
%----------------------------------------------------------------------------%

let LIST_AX1 =  axiom `WORD_WIDTHS` `LIST_AX1`;;

let WORD6_AX2 = axiom `WORD_WIDTHS` `WORD6_AX2`;;

let V_AX1 =     axiom `WORD_WIDTHS` `V_AX1`;;

%----------------------------------------------------------------------------%
%                 The Implementation of MULTIPLEX                            %
%----------------------------------------------------------------------------%

new_definition
   (`MULTIPLEX_IMP_DEF`,
    "MULTIPLEX_IMP(c0,c1,c2,c3,c4,c5,
                   10,11,12,13,14,15,b,
                   c0',c1',c2',c3',c4',c5') =
     ?x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 b'.
      INV(b,b')                 /\
      NAND(15,b',x1)            /\
      NAND(c5,b,x2)             /\
      NAND(14,b',x3)            /\
      NAND(c4,b,x4)             /\
      NAND(13,b',x5)            /\
      NAND(c3,b,x6)             /\
      NAND(12,b',x7)            /\
      NAND(c2,b,x8)             /\
      NAND(11,b',x9)            /\
      NAND(c1,b,x10)            /\
      NAND(10,b',x11)           /\
      NAND(c0,b,x12)            /\
      NAND(x1,x2,c5')           /\
      NAND(x3,x4,c4')           /\
      NAND(x5,x6,c3')           /\
      NAND(x7,x8,c2')           /\
      NAND(x9,x10,c1')          /\
      NAND(x11,x12,c0')");;


%----------------------------------------------------------------------------%
%                     The Specification of MULTIPLEX                         %
%----------------------------------------------------------------------------%

new_definition
   (`MULTIPLEX_SPEC_DEF`,
    "MULTIPLEX_SPEC(c0,c1,c2,c3,c4,c5,
                   10,11,12,13,14,15,b,
                   c0',c1',c2',c3',c4',c5') =
     ((c0',c1',c2',c3',c4',c5') = (b => (c0,c1,c2,c3,c4,c5) |
                                        (10,11,12,13,14,15)))");;

%----------------------------------------------------------------------------%
%                      The Definition of MULTIPLEX                           %
%                                                                            %
%   "MULTIPLEX(incout:word6,loadin:word6,mplxsel:bool) =                     %
%     (mplxsel => incout | loadin)"         on DESIGN theory                 %
```

```
%----------------------------------------------------------------------------%

let NAND_DEF = definition `GATES` `NAND_DEF`;;
let INV_DEF  = definition `GATES` `INV_DEF`;;

let MULTIPLEX_IMP_DEF  = definition `MULTIPLEX_CORRECT` `MULTIPLEX_IMP_DEF`;;
let MULTIPLEX_SPEC_DEF = definition `MULTIPLEX_CORRECT` `MULTIPLEX_SPEC_DEF`;;
let MULTIPLEX_DEF      = definition `DESIGN` `MULTIPLEX_DEF`;;

let MULTIPLEX_EXPAND = EXPAND [NAND_DEF;INV_DEF] MULTIPLEX_IMP_DEF;;


%----------------------------------------------------------------------------%
%         The equivalence of Specification and Implementation                %
%----------------------------------------------------------------------------%

let MULTIPLEX_SPEC_IMP_EQUIV =
 prove_thm
   (`MULTIPLEX_SPEC_IMP_EQUIV`,
    "MULTIPLEX_SPEC(c0,c1,c2,c3,c4,c5,
                    l0,l1,l2,l3,l4,l5,b,
                    c0',c1',c2',c3',c4',c5') =
     MULTIPLEX_IMP(c0,c1,c2,c3,c4,c5,
                    l0,l1,l2,l3,l4,l5,b,
                    c0',c1',c2',c3',c4',c5')",
    REWRITE_TAC[MULTIPLEX_SPEC_DEF;MULTIPLEX_EXPAND]
    THEN BOOL_CASES_TAC "b:bool"
    THEN REWRITE_TAC[PAIR_SPLIT]
    THEN CONJ_SET_TAC);;


%----------------------------------------------------------------------------%
%             The relation between Definition and Specification              %
%----------------------------------------------------------------------------%

let MULTIPLEX_DEF_SPEC_EQUIV =
 prove_thm
   (`MULTIPLEX_DEF_SPEC_EQUIV`,
    "MULTIPLEX_SPEC(c0,c1,c2,c3,c4,c5,l0,l1,l2,l3,l4,l5,
                     b,c0',c1',c2',c3',c4',c5') =
     (V6(c5',c4',c3',c2',c1',c0') =
      MULTIPLEX(V6(c5,c4,c3,c2,c1,c0),V6(l5,l4,l3,l2,l1,l0),b))",
         REWRITE_TAC[MULTIPLEX_SPEC_DEF;MULTIPLEX_DEF;V6_DEF]
         THEN BOOL_CASES_TAC "b:bool"
         THEN CONV_TAC(V_CONV THENC WORD_CONV THENC VAL_CONV
                       THENC EQ_CONV)
         THEN REWRITE_TAC[LIST_AX1;WORD6_AX2;V_AX1;PAIR_SPLIT]
         THEN CONJ_SET_TAC);;

close_theory();;
```

```
%============================================================================%
%  Statement and equivalence of the definition of INCLOGIC (INCLOGIC_DEF),   %
%  the specification of INCLOGIC (INCLOGIC_SPEC), and the implementation     %
%  of INCLOGIC (INCLOGIC_IMP).                                               %
%============================================================================%


new_theory `INCLOGIC_CORRECT`;;

loadt`~/counter/utilities`;;

new_parent `GATES`;;

new_parent `DESIGN`;;

let V6_DEF = definition `WORD_WIDTHS` `V6_DEF`;;

%----------------------------------------------------------------------------%
%  The following 9 facts are required for proving INCLOGIC_DEF_SPEC_EQUIV.   %
%  The first two axioms are for reasoning about tuples.                      %
%  The next five  are required to cope with subgoals of the form             %
%  "...WORD6(...)... = ...WORD6(...)...", "...V(...)... = ...V(...)...", and %
%  "...[...]... = ...[...]...".                                              %
%  The last two show certain inverses.                                       %
%----------------------------------------------------------------------------%

let TUPLE6_AX =       axiom `WORD_WIDTHS` `TUPLE6_AX`;;

let TUPLE6_AX2 =      axiom `WORD_WIDTHS` `TUPLE6_AX2`;;

let LIST_AX1 =        axiom `WORD_WIDTHS` `LIST_AX1`;;

let WORD6_AX1 =       axiom `WORD_WIDTHS` `WORD6_AX1`;;

let WORD6_AX2 =       axiom `WORD_WIDTHS` `WORD6_AX2`;;

let V_AX1 =           axiom `WORD_WIDTHS` `V_AX1`;;

let V_AX2 =           axiom `WORD_WIDTHS` `V_AX2`;;

let VAL6_WORD6_AX2 =  theorem `WORD_WIDTHS` `VAL6_WORD6_AX2`;;

let BITS6_WORD6_AX =  theorem `WORD_WIDTHS` `BITS6_WORD6_AX`;;

%----------------------------------------------------------------------------%
% The following lemmas are required for the proofs.  LEMMA1 is used in the   %
% proofs of both INCLOGIC_SPEC_IMP_EQUIV and INCLOGIC_DEF_SPEC_EQUIV.        %
% Given an assumption "t = T" then LEMMA1 allows resolution                  %
% to reduce that to "t". (This should be on a separate logic theory.)        %
% LEMMA2 is the corresponding theorem for "t = F".                           %
% LEMMA2  is used in INCLOGIC_DEF_SPEC_EQUIV and INCLOGIC_SPEC_IMP_EQUIV.    %
% The rest of the lemmas are all for INCLOGIC_DEF_SPEC.                      %
% LEMMA3  says "(x,y) = (x',y') ==> (x = x') /\ (y = y')"  which             %
% can be used  with IMP_RES_TAC; PAIR_SPLIT, on utilities.ml, has the wrong  %
% (equational) form for resolution. LEMMA3 could be with code for PAIR_SPLIT %
% on utilities.ml.                                                           %
%----------------------------------------------------------------------------%

let LEMMA1 =
 TAC_PROOF
  (([],"(t = T) ==> t"), REWRITE_TAC[]);;


let LEMMA2 =
   TAC_PROOF(([],"(t = F) ==> ~t"),REWRITE_TAC[]);;

let LEMMA3 = GEN_ALL(fst(EQ_IMP_RULE(SPEC_ALL PAIR_SPLIT)));;

%----------------------------------------------------------------------------%
% LEMMA4, to be used as a rewrite rule, says                                 %
% "#000000 = WORD6(V[F;F;F;F;F;F])".  Similarly, LEMMA5 says that            %
% "63 = V[T;T;T;T;T;T]".                                                     %
% LEMMA6, to be used by resolution, says that "~((b0,b1,b2,b3,b4,b5) =       %
% (T,T,T,T,T,T)) = ~(V[b0;b1;b2;b3;b4;b5]=63)".                              %
%----------------------------------------------------------------------------%

let LEMMA4 = theorem `WORD_WIDTHS` `LEMMA4`;;
```

```
let LEMMA5 = theorem `WORD_WIDTHS` `LEMMA5`;;

let LEMMA6 = theorem `WORD_WIDTHS` `LEMMA6`;;

%----------------------------------------------------------------------------%
%                    The specification of INCLOGIC:                          %
%----------------------------------------------------------------------------%

new_definition(`INCLOGIC_SPEC_DEF`,
              "INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,
                             c0',c1',c2',c3',c4',c5') =
               (c5',c4',c3',c2',c1',c0') =
                     (b => (c5,c4,c3,c2,c1,c0) |
                     ((c5,c4,c3,c2,c1,c0) = (T,T,T,T,T,T)) => (F,F,F,F,F,F) |
                     TUPLE6(BITS6(WORD6(V[c5;c4;c3;c2;c1;c0] + 1))))");;

%----------------------------------------------------------------------------%
%                    The implementation of INCLOGIC:                         %
%----------------------------------------------------------------------------%

new_definition(`INCLOGIC_IMP_DEF`,
              "INCLOGIC_IMP(c0,c1,c2,c3,c4,c5,b,
                             c0',c1',c2',c3',c4',c5') =
                ?b' x1 x2 x3 x3' x4 x5.
                 INV(b,b')                /\
                 INV(x3,x3')              /\
                 NAND(b',c0,x1)           /\
                 NAND3(b',c0,c1,x2)       /\
                 NAND4(b',c0,c1,c2,x3)    /\
                 NAND(x3',c3,x4)          /\
                 NAND3(x3', c3,c4,x5)     /\
                 XNOR(c0,b,c0')           /\
                 XNOR(c1,x1,c1')          /\
                 XNOR(c2,x2,c2')          /\
                 XNOR(c3,x3,c3')          /\
                 XNOR(c4,x4,c4')          /\
                 XNOR(c5,x5,c5')");;

%----------------------------------------------------------------------------%
%                        The definition of INCLOGIC:                         %
%                                                                            %
%   "INCLOGIC(count:word6,noinc:bool) =                                      %
%     let countval = VAL6 count in                                           %
%     (noinc => count |                                                      %
%                ((countval=63) => WORD6 0 | WORD6(countval+1)))"            %
%   on DESIGN theory                                                         %
%----------------------------------------------------------------------------%

let prims  =
    map (definition `GATES`)
        (words `INV_DEF NAND_DEF NAND3_DEF NAND4_DEF XNOR_DEF`);;

let INCLOGIC_SPEC_DEF = definition `-` `INCLOGIC_SPEC_DEF`
and INCLOGIC_IMP_DEF  = definition `-` `INCLOGIC_IMP_DEF`
and INCLOGIC_DEF      = definition `DESIGN` `INCLOGIC_DEF` ;;

let INCLOGIC_EXPAND = REWRITE_RULE[](EXPAND prims INCLOGIC_IMP_DEF);;


%----------------------------------------------------------------------------%
%          The equivalence of specification and implementation:              %
%                                                                            %
% The following is proved as two separate cases, each with 32 subcases,     %
% to prevent the system running out of space. The variable c5 is             %
% replaced by T and F explicitly.  Of the 64 cases based on c0...c5,         %
% only one is distinct: when all ci = T, that case will have been            %
% caught by the conditional in INCLOGIC_SPEC, so proof is by contradiction.  %
% That accounts for the final                                                %
% IMP_RES_TAC and RES_TAC in the INCLOGIC_THM_T case below.                  %
% These are very slow proofs.                                                %
%----------------------------------------------------------------------------%

let INCLOGIC_THM_T =
 prove_thm
  (`INCLOGIC_THM_T`,
   "INCLOGIC_SPEC(c0,c1,c2,c3,c4,T,b,c0',c1',c2',c3',c4',c5') =
    INCLOGIC_IMP (c0,c1,c2,c3,c4,T,b,c0',c1',c2',c3',c4',c5')",
    REWRITE_TAC[INCLOGIC_SPEC_DEF;INCLOGIC_EXPAND]
```

```
    THEN BOOL_CASES_TAC "b:bool"
    THEN REWRITE_TAC[PAIR_SPLIT]
    THEN (CONJ_SET_TAC ORELSE ALL_TAC)
    THEN COND_CASES_TAC
    THEN IMP_RES_TAC LEMMA1
    THEN ASM_REWRITE_TAC[PAIR_SPLIT]
    THEN (CONJ_SET_TAC ORELSE ALL_TAC)
    THEN ASM_BOOL_CASES_TAC "c0:bool"
    THEN ASM_BOOL_CASES_TAC "c1:bool"
    THEN ASM_BOOL_CASES_TAC "c2:bool"
    THEN ASM_BOOL_CASES_TAC "c3:bool"
    THEN ASM_BOOL_CASES_TAC "c4:bool"
    THEN PURE_ASM_REWRITE_TAC[]
    THEN CONV_TAC
         (V_CONV
          THENC ADD_CONV
          THENC (WORD_CONV ORELSEC ALL_CONV)
          THENC BITS_CONV
          THENC EQ_CONV)
    THEN REWRITE_TAC[TUPLE6_AX;PAIR_SPLIT]
    THEN (CONJ_SET_TAC ORELSE ALL_TAC)
    THEN IMP_RES_TAC LEMMA2
    THEN RES_TAC);;

let INCLOGIC_THM_F =
 prove_thm
  (`INCLOGIC_THM_F`,
   "INCLOGIC_SPEC(c0,c1,c2,c3,c4,F,b,c0',c1',c2',c3',c4',c5') =
    INCLOGIC_IMP (c0,c1,c2,c3,c4,F,b,c0',c1',c2',c3',c4',c5')",
    REWRITE_TAC[INCLOGIC_SPEC_DEF;INCLOGIC_EXPAND]
    THEN BOOL_CASES_TAC "b:bool"
    THEN REWRITE_TAC[PAIR_SPLIT]
    THEN (CONJ_SET_TAC ORELSE ALL_TAC)
    THEN ASM_BOOL_CASES_TAC "c0:bool"
    THEN ASM_BOOL_CASES_TAC "c1:bool"
    THEN ASM_BOOL_CASES_TAC "c2:bool"
    THEN ASM_BOOL_CASES_TAC "c3:bool"
    THEN ASM_BOOL_CASES_TAC "c4:bool"
    THEN PURE_ASM_REWRITE_TAC[]
    THEN CONV_TAC
         (V_CONV
          THENC ADD_CONV
          THENC (WORD_CONV ORELSEC ALL_CONV)
          THENC BITS_CONV
          THENC EQ_CONV)
    THEN REWRITE_TAC[TUPLE6_AX;PAIR_SPLIT]
    THEN (CONJ_SET_TAC ORELSE ALL_TAC));;

%----------------------------------------------------------------------------%
% Finally, the two subcases are joined together:                             %
%----------------------------------------------------------------------------%

let INCLOGIC_SPEC_IMP_EQUIV =
 prove_thm
  (`INCLOGIC_SPEC_IMP_EQUIV`,
   "INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
    INCLOGIC_IMP (c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5')",
    BOOL_CASES_TAC "c5:bool"
    THEN REWRITE_TAC[INCLOGIC_THM_T;INCLOGIC_THM_F]);;

%----------------------------------------------------------------------------%
%           The relation between definition and specification:               %
%                                                                            %
% This is a quick proof but text is too big for EMACS buffer.                %
%----------------------------------------------------------------------------%

let INCLOGIC_DEF_SPEC_EQUIV =
 prove_thm
  (`INCLOGIC_DEF_SPEC_EQUIV`,
   "INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
    (V6(c5',c4',c3',c2',c1',c0') =
     INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b))",
       REWRITE_TAC[INCLOGIC_SPEC_DEF;INCLOGIC_DEF;V6_DEF;LET_DEF]
       THEN BOOL_CASES_TAC "b:bool"
       THEN COND_CASES_TAC
       THEN CONV_TAC(DEPTH_CONV BETA_CONV THENC
                     V_CONV THENC WORD_CONV THENC VAL_CONV
                     THENC EQ_CONV)
       THEN REWRITE_TAC[LIST_AX1;WORD6_AX2;V_AX1;PAIR_SPLIT]
```

```
%----------------------------------------------------------------------------%
% Up to here the tactic solves the first 2 subgoals, where b = T.            %
%----------------------------------------------------------------------------%

       THENL [IMP_RES_TAC LEMMA1
              THEN REPEAT(CHANGED_TAC(IMP_RES_TAC LEMMA3))
              THEN ASM_REWRITE_TAC[TUPLE6_AX]
              THEN CONV_TAC(V_CONV THENC WORD_CONV THENC VAL_CONV
                            THENC EQ_CONV)
              THEN REWRITE_TAC[LEMMA4;LIST_AX1;WORD6_AX2;
                               V_AX1;PAIR_SPLIT];


%----------------------------------------------------------------------------%
% Up to here the tactic solves the third subgoal, where b = F and            %
% ((c5,c4,c3,c2,c1,c0)=(T,T,T,T,T,T))=T.                                     %
%----------------------------------------------------------------------------%

              IMP_RES_TAC LEMMA2
              THEN IMP_RES_TAC LEMMA6
              THEN IMP_RES_TAC WORD6_AX1
              THEN IMP_RES_TAC BITS6_WORD6_AX
              THEN ASM_REWRITE_TAC[VAL6_WORD6_AX2;TUPLE6_AX2;V_AX2]]);;

%----------------------------------------------------------------------------%
% Up to here the tactic solves the fourth subgoal, where b = F and           %
% ((c5,c4,c3,c2,c1,c0)=(T,T,T,T,T,T,))=F.                                    %
%----------------------------------------------------------------------------%


close_theory();;
```

```
%============================================================================%
%  Statement and equivalence of the definition of NEXTNODE (NEXTNODE_DEF),   %
%  the specification of NEXTNODE (NEXTNODE_SPEC), and the implementation     %
%  of NEXTNODE (NEXTNODE_IMP).                                               %
%============================================================================%

new_theory`NEXTNODE_CORRECT`;;

new_parent `GATES`;;

new_parent `DESIGN`;;

let V2_DEF = definition `WORD_WIDTHS` `V2_DEF`;;

loadt `~/counter/utilities`;;

%----------------------------------------------------------------------------%
%               Implementation of NEXTNODE:                                  %
%----------------------------------------------------------------------------%

new_definition
 (`NEXTNODE_IMP_DEF`,
  "NEXTNODE_IMP(n0,n1,f0,f1,double,n0',n1') =
   ?x1 x2 x3 x4 x5 x6.
    NAND(x4,x5,n0')         /\
    NAND(x5,x6,n1')         /\
    NAND(x1,f1,x4)          /\
    NAND3(x1,f0,x2,x5)      /\
    NAND3(n0,double,x3,x6) /\
    NOR(n1,n0,x1)           /\
    INV(f1,x2)              /\
    INV(n1,x3)");;

%----------------------------------------------------------------------------%
%                       Definition of NEXTNODE                               %
%                                                                            %
%   "NEXTNODE(node:word2,func:word2,double:bool) =                           %
%     let funcnum   = VAL2 func in                                           %
%     let nodenum   = VAL2 node in                                           %
%     let fetchnode = WORD2 0   in                                           %
%     let inc1node  = WORD2 1   in                                           %
%     let inc2node  = WORD2 2   in                                           %
%     let loadnode  = WORD2 3   in                                           %
%     ((nodenum=0) => ((funcnum=0)=>fetchnode|                               %
%                      (funcnum=1)=>loadnode | inc1node) |                    %
%      (nodenum=1) => (double=>inc2node|fetchnode)       |                   %
%                     fetchnode)"                                            %
% on DESIGN theory                                                           %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
%                        Specification of NEXTNODE                           %
%----------------------------------------------------------------------------%

new_definition
 (`NEXTNODE_SPEC_DEF`,
  "NEXTNODE_SPEC(n0,n1,f0,f1,double,n0',n1') =
   ((n1',n0') =
   (((n0=F)/\(n1=F)) => (((f1=F)/\(f0=F)) => (F,F) |
                         ((f1=F)/\(f0=T)) => (T,T) |
                                             (F,T)) |
    ((n0=T)/\(n1=F)) => (double => (T,F) | (F,F)) |
                        (F,F)))");;

let NOR_DEF    = definition `GATES` `NOR_DEF`
and NAND_DEF   = definition `GATES` `NAND_DEF`
and NAND3_DEF  = definition `GATES` `NAND3_DEF`
and INV_DEF    = definition `GATES` `INV_DEF`;;

let NEXTNODE_IMP_DEF   = definition `NEXTNODE_CORRECT` `NEXTNODE_IMP_DEF`
and NEXTNODE_SPEC_DEF  = definition `NEXTNODE_CORRECT` `NEXTNODE_SPEC_DEF`
and NEXTNODE_DEF       = definition `DESIGN` `NEXTNODE_DEF`;;

let prims = [NOR_DEF;NAND_DEF;NAND3_DEF;INV_DEF];;

let NEXTNODE_EXPAND = EXPAND prims NEXTNODE_IMP_DEF;;

%----------------------------------------------------------------------------%
```

```
%             Equivalence of Specification and Implementation                 %
%                                                                             %
% This is a slow proof.                                                       %
%-----------------------------------------------------------------------------%

let NEXTNODE_SPEC_IMP_EQUIV =
 prove_thm
  (`NEXTNODE_SPEC_IMP_EQUIV`,
   "NEXTNODE_SPEC(n0,n1,f0,f1,double,n0',n1') =
     NEXTNODE_IMP(n0,n1,f0,f1,double,n0',n1')",
   REWRITE_TAC[NEXTNODE_SPEC_DEF;NEXTNODE_EXPAND]
   THEN BOOL_CASES_TAC "n0:bool"
   THEN BOOL_CASES_TAC "n1:bool"
   THEN BOOL_CASES_TAC "double:bool"
   THEN BOOL_CASES_TAC "f0:bool"
   THEN BOOL_CASES_TAC "f1:bool"
   THEN REWRITE_TAC[PAIR_SPLIT]
   THEN CONJ_SET_TAC);;

%-----------------------------------------------------------------------------%
%             The relation between Definition and Specification               %
%                                                                             %
% The following is proved in four separate cases, each with 32 subcases,      %
% to prevent system running out of space. The four lemmas together            %
% are used to prove NEXTNODE_THM2. Fairly slow proofs.                        %
%-----------------------------------------------------------------------------%

let NEXTNODE_THM2_TT =
 prove_thm
  (`NEXTNODE_THM2_TT`,
   "NEXTNODE_SPEC(n0,n1,f0,f1,double,T,T) =
     (V2(T,T) = NEXTNODE(V2(n1,n0),V2(f1,f0),double))",
   PURE_REWRITE_TAC[NEXTNODE_DEF;NEXTNODE_SPEC_DEF;LET_DEF;V2_DEF]
   THEN CONV_TAC(DEPTH_CONV BETA_CONV)
   THEN BOOL_CASES_TAC "n0:bool"
   THEN BOOL_CASES_TAC "n1:bool"
   THEN BOOL_CASES_TAC "double:bool"
   THEN BOOL_CASES_TAC "f0:bool"
   THEN BOOL_CASES_TAC "f1:bool"
   THEN CONV_TAC(V_CONV THENC WORD_CONV THENC VAL_CONV THENC EQ_CONV)
   THEN REWRITE_TAC[]
   THEN CONV_TAC EQ_CONV
   THEN REWRITE_TAC[PAIR_SPLIT]);;

let NEXTNODE_THM2_TF =
 prove_thm
  (`NEXTNODE_THM2_TF`,
   "NEXTNODE_SPEC(n0,n1,f0,f1,double,F,T) =
     (V2(T,F) = NEXTNODE(V2(n1,n0),V2(f1,f0),double))",
   PURE_REWRITE_TAC[NEXTNODE_DEF;NEXTNODE_SPEC_DEF;LET_DEF;V2_DEF]
   THEN CONV_TAC(DEPTH_CONV BETA_CONV)
   THEN BOOL_CASES_TAC "n0:bool"
   THEN BOOL_CASES_TAC "n1:bool"
   THEN BOOL_CASES_TAC "double:bool"
   THEN BOOL_CASES_TAC "f0:bool"
   THEN BOOL_CASES_TAC "f1:bool"
   THEN CONV_TAC(V_CONV THENC WORD_CONV THENC VAL_CONV THENC EQ_CONV)
   THEN REWRITE_TAC[]
   THEN CONV_TAC EQ_CONV
   THEN REWRITE_TAC[PAIR_SPLIT]);;

let NEXTNODE_THM2_FT =
 prove_thm
  (`NEXTNODE_THM2_FT`,
   "NEXTNODE_SPEC(n0,n1,f0,f1,double,T,F) =
     (V2(F,T) = NEXTNODE(V2(n1,n0),V2(f1,f0),double))",
   PURE_REWRITE_TAC[NEXTNODE_DEF;NEXTNODE_SPEC_DEF;LET_DEF;V2_DEF]
   THEN CONV_TAC(DEPTH_CONV BETA_CONV)
   THEN BOOL_CASES_TAC "n0:bool"
   THEN BOOL_CASES_TAC "n1:bool"
   THEN BOOL_CASES_TAC "double:bool"
   THEN BOOL_CASES_TAC "f0:bool"
   THEN BOOL_CASES_TAC "f1:bool"
   THEN CONV_TAC(V_CONV THENC WORD_CONV THENC VAL_CONV THENC EQ_CONV)
   THEN REWRITE_TAC[PAIR_SPLIT]
   THEN CONV_TAC EQ_CONV
   THEN REWRITE_TAC[PAIR_SPLIT]);;

let NEXTNODE_THM2_FF =
```

```
 prove_thm
  (`NEXTNODE_THM2_FF`,
   "NEXTNODE_SPEC(n0,n1,f0,f1,double,F,F) =
     (V2(F,F) = NEXTNODE(V2(n1,n0),V2(f1,f0),double))",
   PURE_REWRITE_TAC[NEXTNODE_DEF;NEXTNODE_SPEC_DEF;LET_DEF;V2_DEF]
   THEN CONV_TAC(DEPTH_CONV BETA_CONV)
   THEN BOOL_CASES_TAC "n0:bool"
   THEN BOOL_CASES_TAC "n1:bool"
   THEN BOOL_CASES_TAC "double:bool"
   THEN BOOL_CASES_TAC "f0:bool"
   THEN BOOL_CASES_TAC "f1:bool"
   THEN CONV_TAC(V_CONV THENC WORD_CONV THENC VAL_CONV THENC EQ_CONV)
   THEN REWRITE_TAC[]
   THEN CONV_TAC EQ_CONV
   THEN REWRITE_TAC[PAIR_SPLIT]);;

%----------------------------------------------------------------------------%
% Finally, the four lemmas are combined.                                     %
%----------------------------------------------------------------------------%

let NEXTNODE_DEF_SPEC_EQUIV =
 prove_thm
  (`NEXTNODE_DEF_SPEC_EQUIV`,
   "NEXTNODE_SPEC(n0,n1,f0,f1,double,n0',n1') =
     (V2(n1',n0') = NEXTNODE(V2(n1,n0),V2(f1,f0),double))",
   BOOL_CASES_TAC "n0':bool"
   THEN BOOL_CASES_TAC "n1':bool"
   THEN REWRITE_TAC
        [NEXTNODE_THM2_TT;NEXTNODE_THM2_TF;NEXTNODE_THM2_FT;
         NEXTNODE_THM2_FF]);;


close_theory();;
```

```
%============================================================================%
% In this theory, we prove the equivalence of NEXT and COUNTLOGIC.           %
% (See p. 6, 8, 21 and D-1 in RSRE report.)                                  %
%============================================================================%

new_theory `DESIGN_CORRECT`;;

new_parent `DESIGN`;;

new_parent `HOST`;;

loadt `~/counter/utilities`;;

let WORD2_CASES_AX = axiom `WORD_WIDTHS` `WORD2_CASES_AX`;;

let [INCCON_DEF;MPLXCON_DEF;MULTIPLEX_DEF;INCLOGIC_DEF;NEXTNODE_DEF;
     COUNTLOGIC_DEF] =
    map (definition `DESIGN`)[`INCCON_DEF`;`MPLXCON_DEF`;`MULTIPLEX_DEF`;
                              `INCLOGIC_DEF`;`NEXTNODE_DEF`;
                              `COUNTLOGIC_DEF`];;

let [ADD1_DEF;FETCH_DEF;LOAD_DEF;INC1_DEF;INC2_DEF;NEXT_DEF] =
    map(definition `HOST`)   [`ADD1_DEF`;`FETCH_DEF`;`LOAD_DEF`;`INC1_DEF`;
                              `INC2_DEF`;`NEXT_DEF`];;

%----------------------------------------------------------------------------%
% We first prove "NEXT((count,double,node),loadin,func) =                    %
%                 COUNTLOGIC((count,double,node),loadin,func)",              %
% WITH arguments present, so as to be able to unwind the definitions:        %
%----------------------------------------------------------------------------%

let NEXT_COUNTLOGIC_EQUIV' =
  prove_thm(`NEXT_COUNTLOGIC_EQUIV'`,
            "NEXT      ((count,double,node),loadin,func) =
             COUNTLOGIC((count,double,node),loadin,func)",
            REWRITE_TAC[COUNTLOGIC_DEF;NEXT_DEF;ADD1_DEF;LOAD_DEF;
                        INC1_DEF;INC2_DEF;FETCH_DEF;INCCON_DEF;
                        MPLXCON_DEF;MULTIPLEX_DEF;INCLOGIC_DEF;NEXTNODE_DEF]
            THEN REWRITE_TAC[LET_DEF]
            THEN CONV_TAC(DEPTH_CONV BETA_CONV)
            THEN GDISJ_CASES_TAC(SPEC "node:word2" WORD2_CASES_AX)
            THEN CONV_TAC(VAL_CONV THENC EQ_CONV)
            THEN REWRITE_TAC[]
            THENL [GDISJ_CASES_TAC(SPEC "func:word2" WORD2_CASES_AX)
                   THEN CONV_TAC(VAL_CONV THENC EQ_CONV)
                   THEN REWRITE_TAC[];
                   BOOL_CASES_TAC "double:bool"
                   THEN GDISJ_CASES_TAC(SPEC "func:word2" WORD2_CASES_AX)
                   THEN CONV_TAC(BITS_CONV THENC EL_CONV)
                   THEN REWRITE_TAC[]] );;

%----------------------------------------------------------------------------%
% We next prove simply that "NEXT = COUNTLOGIC", as this is a more conven-   %
% ient form for use in COUNTER_CORRECT.th; otherwise there is a problem of   %
% unifying varstructs which is beyond the scope of rewriting and becomes     %
% awkward. A forward proof is easiest.                                       %
%                                                                            %
% th1 =                                                                      %
%  |- !count double node loadin func.                                       %
%       NEXT((count,double,node),loadin,func) =                              %
%       COUNTLOGIC((count,double,node),loadin,func)                          %
%  th2 =                                                                     %
%  |- NEXT                                                                  %
%     ((FST(FST x),FST(SND(FST x)),SND(SND(FST x))),FST(SND x),SND(SND x)) = %
%     COUNTLOGIC                                                             %
%     ((FST(FST x),FST(SND(FST x)),SND(SND(FST x))),FST(SND x),SND(SND x))   %
%  th3 = |- NEXT x = COUNTLOGIC x                                            %
%  th4 = |- !x. NEXT x = COUNTLOGIC x                                        %
%  th5 = |- NEXT = COUNTLOGIC                                                %
%----------------------------------------------------------------------------%

let th1 = GEN "count:word6"(GEN "double:bool"(GEN "node:word2"
          (GEN "loadin:word6"(GEN "func:word2" NEXT_COUNTLOGIC_EQUIV'))));;

let th2 = let x = "x:(word6 # (bool # word2)) # (word6 # word2)"
          in SPECL ["FST(FST `x)";
                    "FST(SND(FST `x))";
                    "SND(SND(FST `x))";
                    "FST(SND `x)";
```

```
                    "SND(SND `x)"] th1;;

let th3 = REWRITE_RULE[PAIR] th2;;

let th4 = let x = "x:(word6 # (bool # word2)) # (word6 # word2)"
          in GEN x th3;;

let th5 = EXT th4;;

let NEXT_COUNTLOGIC_EQUIV =
  prove_thm(`NEXT_COUNTLOGIC_EQUIV`,
            "NEXT = COUNTLOGIC",
            REWRITE_TAC[th5]
);;

close_theory();;
```

```
%============================================================================%
% This file contains commands for proving the theorem HOST_CORRECT which     %
% shows that sequences of steps of the host machine correspond to single     %
% steps of the counter.                                                      %
%============================================================================%

new_theory `HOST_CORRECT`;;

new_parent `COUNTER`;;
new_parent `HOST`;;

%----------------------------------------------------------------------------%
% The file `utilities` contains some general theorems and rules that are     %
% used later.                                                                %
%----------------------------------------------------------------------------%

loadt `~/counter/utilities`;;

%----------------------------------------------------------------------------%
% The state of the host machine is a triple of type ":`major".               %
% The components of this triple are the contents of the registers COUNT,     %
% DOUBLE and NODE shown in fig. 4. It is convenient to define constants to   %
% extract these components.                                                  %
%----------------------------------------------------------------------------%

let major = ":word6#bool#word2";;

let COUNT_DEF =
 new_definition
  (`COUNT_DEF`, "COUNT(s:`major) = FST s");;

let DOUBLE_DEF =
 new_definition
  (`DOUBLE_DEF`, "DOUBLE(s:`major) = FST(SND s)");;

let NODE_DEF =
 new_definition
  (`NODE_DEF`, "NODE(s:`major) = SND(SND s)");;

%----------------------------------------------------------------------------%
% The theorem MAJOR_LEMMA is |- !s. s = (COUNT s, DOUBLE s, NODE s)          %
%----------------------------------------------------------------------------%

let MAJOR_LEMMA =
 prove_thm
  (`MAJOR_LEMMA`,
   "!s:`major. s = (COUNT s, DOUBLE s, NODE s)",
   REWRITE_TAC[COUNT_DEF;DOUBLE_DEF;NODE_DEF]);;

%----------------------------------------------------------------------------%
% The host is correct if it has two properties.                              %
%                                                                            %
%    1. Termination: any sequence of host machine states starting in a       %
%       primary state (i.e. one with node=#00) eventually reaches another    %
%       primary state.                                                       %
%                                                                            %
%    2. Correctness: the sequence of primary states of the host corresponds  %
%       to the state transitions defined by the behavioural specification    %
%       function COUNTER.                                                    %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% To formalize the statement of termination and correctness it is convenient %
% to define the function "NEXTTIME" such that "NEXTTIME(t,t')f" is true if   %
% and only if t' is the next time after t that f is true.                    %
%                                                                            %
% For example "NEXTTIME(t,t')(\x. NODE(state_sig x)=#00)" is true iff t' is  %
% the first time after t that the NODE value of "state_sig" is "#00".        %
%----------------------------------------------------------------------------%

let NEXTTIME_DEF =
 new_definition
  (`NEXTTIME`,
   "!f x1 x2.
     NEXTTIME (x1,x2) f =
      (x1 < x2) /\ (f x2) /\ (!x. (x1 < x) /\ (x < x2) ==> ~f x)");;

%----------------------------------------------------------------------------%
```

```
% A function "state_sig:num->`major" will represent a sequence of host       %
% machine states (the state at time t being "state_sig t") if:               %
%                                                                            %
%   (!t.                                                                     %
%      state_sig(t + 1) =                                                    %
%       NEXT(state_sig t,loadin_sig t,func_sig t))                           %
%                                                                            %
% We can thus formulate the termination of the host as:                      %
%                                                                            %
%   (!t.                                                                   ° %
%      state_sig(t + 1) =                                                    %
%       NEXT(state_sig t,loadin_sig t,func_sig t)) /\                        %
%   (NODE(state_sig t) = #00)                                                %
%   ==>                                                                      %
%   ?t'.NEXTTIME(t,t')(\x.NODE(state_sig x)=#00)                             %
%                                                                            %
% This is proved as the theorem HOST_TERMINATES below.                       %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% The host is correct if whenever a primary state occurs at time t and then  %
% the next time a primary state occurs is at time t', then the value of the  %
% COUNT component of the state at time t' equals the result of applying the  %
% behavioural specification function COUNTER to a triple consisting of:      %
%                                                                            %
%    1. the COUNT component of the state at time t,                          %
%    2. the value input at loadin at time t+1, and                           %
%    3. the value input at func at time t                                    %
%                                                                            %
% This is the partial correctness of the host. It is formalized as:          %
%                                                                            %
%    (!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\      %
%    (NODE(state_sig t)=#00)                                           /\     %
%    NEXTTIME (t,t') (\x. NODE(state_sig x)=#00)                             %
%    ==>                                                                     %
%    (COUNT(state_sig t') =                                                  %
%      COUNTER(COUNT(state_sig t),loadin_sig(t+1),func_sig t))               %
%                                                                            %
% This is proved as the theorem HOST_PCORRECT below.                         %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% Combining partial correctness and terminations we get the following        %
% statement of total correctness which is proved as HOST_CORRECT below.      %
%----------------------------------------------------------------------------%


%============================================================================%
% We now start the proof. First we retrieve some axioms from the theory      %
% `WORD_WIDTHS`. These will eventually be theorems from appropriate          %
% definitions.                                                               %
%============================================================================%

let WORD2_CASES_AX = axiom `WORD_WIDTHS` `WORD2_CASES_AX`
and VAL6_LESS_AX   = axiom `WORD_WIDTHS` `VAL6_LESS_AX`
and VAL6_WORD6_AX  = axiom `WORD_WIDTHS` `VAL6_WORD6_AX`;;

%----------------------------------------------------------------------------%
% We retrieve the behavioural definition of the counter from the theory      %
% `COUNTER`.                                                                 %
%----------------------------------------------------------------------------%

let COUNTER_DEF = definition `COUNTER` `COUNTER_DEF`;;

%----------------------------------------------------------------------------%
% We retrieve the definition of the host machine from the theory `HOST`.     %
%----------------------------------------------------------------------------%

let NEXT_DEF = definition `HOST` `NEXT_DEF`;;

let HOST_PRIMS =
 map
 (definition `HOST`)
 (words `ADD1_DEF FETCH_DEF LOAD_DEF INC1_DEF INC2_DEF`);;

%----------------------------------------------------------------------------%
% Th ML function EXECUTE_CONV is a conversion for generating theorems of the %
% form:                                                                      %
%                                                                            %
```

```
%    |- NEXT((count,double,w),loadin,func) = ...                              %
%                                                                             %
% where w is a particular 2-bit word.                                         %
%-----------------------------------------------------------------------------%

let EXECUTE_CONV =
 LIST_REWRITE_CONV([NEXT_DEF;LET_DEF]@HOST_PRIMS)
 THENC DEPTH_CONV BETA_CONV
 THENC VAL_CONV
 THENC WORD_CONV
 THENC BITS_CONV
 THENC EL_CONV
 THENC EQ_CONV
 THENC LIST_REWRITE_CONV[];;

%-----------------------------------------------------------------------------%
% We now apply EXECUTE_CONV to the four possible 2-bit words.                 %
%-----------------------------------------------------------------------------%

let NEXT_LEMMA00 = EXECUTE_CONV "NEXT((count,double,#00),loadin,func)"
and NEXT_LEMMA01 = EXECUTE_CONV "NEXT((count,double,#01),loadin,func)"
and NEXT_LEMMA10 = EXECUTE_CONV "NEXT((count,double,#10),loadin,func)"
and NEXT_LEMMA11 = EXECUTE_CONV "NEXT((count,double,#11),loadin,func)";;

%-----------------------------------------------------------------------------%
% The four lemmas produced by EXECUTE_CONV are:                               %
%                                                                             %
%   NEXT_LEMMA00 =                                                            %
%   |- NEXT((count,double,#00),loadin,func) =                                 %
%      ((VAL2 func = 0) =>                                                    %
%       (count,EL 0(BITS2 func),#00) |                                        %
%       ((VAL2 func = 1) =>                                                   %
%        (count,EL 0(BITS2 func),#11) |                                       %
%        (count,EL 0(BITS2 func),#01)))                                       %
%                                                                             %
%   NEXT_LEMMA01 =                                                            %
%   |- NEXT((count,double,#01),loadin,func) =                                 %
%      (double =>                                                             %
%       (((VAL6 count = 63) => #000000 | WORD6((VAL6 count) + 1)),            %
%        EL 0(BITS2 func),#10) |                                              %
%       (((VAL6 count = 63) => #000000 | WORD6((VAL6 count) + 1)),            %
%        EL 0(BITS2 func),#00))                                               %
%                                                                             %
%   NEXT_LEMMA10 =                                                            %
%   |- NEXT((count,double,#10),loadin,func) =                                 %
%      ((VAL6 count = 63) => #000000 | WORD6((VAL6 count) + 1)),              %
%      EL 0(BITS2 func),#00                                                   %
%                                                                             %
%   NEXT_LEMMA11 =                                                            %
%   |- NEXT((count,double,#11),loadin,func) = loadin,EL 0(BITS2 func),#00     %
%                                                                             %
%                                                                             %
% NEXT_LEMMA00 can be simplified by rewriting with                            %
% |- (p=>(x,y1)|(x,y2)) = (x,(p=>y1|y2)) and NEXT_LEMMA01 can be simplified  %
% by rewriting with |- (p=>x|x) = x. We prove these simplifications as part    %
% of the theorem COND_LEMMA.  This could be on utilities.ml.                  %
%-----------------------------------------------------------------------------%

let COND_LEMMA =
 TAC_PROOF
  (([], "!(p:bool) (x x1 x2:*) (y y1 y2:**).
           ((p=>x|x) = x)                                  /\
           ((p=>(x,y1)|(x,y2)) = (x, (p=>y1|y2))) /\
           ((p=>(x1,y)|(x2,y)) = ((p=>x1|x2), y))"),
   REPEAT GEN_TAC
    THEN BOOL_CASES_TAC "p:bool"
    THEN REWRITE_TAC[]);;

%-----------------------------------------------------------------------------%
% Using COND_LEMMA we simplify the four NEXT-lemmas and gather the resulting  %
% theorems into the list NEXT_LEMMAS.                                         %
%                                                                             %
%   NEXT_LEMMAS =                                                             %
%   [|- NEXT((count,double,#00),loadin,func) =                                %
%       count,EL 0(BITS2 func),                                               %
%       ((VAL2 func = 0) => #00 | ((VAL2 func = 1) => #11 | #01));            %
%    |- NEXT((count,double,#01),loadin,func) =                                %
%       ((VAL6 count = 63) => #000000 | WORD6((VAL6 count) + 1)),             %
%       EL 0(BITS2 func),(double => #10 | #00);                               %
```

```
%    |- NEXT((count,double,#10),loadin,func) =                                %
%       ((VAL6 count = 63) => #000000 | WORD6((VAL6 count) + 1)),             %
%       EL 0(BITS2 func),#00;                                                 %
%    |- NEXT((count,double,#11),loadin,func) = loadin,EL 0(BITS2 func),#00]  %
%                                                                            %
%----------------------------------------------------------------------------%

let NEXT_LEMMAS =
 map
  (REWRITE_RULE[COND_LEMMA])
  [NEXT_LEMMA00;NEXT_LEMMA01;NEXT_LEMMA10;NEXT_LEMMA11];;

%----------------------------------------------------------------------------%
% EXECUTE_NEXT is an ML function that takes a 2-bit word w and generates a   %
% theorem of the form:                                                       %
%                                                                            %
%   |- (!t.                                                                 %
%        state_sig(t + 1) =  NEXT(state_sig t,loadin_sig t,func_sig t)) ==>  %
%      (NODE(state_sig t) = w) ==>                                           %
%      (COUNT(state_sig(t + 1))  = ... )  /\                                 %
%      (DOUBLE(state_sig(t + 1)) = ... )  /\                                 %
%      (NODE(state_sig(t + 1))   = ... )                                     %
%                                                                            %
% and saves is as the theorem `NEXT_THMw`.                                   %
%----------------------------------------------------------------------------%

let EXECUTE_NEXT w =
 let th1 =
      ASSUME "(!t.
                state_sig(t+1) =
                 NEXT(state_sig t,loadin_sig t,func_sig t))"
 and th2 =
      ASSUME "NODE(state_sig t) = `w"
 in
 let th3 =
      SUBS[SPEC "(state_sig:num->`major)t" MAJOR_LEMMA](SPEC "t:num" th1)
 in
 let th4 = PURE_REWRITE_RULE(th2. NEXT_LEMMAS) th3
 in
 let th5 = SUBS[SPEC "(state_sig:num->`major)(t+1)" MAJOR_LEMMA]th4
 in
 let th6 = DISCH_ALL(PURE_REWRITE_RULE[PAIR_SPLIT]th5)
 in
 save_thm
  (`NEXT_THM``implode(tl(explode(fst(dest_const w)))), th6);;

%----------------------------------------------------------------------------%
% NEXT_THM00 is:                                                             %
%                                                                            %
%   |- (!t.state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==> %
%      (NODE(state_sig t) = #00) ==>                                         %
%      (COUNT(state_sig(t + 1)) = COUNT(state_sig t)) /\                     %
%      (DOUBLE(state_sig(t + 1)) = EL 0(BITS2(func_sig t))) /\               %
%      (NODE(state_sig(t + 1)) =                                             %
%       ((VAL2(func_sig t) = 0) =>                                           %
%        #00 |                                                               %
%        ((VAL2(func_sig t) = 1) => #11 | #01)))                             %
%                                                                            %
%----------------------------------------------------------------------------%

let NEXT_THM00 = EXECUTE_NEXT "#00";;

%----------------------------------------------------------------------------%
% NEXT_THM01 is:                                                             %
%                                                                            %
%   |- (!t.state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==> %
%      (NODE(state_sig t) = #01) ==>                                         %
%      (COUNT(state_sig(t + 1)) =                                            %
%       ((VAL6(COUNT(state_sig t)) = 63) =>                                  %
%        #000000 |                                                           %
%        WORD6((VAL6(COUNT(state_sig t))) + 1))) /\                          %
%      (DOUBLE(state_sig(t + 1)) = EL 0(BITS2(func_sig t))) /\               %
%      (NODE(state_sig(t + 1)) = (DOUBLE(state_sig t) => #10 | #00))         %
%                                                                            %
%----------------------------------------------------------------------------%

let NEXT_THM01 = EXECUTE_NEXT "#01";;

%----------------------------------------------------------------------------%
```

```
% NEXT_THM10 is:                                                              %
%                                                                             %
%   |- (!t.state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==> %
%      (NODE(state_sig t) = #10) ==>                                          %
%      (COUNT(state_sig(t + 1)) =                                             %
%       ((VAL6(COUNT(state_sig t)) = 63) =>                                   %
%        #000000 |                                                            %
%        WORD6((VAL6(COUNT(state_sig t))) + 1))) /\                           %
%      (DOUBLE(state_sig(t + 1)) = EL 0(BITS2(func_sig t))) /\                %
%      (NODE(state_sig(t + 1)) = #00)                                         %
%                                                                             %
%-----------------------------------------------------------------------------%

let NEXT_THM10 = EXECUTE_NEXT "#10";;

%-----------------------------------------------------------------------------%
% NEXT_THM11 is:                                                              %
%                                                                             %
%   |- (!t.state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==> %
%      (NODE(state_sig t) = #11) ==>                                          %
%      (COUNT(state_sig(t + 1)) = loadin_sig t) /\                            %
%      (DOUBLE(state_sig(t + 1)) = EL 0(BITS2(func_sig t))) /\                %
%      (NODE(state_sig(t + 1)) = #00)                                         %
%                                                                             %
%-----------------------------------------------------------------------------%

let NEXT_THM11 = EXECUTE_NEXT "#11";;

%-----------------------------------------------------------------------------%
% The four theorems just proved are gathered together into a list and bound   %
% to the ML name NEXT_THMS. This list is used in the rule NEXT_STEP           %
% described below.                                                            %
%-----------------------------------------------------------------------------%

let NEXT_THMS =
 [NEXT_THM00;NEXT_THM01;
  NEXT_THM10;NEXT_THM11];;

%-----------------------------------------------------------------------------%
% NEXT_INIT is an ML function that takes a 2-bit word w, assumes that         %
% "func_sig t = w" and then, using NEXT_THM00, generates a theorem of the     %
% form:                                                                       %
%                                                                             %
%  func_sig t = w,                                                            %
%  !t. state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t),           %
%  NODE(state_sig t) = #00                                                    %
%  |- (COUNT(state_sig(t + 1)) = ...) /\                                      %
%     (DOUBLE(state_sig(t + 1)) = ...) /\                                     %
%     (NODE(state_sig(t + 1)) = ...)                                          %
%                                                                             %
%-----------------------------------------------------------------------------%

let NEXT_INIT w =
 let stepthm =
  ASSUME "(!t. state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t))"
 and nodethm = ASSUME "NODE(state_sig(t:num)) = #00"
 and functhm = ASSUME "(func_sig:num->word2)t = w"
 in
 let th1 = NEXT_THM00
 in
 let th2 = MP th1 stepthm
 in
 let th3 = MP th2 nodethm
 in
 let th4 = REWRITE_RULE[functhm]th3
 in
 CONV_RULE
  (VAL_CONV THENC BITS_CONV THENC EL_CONV THENC EQ_CONV THENC COND_CONV)
  th4;;

%-----------------------------------------------------------------------------%
% Let "t_m" mean "(((t+1)+1)...)+1" where there are m 1s. Then the ML         %
% function NEXT_STEP takes as argument a theorem of the form:                 %
%                                                                             %
%    |- (COUNT(state_sig(t_m))  = ...) /\                                     %
%       (DOUBLE(state_sig(t_m)) = ...) /\                                     %
%       (NODE(state_sig(t_m))   = ...)                                        %
%                                                                             %
% and from this uses NEXT_THMS to generate a theorem of the form:             %
```

```
%                                                                            %
%    |- (COUNT(state_sig(t_(m+1)))  = ...) /\                                %
%       (DOUBLE(state_sig(t_(m+1))) = ...) /\                                %
%       (NODE(state_sig(t_(m+1)))   = ...)                                   %
%                                                                            %
% The function NEXT_STEP enables us to derive theorems corresponding to each %
% state transition in Fig. 3. The state changes along the spanning tree in   %
% Fig 7. are then derived by iterating NEXT_STEP. This iteration is done     %
% with the ML function NEXT_RUN described below.                             %
%----------------------------------------------------------------------------%

let NEXT_STEP th =
 let [Cthm;Dthm;Nthm] = CONJUNCTS th
 and nextthm =
      ASSUME
       "(!t. state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t))"
 in
 let nval = rhs(concl Nthm)
 and t    = rand(rand(lhs(concl Nthm)))
 in
 let stepthm =
  INST[t,"t:num"]
  (if nval = "#00" then NEXT_THM00
   if nval = "#10" then NEXT_THM10
   if nval = "#01" then NEXT_THM01
   if nval = "#11" then NEXT_THM11
                   else fail)
 in
 let th1 = REWRITE_RULE[th](MP(MP stepthm nextthm)Nthm)
 in
 CONV_RULE
  (VAL_CONV THENC BITS_CONV THENC EL_CONV THENC EQ_CONV THENC COND_CONV)
  th1;;

%----------------------------------------------------------------------------%
% The function NEXT_RUN takes a 2-bit word w, assumes that "func_sig t = w"  %
% and then (using NEXT_INIT and NEXT_STEP) generates a list of theorems      %
% describing the states at the nodes in the spanning tree of the host        %
% machine. NEXT_RUN is recursive and terminates when a state whose NODE      %
% component is #00 is reached.                                               %
%----------------------------------------------------------------------------%

let NEXT_RUN w =
 letrec fn th =
  let nval = rhs(concl(hd(tl(tl(CONJUNCTS th)))))
  in
  (message(fst(dest_const nval));
  (if nval="#00"
    then [th]
   if (nval="#01" or nval="#10" or nval="#11")
    then th.(fn(NEXT_STEP th))
    else fail))
 in
 fn(NEXT_INIT w);;

%----------------------------------------------------------------------------%
% The ML lists PATHA_THMS, PATHB_THMS, PATHC_THMS and PATHD_THMS describe    %
% the behaviour of the host machine when it is executing paths A, B, C and D %
% of the spanning tree (Fig. 7). After deriving these lists of theorems we   %
% conjoin the elements of the lists to get single theorems for each path.    %
% These are saved as PATHA_THM, PATHB_THM, PATHC_THM and PATHD_THM.          %
%----------------------------------------------------------------------------%

let PATHA_THMS = NEXT_RUN "#00";;

let PATHB_THMS = NEXT_RUN "#01";;

let PATHC_THMS = NEXT_RUN "#10";;

let PATHD_THMS = NEXT_RUN "#11";;

%----------------------------------------------------------------------------%
% PATHA_THM is:                                                              %
%                                                                            %
%   |- (NODE(state_sig t) = #00) ==>                                         %
%      (!t.state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==> %
%      (func_sig t = #00) ==>                                                %
%      (COUNT(state_sig(t + 1)) = COUNT(state_sig t)) /\                     %
%      (DOUBLE(state_sig(t + 1)) = F) /\                                     %
```

```
%      (NODE(state_sig(t + 1)) = #00)                                          %
%                                                                              %
%-------------------------------------------------------------------------------%

let PATHA_THM =
 save_thm(`PATHA_THM`, DISCH_ALL(LIST_CONJ PATHA_THMS));;

%-------------------------------------------------------------------------------%
% PATHB_THM is:                                                                %
%                                                                              %
%   |- (NODE(state_sig t) = #00) ==>                                           %
%      (!t.state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==> %
%      (func_sig t = #01) ==>                                                  %
%      ((COUNT(state_sig(t + 1)) = COUNT(state_sig t)) /\                      %
%       (DOUBLE(state_sig(t + 1)) = T) /\                                      %
%       (NODE(state_sig(t + 1)) = #11)) /\                                     %
%      (COUNT(state_sig((t + 1) + 1)) = loadin_sig(t + 1)) /\                  %
%      (DOUBLE(state_sig((t + 1) + 1)) = EL 0(BITS2(func_sig(t + 1)))) /\      %
%      (NODE(state_sig((t + 1) + 1)) = #00)                                    %
%                                                                              %
%-------------------------------------------------------------------------------%

let PATHB_THM =
 save_thm(`PATHB_THM`, DISCH_ALL(LIST_CONJ PATHB_THMS));;

%-------------------------------------------------------------------------------%
% PATHC_THM is:                                                                %
%                                                                              %
%   |- (NODE(state_sig t) = #00) ==>                                           %
%      (!t.state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==> %
%      (func_sig t = #10) ==>                                                  %
%      ((COUNT(state_sig(t + 1)) = COUNT(state_sig t)) /\                      %
%       (DOUBLE(state_sig(t + 1)) = F) /\                                      %
%       (NODE(state_sig(t + 1)) = #01)) /\                                     %
%      (COUNT(state_sig((t + 1) + 1)) =                                        %
%       ((VAL6(COUNT(state_sig t)) = 63) =>                                    %
%        #000000 |                                                             %
%        WORD6((VAL6(COUNT(state_sig t))) + 1))) /\                            %
%      (DOUBLE(state_sig((t + 1) + 1)) = EL 0(BITS2(func_sig(t + 1)))) /\      %
%      (NODE(state_sig((t + 1) + 1)) = #00)                                    %
%                                                                              %
%-------------------------------------------------------------------------------%

let PATHC_THM =
 save_thm(`PATHC_THM`, DISCH_ALL(LIST_CONJ PATHC_THMS));;

%-------------------------------------------------------------------------------%
% PATHD_THM is:                                                                %
%                                                                              %
%   |- (NODE(state_sig t) = #00) ==>                                           %
%      (!t.state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) ==> %
%      (func_sig t = #11) ==>                                                  %
%      ((COUNT(state_sig(t + 1)) = COUNT(state_sig t)) /\                      %
%       (DOUBLE(state_sig(t + 1)) = T) /\                                      %
%       (NODE(state_sig(t + 1)) = #01)) /\                                     %
%      ((COUNT(state_sig((t + 1) + 1)) =                                       %
%        ((VAL6(COUNT(state_sig t)) = 63) =>                                   %
%         #000000 |                                                            %
%         WORD6((VAL6(COUNT(state_sig t))) + 1))) /\                           %
%       (DOUBLE(state_sig((t + 1) + 1)) = EL 0(BITS2(func_sig(t + 1)))) /\     %
%       (NODE(state_sig((t + 1) + 1)) = #10)) /\                               %
%      (COUNT(state_sig(((t + 1) + 1) + 1)) =                                  %
%       ((VAL6                                                                 %
%         ((VAL6(COUNT(state_sig t)) = 63) =>                                  %
%          #000000 |                                                           %
%          WORD6((VAL6(COUNT(state_sig t))) + 1)) =                            %
%         63) =>                                                               %
%        #000000 |                                                             %
%        WORD6                                                                 %
%        ((VAL6                                                                %
%          ((VAL6(COUNT(state_sig t)) = 63) =>                                 %
%           #000000 |                                                          %
%           WORD6((VAL6(COUNT(state_sig t))) + 1))) +                          %
%         1))) /\                                                              %
%      (DOUBLE(state_sig(((t + 1) + 1) + 1)) =                                 %
%       EL 0(BITS2(func_sig((t + 1) + 1)))) /\                                 %
%      (NODE(state_sig(((t + 1) + 1) + 1)) = #00)                              %
%                                                                              %
%-------------------------------------------------------------------------------%
```

```
let PATHD_THM =
 save_thm(`PATHD_THM`, DISCH_ALL(LIST_CONJ PATHD_THMS));;

%----------------------------------------------------------------------------%
% In order to prove our main goals we need some lemmas about numbers. These  %
% are:                                                                       %
%                                                                            %
%  INTERVAL_LEMMA1                                                           %
%    |- !t x. t < x /\ x < t = F                                             %
%                                                                            %
%  INTERVAL_LEMMA2                                                           %
%    |- !t x. t < x /\ x < (t + 1) = F                                       %
%                                                                            %
%  INTERVAL_LEMMA3                                                           %
%    |- !t x. t < x /\ x < ((t + 1) + 1) = (x = t + 1)                       %
%                                                                            %
%  INTERVAL_LEMMA4                                                           %
%    |- !t1 t2 t.                                                           %
%        t1 < t /\ t < t2 =                                                  %
%        (t = t1 + 1) /\ (t1 + 1) < t2 \/ (t1 + 1) < t /\ t < t2             %
%                                                                            %
%  INTERVAL_LEMMA5                                                           %
%    |- !t1 t2 t.                                                           %
%        (t1 + 1) < t2 ==>                                                   %
%        (t1 < t /\ t < t2 = (t = t1 + 1) \/ (t1 + 1) < t /\ t < t2)         %
%                                                                            %
%  LESS_ADD1_REFL                                                            %
%    |- !n. n < (n + 1)                                                     %
%                                                                            %
%----------------------------------------------------------------------------%

let INTERVAL_LEMMA1 =
 prove_thm
  (`INTERVAL_LEMMA1`,
   "!t x.
     (t < x /\ x < t) = F",
   REWRITE_TAC[LESS_ANTISYM]);;

let INTERVAL_LEMMA2 =
 prove_thm
  (`INTERVAL_LEMMA2`,
   "!t x.
     (t < x /\ x < t+1) = F",
   REPEAT GEN_TAC
    THEN REWRITE_TAC[SYM(SPEC_ALL ADD1);LESS_THM]
    THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]
    THEN IMP_RES_TAC
          (DISCH_ALL(MP(SPEC_ALL NOT_LESS_EQ)(SYM(ASSUME "n:num=m"))))
    THEN IMP_RES_TAC LESS_ANTISYM);;

let INTERVAL_LEMMA3 =
 prove_thm
  (`INTERVAL_LEMMA3`,
   "!t x.
     (t < x /\ x < (t+1)+1) = (x = t+1)",
   REPEAT GEN_TAC
    THEN REWRITE_TAC[num_CONV "1";ADD_CLAUSES]
    THEN EQ_TAC
    THEN ASM_REWRITE_TAC[LESS_THM]
    THEN REPEAT STRIP_TAC
    THEN IMP_RES_TAC
          (DISCH_ALL
           (MP(SPECL["t:num";"x:num"] NOT_LESS_EQ)(SYM(ASSUME "x:num=t"))))
    THEN IMP_RES_TAC LESS_ANTISYM
    THEN ASM_REWRITE_TAC[LESS_THM;LESS_SUC_REFL]);;

let INTERVAL_LEMMA4 =
 prove_thm
  (`INTERVAL_LEMMA4`,
   "!t1 t2 t.
     (t1 < t /\ t < t2) = ((t = t1+1) /\ ((t1+1) < t2)) \/
                         (((t1+1) < t) /\ (t < t2))",
   REPEAT GEN_TAC
    THEN REWRITE_TAC[SYM(SPEC_ALL ADD1)]
    THEN EQ_TAC
    THEN REPEAT STRIP_TAC
    THEN IMP_RES_TAC SUC_LESS
```

```
    THEN ASM_REWRITE_TAC[LESS_SUC_REFL]
    THEN ASM_CASES_TAC "t = SUC t1"
    THEN ASM_REWRITE_TAC[LESS_REFL]
    THENL
     [REWRITE_TAC[SYM(ASSUME "t = SUC t1")]
       THEN ASM_REWRITE_TAC[];
      ASSUME_TAC(NOT_EQ_SYM(ASSUME"~(t = SUC t1)"))
       THEN IMP_RES_TAC LESS_SUC_EQ_COR]);;

let INTERVAL_LEMMA5 =
 prove_thm
  (`INTERVAL_LEMMA5`,
   "!t1 t2 t.
     ((t1+1) < t2)
     ==>
     ((t1 < t /\ t < t2) = (t = t1+1) \/ ((t1+1) < t /\ t < t2))",
   REPEAT GEN_TAC
    THEN DISCH_TAC
    THEN SUBST_TAC[SPEC_ALL INTERVAL_LEMMA4]
    THEN EQ_TAC
    THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let LESS_ADD1_REFL =
 prove_thm
  (`LESS_ADD1_REFL`,
   "!n. n < (n+1)",
   REWRITE_TAC[SYM(SPEC_ALL ADD1);LESS_SUC_REFL]);;

%----------------------------------------------------------------------------%
% The ML function LESS_ADD_RULE : term -> num -> thm takes a term t and      %
% as arguments and then generates the theorem:                               %
%                                                                            %
%    |- t < ((...((t+1)+1)...)+1)                                           %
%                                                                            %
%----------------------------------------------------------------------------%

letrec LESS_ADD1_RULE t n =
 if n=0
  then SPEC t LESS_ADD1_REFL
  else
  (let th1 = SPEC t LESS_ADD1_REFL
   and th2 = LESS_ADD1_RULE "~t+1" (n-1)
   in
   MATCH_MP LESS_TRANS (CONJ th1 th2));;

%----------------------------------------------------------------------------%
% Let "t_m" mean "(((t+1)+1)...)+1" where there are m 1s.  Then the ML       %
% function INTERVAL_RULE takes as argument a pair of terms (t,x) and a       %
% number n and from them generates theorems as shown below:                  %
%                                                                            %
%                                                                            %
%   ("t","x") 0  --> |- (t<x) /\ (x<t)   = F                                 %
%   ("t","x") 1  --> |- (t<x) /\ (x<t_1) = F                                 %
%   ("t","x") 2  --> |- (t<x) /\ (x<t_2) = (x=t_1)                           %
%   ("t","x") 3  --> |- (t<x) /\ (x<t_3) = (x=t_1) \/ (x=t_2)                %
%           .                                                                %
%           .                                                                %
%           .                                                                %
%                                                                            %
%   ("t","x") n  --> |- (t<x) /\ (x<t_n) = (x=t_1) \/ ... \/ (x=t_(n-1))     %
%                                                                            %
%----------------------------------------------------------------------------%

letrec INTERVAL_RULE (t,x) n =
 if n=0
  then SPECL[t;x]INTERVAL_LEMMA1
 if n=1
  then SPECL[t;x]INTERVAL_LEMMA2
 if n=2
  then SPECL[t;x]INTERVAL_LEMMA3
  else
  (let th1 = INTERVAL_RULE("~t+1",x) (n-1)
   and th2 = LESS_ADD1_RULE "~t+1" (n-2)
   in
   let tn = hd(tl(snd(strip_comb(concl th2))))
   in
   let th3 = MP(SPECL[t;tn;x]INTERVAL_LEMMA5)th2
   in
```

```
   SUBS[th1]th3);;

%----------------------------------------------------------------------------%
% The four theorems that follow give exact values for t' where if the NODE   %
% value of the state at t is #00 then t' is the next time after t that the   %
% NODE value of the state is #00.                                            %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% PATHA_NEXT_THM gives the time taken to go down path A in Fig. 7.           %
%----------------------------------------------------------------------------%

let PATHA_NEXT_THM =
 prove_thm
  (`PATHA_NEXT_THM`,
   "(!t. state_sig(t + 1) =  NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t) = #00)                                            /\
    (func_sig t = #00)
    ==>
    NEXTTIME(t,t+1)(\x.NODE(state_sig x)=#00)",
   REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC
          [NEXTTIME_DEF;
           LESS_ADD1_RULE "t:num" 0;
           INTERVAL_RULE("t:num","x:num")1]
    THEN CONV_TAC(DEPTH_CONV BETA_CONV)
    THEN IMP_RES_TAC PATHA_THM);;

%----------------------------------------------------------------------------%
% PATHB_NEXT_THM gives the time taken to go down path B in Fig. 7.           %
%----------------------------------------------------------------------------%

let PATHB_NEXT_THM =
 prove_thm
  (`PATHB_NEXT_THM`,
   "(!t. state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t) = #00)                                          /\
    (func_sig t = #01)
    ==>
    NEXTTIME(t,(t+1)+1)(\x.NODE(state_sig x)=#00)",
   REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC
          [NEXTTIME_DEF;
           LESS_ADD1_RULE "t:num" 1;
           INTERVAL_RULE("t:num","x:num")2]
    THEN CONV_TAC(DEPTH_CONV BETA_CONV)
    THEN REWRITE_TAC[UNDISCH_ALL PATHB_THM]
    THEN GEN_TAC
    THEN DISCH_TAC
    THEN FILTER_ASM_REWRITE_TAC
          (\t.(lhs t="x:num")?false)
          [UNDISCH_ALL PATHB_THM]
    THEN CONV_TAC EQ_CONV
    THEN REWRITE_TAC[]);;

%----------------------------------------------------------------------------%
% PATHC_NEXT_THM gives the time taken to go down path C in Fig. 7.           %
%----------------------------------------------------------------------------%

let PATHC_NEXT_THM =
 prove_thm
  (`PATHC_NEXT_THM`,
   "(!t. state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t) = #00)                                          /\
    (func_sig t = #10)
    ==>
    NEXTTIME(t,(t+1)+1)(\x.NODE(state_sig x)=#00)",
   REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC
          [NEXTTIME_DEF;
           LESS_ADD1_RULE "t:num" 1;
           INTERVAL_RULE("t:num","x:num")2]
    THEN CONV_TAC(DEPTH_CONV BETA_CONV)
    THEN REWRITE_TAC[UNDISCH_ALL PATHC_THM]
    THEN GEN_TAC
    THEN DISCH_TAC
    THEN FILTER_ASM_REWRITE_TAC
          (\t.(lhs t="x:num")?false)
          [UNDISCH_ALL PATHC_THM]
```

```
    THEN CONV_TAC EQ_CONV
    THEN REWRITE_TAC[]);;

%----------------------------------------------------------------------------%
% PATHD_NEXT_THM gives the time taken to go down path D in Fig. 7.           %
%----------------------------------------------------------------------------%

let PATHD_NEXT_THM =
 prove_thm
  (`PATHD_NEXT_THM`,
   "(!t. state_sig(t + 1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t) = #00)                                          /\
    (func_sig t = #11)
    ==>
    NEXTTIME(t,((t+1)+1)+1)(\x.NODE(state_sig x)=#00)",
   REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC
          [NEXTTIME_DEF;
           LESS_ADD1_RULE "t:num" 2;
           INTERVAL_RULE("t:num","x:num")3]
    THEN CONV_TAC(DEPTH_CONV BETA_CONV)
    THEN REWRITE_TAC[UNDISCH_ALL PATHD_THM]
    THEN STRIP_TAC
    THEN STRIP_TAC
    THEN FILTER_ASM_REWRITE_TAC
          (\t.(lhs t="x:num")?false)
          [UNDISCH_ALL PATHD_THM]
    THEN CONV_TAC EQ_CONV
    THEN REWRITE_TAC[]);;

%----------------------------------------------------------------------------%
% We can now prove the termination of the host machine.                      %
%----------------------------------------------------------------------------%

let HOST_TERMINATES =
 prove_thm
  (`HOST_TERMINATES`,
   "(!t.
       state_sig(t + 1) =
       NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t) = #00)
    ==>
    ?t'.NEXTTIME(t,t')(\x.NODE(state_sig x)=#00)",
  GDISJ_CASES_TAC(SPEC "(func_sig:num->word2)t" WORD2_CASES_AX)
   THEN STRIP_TAC
   THENL
    [EXISTS_TAC "t+1"
      THEN IMP_RES_TAC PATHA_NEXT_THM
      THEN ASM_REWRITE_TAC[];
     EXISTS_TAC "(t+1)+1"
      THEN IMP_RES_TAC PATHB_NEXT_THM
      THEN ASM_REWRITE_TAC[];
     EXISTS_TAC "(t+1)+1"
      THEN IMP_RES_TAC PATHC_NEXT_THM
      THEN ASM_REWRITE_TAC[];
     EXISTS_TAC "((t+1)+1)+1"
      THEN IMP_RES_TAC PATHD_NEXT_THM
      THEN ASM_REWRITE_TAC[]]);;

%----------------------------------------------------------------------------%
% NEXTTIME_LEMMA below shows that there is a unique next time that f is true %
% after time t.                                                              %
%----------------------------------------------------------------------------%

let NEXTTIME_LEMMA =
 prove_thm
  (`NEXTTIME_LEMMA`,
   "!f x1 x2.
     NEXTTIME (x1,x2) f /\ NEXTTIME (x1,x3) f ==> (x2 = x3)",
   REPEAT GEN_TAC
    THEN REWRITE_TAC[NEXTTIME_DEF]
    THEN STRIP_TAC
    THEN ASM_CASES_TAC "x2:num = x3"
    THEN ASM_CASES_TAC "x2 < x3"
    THEN ASM_REWRITE_TAC[]
    THEN IMP_RES_TAC LESS_CASES_IMP
    THEN RES_TAC);;

%----------------------------------------------------------------------------%
```

```
% HOST_NEXT_THM combines the various lemmas along the paths of the spanning  %
% of the host (Fig. 7).                                                       %
%-----------------------------------------------------------------------------%

let HOST_NEXT_THM =
 prove_thm
  (`HOST_NEXT_THM`,
   "(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t)=#00)                                          /\
    NEXTTIME (t,t') (\x. NODE(state_sig x)=#00)
    ==>
    (t' = ((func_sig t = #00) => t+1         |
           (func_sig t = #01) => (t+1)+1     |
           (func_sig t = #10) => (t+1)+1     |
           (func_sig t = #11) => ((t+1)+1)+1 |
                                 ARB))",
   GDISJ_CASES_TAC(SPEC "(func_sig:num->word2)t" WORD2_CASES_AX)
    THEN CONV_TAC EQ_CONV
    THEN REWRITE_TAC[]
    THEN REPEAT STRIP_TAC
    THENL
     [IMP_RES_TAC PATHA_NEXT_THM;
      IMP_RES_TAC PATHB_NEXT_THM;
      IMP_RES_TAC PATHC_NEXT_THM;
      IMP_RES_TAC PATHD_NEXT_THM]
    THEN IMP_RES_TAC NEXTTIME_LEMMA);;

%-----------------------------------------------------------------------------%
% COUNTER_DEF_SIMP is an unwound version of the top level specification of    %
% counter. It is:                                                             %
%                                                                             %
%  |- COUNTER(count,loadin,func) =                                           %
%     ((VAL2 func = 0) =>                                                     %
%      count |                                                                %
%      ((VAL2 func = 1) =>                                                    %
%       loadin |                                                              %
%       ((VAL2 func = 2) =>                                                   %
%        ((VAL6 count = 63) => #000000 | WORD6((VAL6 count) + 1)) |           %
%        ((VAL2 func = 3) =>                                                  %
%         ((VAL6 count = 63) =>                                               %
%          #000001 |                                                          %
%          ((VAL6 count = 62) => #000000 | WORD6((VAL6 count) + 2))) |        %
%         ARB))))                                                             %
%                                                                             %
%-----------------------------------------------------------------------------%

let COUNTER_DEF_SIMP =
 CONV_RULE
  (LIST_REWRITE_CONV[LET_DEF]
    THENC DEPTH_CONV BETA_CONV
    THENC WORD_CONV)
  COUNTER_DEF;;

%-----------------------------------------------------------------------------%
% We now prove some annoying trivial lemmas for use later.                    %
%                                                                             %
% LESS64_LEMMA     |- n < 64 ==> ~(n = 63) ==> ~(n = 62) ==> n < 62          %
% LESS62_63_LEMMA  |- n < 62 ==> n < 63                                      %
% SUC63_64_LEMMA   |- n < 63 ==> (SUC n) < 64                               %
%-----------------------------------------------------------------------------%

let LESS64_LEMMA =
 prove_thm
  (`LESS64_LEMMA`,
   "(n<64) ==> ~(n=63) ==> ~(n=62) ==> (n<62)",
   REWRITE_TAC[num_CONV "64";num_CONV "63";LESS_THM]
    THEN STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let LESS62_63_LEMMA =
 prove_thm
  (`LESS62_63_LEMMA`,
   "(n<62) ==> (n<63)",
   REWRITE_TAC[num_CONV "63";LESS_THM]
    THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let SUC63_64_LEMMA =
 prove_thm
```

```
  (`SUC63_64_LEMMA`,
   "(n<63) ==> (SUC n < 64)",
   DISCH_TAC
    THEN IMP_RES_TAC LESS_MONO
    THEN ASM_REWRITE_TAC[num_CONV "64"]);;

%----------------------------------------------------------------------------%
% The grossly ad-hoc lemma below solves the difficult case of the theorem    %
% HOST_PCORRECT. It says that incrementing twice is equivalent to adding     %
% two. The messiness is due to the need to handle the cases when the value   %
% of the COUNT component of the state is 62 or 63.                           %
%----------------------------------------------------------------------------%

let ADD2_LEMMA =
 prove_thm
  (`ADD2_LEMMA`,
   "((VAL6
      ((VAL6(COUNT(state_sig t)) = 63) =>
       #000000 |
       WORD6((VAL6(COUNT(state_sig t))) + 1)) =
      63) =>
     #000000 |
     WORD6
     ((VAL6
       ((VAL6(COUNT(state_sig t)) = 63) =>
        #000000 |
        WORD6((VAL6(COUNT(state_sig t))) + 1))) +
      1)) =
    ((VAL6(COUNT(state_sig t)) = 63) =>
     #000001 |
     ((VAL6(COUNT(state_sig t)) = 62) =>
      #000000 |
      WORD6((VAL6(COUNT(state_sig t))) + 2)))",
   ASM_CASES_TAC "(VAL6(COUNT(state_sig t)) = 63)"
    THEN ASM_CASES_TAC "(VAL6(COUNT(state_sig t)) = 62)"
    THEN ASM_REWRITE_TAC[]
    THEN REPEAT
          (CHANGED_TAC
           (CONV_TAC
            (EQ_CONV
              THENC ADD_CONV
              THENC (WORD_CONV ORELSEC ALL_CONV)
              THENC VAL_CONV
              THENC COND_CONV)))
    THEN ASSUME_TAC(SPEC "COUNT(state_sig t)" VAL6_LESS_AX)
    THEN PURE_REWRITE_TAC
          [num_CONV "2";num_CONV "1";ADD_CLAUSES;
           num_CONV "63";SYM(SPEC_ALL ADD1);INV_SUC_EQ]
    THEN IMP_RES_TAC LESS64_LEMMA
    THEN IMP_RES_TAC LESS62_63_LEMMA
    THEN IMP_RES_TAC SUC63_64_LEMMA
    THEN IMP_RES_TAC VAL6_WORD6_AX
    THEN ASM_REWRITE_TAC[INV_SUC_EQ]);;

%----------------------------------------------------------------------------%
% We can now prove the partial correctness of the host machine.              %
%----------------------------------------------------------------------------%

let HOST_PCORRECT =
 prove_thm
  (`HOST_PCORRECT`,
   "(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t)=#00)                                          /\
    NEXTTIME (t,t') (\x. NODE(state_sig x)=#00)
    ==>
    (COUNT(state_sig t') =
      COUNTER(COUNT(state_sig t),loadin_sig(t+1),func_sig t))",
   REWRITE_TAC[COUNTER_DEF_SIMP]
    THEN GDISJ_CASES_TAC(SPEC "(func_sig:num->word2)t" WORD2_CASES_AX)
    THEN CONV_TAC(VAL_CONV THENC EQ_CONV)
    THEN REWRITE_TAC[]
    THEN REPEAT STRIP_TAC
    THEN IMP_RES_TAC HOST_NEXT_THM
    THEN FILTER_ASM_REWRITE_TAC
          (\t.((lhs t="t':num")or(lhs t="(func_sig:num->word2)t"))?false)
          []
    THEN CONV_TAC EQ_CONV
    THEN REWRITE_TAC[]
    THENL
```

```
     [REWRITE_TAC[hd(rev PATHA_THMS)];
      REWRITE_TAC[hd(rev PATHB_THMS)];
      REWRITE_TAC[hd(rev PATHC_THMS)];
      REWRITE_TAC[hd(rev PATHD_THMS)]]
    THEN REWRITE_TAC[ADD2_LEMMA]);;

%----------------------------------------------------------------------------%
% Combining HOST_TERMINATES and HOST_PCORRECT we can prove the theorem       %
% HOST_CORRECT below which expresses the total correctness of the host.      %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% The derived inference rule EXISTS_AND_RULE is useful for combining         %
% termination and partial correctness to get total correctness.              %
% This could be on utilities.ml.                                             %
%                                                                            %
% EXISTS_AND_RULE : (thm # thm) -> thm                                       %
%                                                                            %
%            A1 |- ?x. tm[x]        A2 |- !x. tm[x] ==> tm'[x]              %
%            ---------------------------------------------                   %
%                     A1 u A2 |- ?x. tm[x] /\ tm'[x]                         %
%                                                                            %
%                                                                            %
%----------------------------------------------------------------------------%

let EXISTS_AND_RULE(th1,th2) =
 let x,t1 = dest_exists(concl th1)
 and t2   = snd(dest_imp(snd(dest_forall(concl th2))))
 in
 let etm = mk_select(x,t1)
 in
 let th3 = SELECT_RULE th1
 in
 let th4 = CONJ th3 (MP(SPEC etm th2)th3)
 in
 EXISTS ("?`x.`t1/\`t2",etm) th4;;

%----------------------------------------------------------------------------%
% By forward proof we can now deduce:                                        %
%                                                                            %
%   HOST_CORRECT_LEMMA =                                                     %
%   ... |- ?t'.                                                             %
%           NEXTTIME(t,t')(\x. NODE(state_sig x) = #00) /\                   %
%           (COUNT(state_sig t') =                                           %
%            COUNTER(COUNT(state_sig t),loadin_sig(t + 1),func_sig t))       %
%                                                                            %
%----------------------------------------------------------------------------%

let HOST_CORRECT_LEMMA =
 EXISTS_AND_RULE
  (UNDISCH_ALL(hd(IMP_CANON HOST_TERMINATES)),
   GEN_ALL(UNDISCH(UNDISCH(hd(IMP_CANON HOST_PCORRECT)))));;


let HOST_CORRECT =
 prove_thm
  (`HOST_CORRECT`,
   "(!t. state_sig(t+1) = NEXT(state_sig t,loadin_sig t,func_sig t)) /\
    (NODE(state_sig t)=#00)
    ==>
    ?t'. NEXTTIME (t,t') (\x. NODE(state_sig x)=#00) /\
         (COUNT(state_sig t') =
           COUNTER(COUNT(state_sig t),loadin_sig(t+1),func_sig t))",
   REPEAT STRIP_TAC
    THEN IMP_RES_TAC(DISCH_ALL HOST_CORRECT_LEMMA));;

%----------------------------------------------------------------------------%
% This completes the top-level verification of the counter so we can close   %
% the theory `HOST_CORRECT`.                                                 %
%----------------------------------------------------------------------------%

close_theory();;
```

```
%============================================================================%
% The definition, specification and implementation of COUNTLOGIC, and the    %
% relations between them. This represents the combinatorial behaviour of the %
% whole circuit.                                                             %
%                                                                            %
% The theorems used are:                                                     %
%    INCCON_DEF_SPEC_EQUIV from INCCON_CORRECT.th                            %
%    INCCON_SPEC_IMP_EQUIV from INCCON_CORRCET.th, and likewise for MPLXCON, %
% MULTIPLEX, INCLOGIC, and NEXTNODE.                                         %
%                                                                            %
% The main result is:                                                        %
%    COUNTLOGIC_DEF_IMP_EQUIV                                                %
%============================================================================%

%----------------------------------------------------------------------------%
% Notice that there is no "box" for the computation of DOUBLE in Fig. 5.     %
% It would have been cleaner to have had one, and to have defined its simple %
% behaviour by a function DOUBLE_SPEC, like the boxes MULTIPLEX and so on.   %
% Instead, the specification is just an equation "double' = f0", where       %
% double' is the output variable and f0 is the l.s.b. of func.               %
%----------------------------------------------------------------------------%

new_theory `CIRCUIT_CORRECT`;;

map new_parent [`INCLOGIC_CORRECT`;`MULTIPLEX_CORRECT`;`INCCON_CORRECT`;
                `MPLXCON_CORRECT`;`NEXTNODE_CORRECT`];;

let [V2_DEF;V6_DEF] = map (definition `WORD_WIDTHS`)[`V2_DEF`;`V6_DEF`];;

let BITS2_AX = theorem `WORD_WIDTHS` `BITS2_AX`;;

let BITS6_AX = axiom   `WORD_WIDTHS` `BITS6_AX`;;

let EL_BITS2_AX = theorem `WORD_WIDTHS` `EL_BITS2_AX`;;

let BITS6_WORD6_AX2 = theorem `WORD_WIDTHS` `BITS6_WORD6_AX2`;;

let V6_THM1 = theorem `WORD_WIDTHS` `V6_THM1`;;

loadt `~/counter/utilities`;;

let major = ":word6#bool#word2";;

let [INCCON_DEF;MPLXCON_DEF;MULTIPLEX_DEF;INCLOGIC_DEF;NEXTNODE_DEF] =
    map(definition `DESIGN`) [`INCCON_DEF`;`MPLXCON_DEF`;`MULTIPLEX_DEF`;
                              `INCLOGIC_DEF`;`NEXTNODE_DEF`];;

let [INCCON_SPEC_DEF;MPLXCON_SPEC_DEF;MULTIPLEX_SPEC_DEF;INCLOGIC_SPEC_DEF;
     NEXTNODE_SPEC_DEF] =
    [definition `INCCON_CORRECT`     `INCCON_SPEC_DEF`;
     definition `MPLXCON_CORRECT`    `MPLXCON_SPEC_DEF`;
     definition `MULTIPLEX_CORRECT`  `MULTIPLEX_SPEC_DEF`;
     definition `INCLOGIC_CORRECT`   `INCLOGIC_SPEC_DEF`;
     definition `NEXTNODE_CORRECT`   `NEXTNODE_SPEC_DEF`];;

let [INCCON_DEF_SPEC_EQUIV;
     MPLXCON_DEF_SPEC_EQUIV;
     MULTIPLEX_DEF_SPEC_EQUIV;
     INCLOGIC_DEF_SPEC_EQUIV;
     NEXTNODE_DEF_SPEC_EQUIV] =
    [theorem `INCCON_CORRECT`     `INCCON_DEF_SPEC_EQUIV`;
     theorem `MPLXCON_CORRECT`    `MPLXCON_DEF_SPEC_EQUIV`;
     theorem `MULTIPLEX_CORRECT`  `MULTIPLEX_DEF_SPEC_EQUIV`;
     theorem `INCLOGIC_CORRECT`   `INCLOGIC_DEF_SPEC_EQUIV`;
     theorem `NEXTNODE_CORRECT`   `NEXTNODE_DEF_SPEC_EQUIV`];;


let [INCCON_SPEC_IMP_EQUIV;
     MPLXCON_SPEC_IMP_EQUIV;
     MULTIPLEX_SPEC_IMP_EQUIV;
     INCLOGIC_SPEC_IMP_EQUIV;
     NEXTNODE_SPEC_IMP_EQUIV] =
    [theorem `INCCON_CORRECT`     `INCCON_SPEC_IMP_EQUIV`;
     theorem `MPLXCON_CORRECT`    `MPLXCON_SPEC_IMP_EQUIV`;
     theorem `MULTIPLEX_CORRECT`  `MULTIPLEX_SPEC_IMP_EQUIV`;
     theorem `INCLOGIC_CORRECT`   `INCLOGIC_SPEC_IMP_EQUIV`;
     theorem `NEXTNODE_CORRECT`   `NEXTNODE_SPEC_IMP_EQUIV`];;
```

```
%----------------------------------------------------------------------------%
%                    The Implementation of COUNTLOGIC                        %
%----------------------------------------------------------------------------%

let ty26 = ":(bool#bool#bool#bool#bool#bool#bool#bool#bool#bool#bool#bool#
             bool#bool#bool#bool#bool#bool#bool#bool#bool#bool#bool#
             bool#bool)
            -> bool";;

let COUNTLOGIC_IMP_DEF =
 new_definition
  (`COUNTLOGIC_IMP_DEF`,
   "COUNTLOGIC_IMP:`ty26(c0,c1,c2,c3,c4,c5,n0,n1,
                         l0,l1,l2,l3,l4,l5,f0,f1,double,
                         c0',c1',c2',c3',c4',c5',n0',n1',double') =
           ? b1 b2 d0 d1 d2 d3 d4 d5.
             INCCON_IMP(n0,n1,b2)                          /\
             MPLXCON_IMP(n0,n1,b1)                         /\
             NEXTNODE_IMP(n0,n1,f0,f1,double,n0',n1')      /\
             INCLOGIC_IMP(c0,c1,c2,c3,c4,c5,b2,
                          d0,d1,d2,d3,d4,d5)               /\
             MULTIPLEX_IMP(d0,d1,d2,d3,d4,d5,
                           l0,l1,l2,l3,l4,l5,b1,
                           c0',c1',c2',c3',c4',c5')        /\
             (double' = f0)");;

%----------------------------------------------------------------------------%
%                     The definition of COUNTLOGIC                           %
%                                                                            %
% let COUNTLOGIC_DEF =                                                       %
%  new_definition                                                            %
%    (`COUNTLOGIC_DEF`,                                                      %
%     "COUNTLOGIC((count,double,node):`major,loadin:word6,func:word2) =      %
%       let twice = EL 0 (BITS2 func) in                                     %
%      (MULTIPLEX(INCLOGIC(count,INCCON node),loadin,MPLXCON node),          %
%        twice,                                                              %
%        NEXTNODE(node,func,double))"                                        %
% on DESIGN theory                                                           %
%----------------------------------------------------------------------------%

let COUNTLOGIC_DEF = definition `DESIGN` `COUNTLOGIC_DEF`;;

%----------------------------------------------------------------------------%
%                    The Specification of COUNTLOGIC                         %
%----------------------------------------------------------------------------%


let COUNTLOGIC_SPEC_DEF =
 new_definition
  (`COUNTLOGIC_SPEC_DEF`,
   "COUNTLOGIC_SPEC:`ty26(c0,c1,c2,c3,c4,c5,n0,n1,
                          l0,l1,l2,l3,l4,l5,f0,f1,double,
                          c0',c1',c2',c3',c4',c5',n0',n1',double') =
           ? b1 b2 d0 d1 d2 d3 d4 d5.
             INCCON_SPEC(n0,n1,b2)                         /\
             MPLXCON_SPEC(n0,n1,b1)                        /\
             NEXTNODE_SPEC(n0,n1,f0,f1,double,n0',n1')     /\
             INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b2,
                           d0,d1,d2,d3,d4,d5)              /\
             MULTIPLEX_SPEC(d0,d1,d2,d3,d4,d5,
                            l0,l1,l2,l3,l4,l5,b1,
                            c0',c1',c2',c3',c4',c5')       /\
             (double' = f0)");;

%----------------------------------------------------------------------------%
%          The equivalence of Specification and Implementation               %
%----------------------------------------------------------------------------%

let COUNTLOGIC_SPEC_IMP_EQUIV =
  prove_thm(`COUNTLOGIC_SPEC_IMP_EQUIV`,
            "COUNTLOGIC_SPEC(c0,c1,c2,c3,c4,c5,n0,n1,
                             l0,l1,l2,l3,l4,l5,f0,f1,double,
                             c0',c1',c2',c3',c4',c5',n0',n1',double') =
             COUNTLOGIC_IMP (c0,c1,c2,c3,c4,c5,n0,n1,
                             l0,l1,l2,l3,l4,l5,f0,f1,double,
                             c0',c1',c2',c3',c4',c5',n0',n1',double')",
            REWRITE_TAC[COUNTLOGIC_SPEC_DEF;COUNTLOGIC_IMP_DEF;
                        INCCON_SPEC_IMP_EQUIV;MPLXCON_SPEC_IMP_EQUIV;
                        MULTIPLEX_SPEC_IMP_EQUIV;INCLOGIC_SPEC_IMP_EQUIV;
```

```
                        NEXTNODE_SPEC_IMP_EQUIV]);;

%----------------------------------------------------------------------------%
%          The relation between Definition and Specification.                %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% We would like to EXPAND out COUNTLOGIC_SPEC_DEF, to get rid of the         %
% existential quantifiers. Normally, we would EXPAND it with MULTIPLEX_DEF_  %
% SPEC_EQUIV etc. to replace the expressions "MULTIPLEX_SPEC(...)" etc.      %
% However, doing just that would then introduce expressions of the form      %
% "V6(...) = ...", with compound left hand sides.  EXPAND cannot cope with   %
% these, so instead, MULTIPLEX_DEF_SPEC_EQUIV and INCLOGIC_DEF_SPEC_EQUIV    %
% are rewritten so they do not introduce expressions of this form.  Then     %
% EXPAND can be used as usual. INCCON and MPLXCON don't cause the problem    %
% because the left hand sides ARE variables.  NEXTNODE_SPEC is not applied   %
% to any of the existentially quantified variables so need not be changed.   %
%                                                                            %
% HOWEVER, though EXPAND does work with these changes, it rewrites too       %
% much and creates huge formulae.  Temporary hacks are shown below.          %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% This is what makes EXPAND applicable, but rewrites too much:               %
%                                                                            %
% let [MULTIPLEX_DEF_SPEC_EQUIV';                                            %
%      INCLOGIC_DEF_SPEC_EQUIV'] =                                           %
%     map (REWRITE_RULE[BITS2_AX;BITS6_AX]) [MULTIPLEX_DEF_SPEC_EQUIV;       %
%                                            INCLOGIC_DEF_SPEC_EQUIV];;      %
% so that                                                                    %
%                                                                            %
% INCLOGIC_DEF_SPEC_EQUIV' =                                                 %
% |- INCLOGIC_SPEC(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =            %
%    (c5' = EL 5(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b)))) /\               %
%    (c4' = EL 4(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b)))) /\               %
%    (c3' = EL 3(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b)))) /\               %
%    (c2' = EL 2(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b)))) /\               %
%    (c1' = EL 1(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b)))) /\               %
%    (c0' = EL 0(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b))))                  %
%                                                                            %
% and similarly for MULTIPLEX_DEF_SPEC_EQUIV'. Then, unfortunately, if we    %
% say                                                                        %
%                                                                            %
% let COUNTLOGIC_EXPAND = EXPAND [INCCON_DEF_SPEC_EQUIV;                     %
%                                 MPLXCON_DEF_SPEC_EQUIV;                    %
%                                 MULTIPLEX_DEF_SPEC_EQUIV';                 %
%                                 INCLOGIC_DEF_SPEC_EQUIV';                  %
%                                 NEXTNODE_DEF_SPEC_EQUIV]                   %
%                                COUNTLOGIC_SPEC_DEF;;                       %
%                                                                            %
% AND if the clause "double' = f0" were in predicate form (for UNFOLD, which %
% is called by EXPAND), we would get too much rewriting:                     %
%                                                                            %
% COUNTLOGIC_EXPAND =                                                        %
% |- COUNTLOGIC_SPEC                                                        %
%    (c0,c1,c2,c3,c4,c5,n0,n1,l0,l1,l2,l3,l4,l5,f0,f1,double,c0',c1',c2',    %
%     c3',c4',c5',n0',n1',double') =                                         %
%    (c5' =                                                                  %
%     EL                                                                     %
%     5                                                                      %
%     (BITS6                                                                 %
%      (MULTIPLEX                                                            %
%       (V6                                                                  %
%        (EL 5(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 4(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 3(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 2(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 1(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 0(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0)))))),   %
%        V6(l5,l4,l3,l2,l1,l0),MPLXCON(V2(n1,n0)))))) /\                     %
%    (c4' =  ...                ...                ...                      %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% These are the temporary hacks to get round all this,                       %
% (where COUNTLOGIC_EXPAND is the desired form of COUNTLOGIC_SPEC_DEF):       %
%----------------------------------------------------------------------------%

let thl = [INCCON_DEF_SPEC_EQUIV;MPLXCON_DEF_SPEC_EQUIV;
```

```
                                MULTIPLEX_DEF_SPEC_EQUIV;
                                INCLOGIC_DEF_SPEC_EQUIV;
                                NEXTNODE_DEF_SPEC_EQUIV] ;;

let th = COUNTLOGIC_SPEC_DEF;;

%----------------------------------------------------------------------------%
% th1 =                                                                      %
% |- COUNTLOGIC_SPEC                                                         %
%    (c0,c1,c2,c3,c4,c5,n0,n1,10,11,12,13,14,15,f0,f1,double,c0',c1',c2',    %
%     c3',c4',c5',n0',n1',double') =                                         %
%    (?b1 b2 d0 d1 d2 d3 d4 d5.                                              %
%      (b2 = INCCON(V2(n1,n0))) /\                                           %
%      (b1 = MPLXCON(V2(n1,n0))) /\                                          %
%      (V2(n1',n0') = NEXTNODE(V2(n1,n0),V2(f1,f0),double)) /\               %
%      (V6(d5,d4,d3,d2,d1,d0) = INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)) /\       %
%      (V6(c5',c4'.c3',c2',c1',c0') =                                        %
%       MULTIPLEX(V6(d5,d4,d3,d2,d1,d0),V6(15,14,13,12,11,10),b1)) /\        %
%      (double' = f0))                                                       %
%----------------------------------------------------------------------------%

let th1 = REWRITE_RULE th1 th;;

%----------------------------------------------------------------------------%
% BITS6_LEMMA =                                                             %
% |- (V6(d0,d1,d2,d3,d4,d5) = INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)) =         %
%    (d0 = EL 5(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)))) /\              %
%    (d1 = EL 4(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)))) /\     ...      %
%----------------------------------------------------------------------------%

let BITS6_LEMMA =
 SPECL
  ["d0:bool";"d1:bool";"d2:bool";"d3:bool";"d4:bool";"d5:bool";
   "INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)"]
  BITS6_AX;;

%----------------------------------------------------------------------------%
% th2 =                                                                      %
% |- COUNTLOGIC_SPEC                                                         %
%    (c0,c1,c2,c3,c4,c5,n0,n1,10,11,12,13,14,15,f0,f1,double,c0',c1',c2',    %
%     c3',c4',c5',n0',n1',double') =                                         %
%    (?b1 b2 d0 d1 d2 d3 d4 d5.                                              %
%      (b2 = INCCON(V2(n1,n0))) /\                                           %
%      (b1 = MPLXCON(V2(n1,n0))) /\                                          %
%      (V2(n1',n0') = NEXTNODE(V2(n1,n0),V2(f1,f0),double)) /\               %
%      ((d5 = EL 5(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)))) /\            %
%       (d4 = EL 4(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)))) /\            %
%       (d3 = EL 3(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)))) /\            %
%       (d2 = EL 2(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)))) /\            %
%       (d1 = EL 1(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2)))) /\            %
%       (d0 = EL 0(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),b2))))) /\           %
%      (V6(c5',c4',c3',c2',c1',c0') =                                        %
%       MULTIPLEX(V6(d5,d4,d3,d2,d1,d0),V6(15,14,13,12,11,10),b1)) /\        %
%      (double' = f0))                                                       %
%----------------------------------------------------------------------------%

let th2 = PURE_REWRITE_RULE[BITS6_LEMMA]th1;;

%----------------------------------------------------------------------------%
% th3 =                                                                      %
% |- COUNTLOGIC_SPEC                                                         %
%    (c0,c1,c2,c3,c4,c5,n0,n1,10,11,12,13,14,15,f0,f1,double,c0',c1',c2',    %
%     c3',c4',c5',n0',n1',double') =                                         %
%    (?b1 b2 d0 d1 d2 d3 d4 d5.                                              %
%      (V6(c5',c4',c3',c2',c1',c0') =                                        %
%       MULTIPLEX                                                            %
%       (V6                                                                  %
%        (EL 5(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 4(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 3(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 2(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 1(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),    %
%         EL 0(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0)))))),   %
%        V6(15,14,13,12,11,10),MPLXCON(V2(n1,n0)))) /\                       %
%      (d5 =                                                                 %
%       EL 5(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0)))))) /\   %
%      (d4 =                                                                 %
%       EL 4(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0)))))) /\   %
%      (d3 =                                                                 %
```

```
%       EL 3(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))) /\   %
%      (d2 =                                                                %
%       EL 2(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))) /\   %
%      (d1 =                                                                %
%       EL 1(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))) /\   %
%      (d0 =                                                                %
%       EL 0(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))) /\   %
%      (b2 = INCCON(V2(n1,n0))) /\                                          %
%      (b1 = MPLXCON(V2(n1,n0))) /\                                         %
%      (V2(n1',n0') = NEXTNODE(V2(n1,n0),V2(f1,f0),double)) /\              %
%      (double' = f0))                                                      %
%-------------------------------------------------------------------------%

let th3 = OLD_UNWIND_RULE th2;;

%-------------------------------------------------------------------------%
% th4 =                                                                   %
% |- COUNTLOGIC_SPEC                                                     %
%    (c0,c1,c2,c3,c4,c5,n0,n1,10,11,12,13,14,15,f0,f1,double,c0',c1',c2', %
%     c3',c4',c5',n0',n1',double') =                                      %
%    (V6(c5',c4',c3',c2',c1',c0') =                                       %
%     MULTIPLEX                                                           %
%     (V6                                                                 %
%      (EL 5(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),  %
%       EL 4(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),  %
%       EL 3(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),  %
%       EL 2(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),  %
%       EL 1(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),  %
%       EL 0(BITS6(INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))))),  %
%      V6(15,14,13,12,11,10),MPLXCON(V2(n1,n0)))) /\                     %
%    (V2(n1',n0') = NEXTNODE(V2(n1,n0),V2(f1,f0),double)) /\              %
%    (double' = f0)                                                       %
%-------------------------------------------------------------------------%

let th4 = PRUNE_RULE th3;;

%-------------------------------------------------------------------------%
% COUNTLOGIC_EXPAND =                                                     %
% |- COUNTLOGIC_SPEC                                                     %
%    (c0,c1,c2,c3,c4,c5,n0,n1,10,11,12,13,14,15,f0,f1,double,c0',c1',c2', %
%     c3',c4',c5',n0',n1',double') =                                      %
%    (V6(c5',c4',c3',c2',c1',c0') =                                       %
%     MULTIPLEX                                                           %
%     (INCLOGIC(V6(c5,c4,c3,c2,c1,c0),INCCON(V2(n1,n0))),                %
%      V6(15,14,13,12,11,10),MPLXCON(V2(n1,n0)))) /\                     %
%    (V2(n1',n0') = NEXTNODE(V2(n1,n0),V2(f1,f0),double)) /\              %
%    (double' = f0)                                                       %
%-------------------------------------------------------------------------%

let COUNTLOGIC_EXPAND = REWRITE_RULE [V6_THM1] th4;;

%-------------------------------------------------------------------------%
% That is the end of the hacks.                                           %
%-------------------------------------------------------------------------%


let COUNTLOGIC_DEF_SPEC_EQUIV =
    prove_thm (`COUNTLOGIC_DEF_SPEC_EQUIV`,
 "COUNTLOGIC_SPEC(c0,c1,c2,c3,c4,c5,n0,n1,
                  10,11,12,13,14,15,f0,f1,double,
                  c0',c1',c2',c3',c4',c5',n0',n1',double') =
  ( (V6(c5',c4',c3',c2',c1',c0'),double',V2(n1',n0')) =
    COUNTLOGIC((V6(c5,c4,c3,c2,c1,c0),double,V2(n1,n0)),
               V6(15,14,13,12,11,10),
               V2(f1,f0)) )",
           REWRITE_TAC[COUNTLOGIC_EXPAND;COUNTLOGIC_DEF;LET_DEF;EL_BITS2_AX]
           THEN CONV_TAC(DEPTH_CONV BETA_CONV)
           THEN REWRITE_TAC[PAIR_SPLIT]
           THEN CONJ_SET_TAC);;

%-------------------------------------------------------------------------%
%    The Relation between Definition and Implementation                   %
%-------------------------------------------------------------------------%

let COUNTLOGIC_DEF_IMP_EQUIV =
  prove_thm(`COUNTLOGIC_DEF_IMP_EQUIV`,
           "COUNTLOGIC_IMP (c0,c1,c2,c3,c4,c5,n0,n1,
                            10,11,12,13,14,15,f0,f1,double,
                            c0',c1',c2',c3',c4',c5',n0',n1',double') =
```

```
            ( (V6(c5',c4',c3',c2',c1',c0'),double',V2(n1',n0')) =
             COUNTLOGIC((V6(c5,c4,c3,c2,c1,c0),double,V2(n1,n0)),
                         V6(15,14,13,12,11,10),
                         V2(f1,f0)) )",
            REWRITE_TAC[SYM COUNTLOGIC_SPEC_IMP_EQUIV;
                        COUNTLOGIC_DEF_SPEC_EQUIV]);;

close_theory();;
```

```
%============================================================================%
% The ELLA definitions (p. 16) are translated into HOL, and shown equi-      %
% valent to the implementation level.                                        %
%============================================================================%

new_theory `ELLA_CORRECT`;;

map new_parent [`INCCON_CORRECT`;`MPLXCON_CORRECT`;`MULTIPLEX_CORRECT`;
                `INCLOGIC_CORRECT`;`NEXTNODE_CORRECT`];;

loadt `~/counter/utilities`;;

let V2_DEF = definition `WORD_WIDTHS` `V2_DEF`;;

let V6_DEF = definition `WORD_WIDTHS` `V6_DEF`;;

%----------------------------------------------------------------------------%
% We introduce gate constants, of a functional rather than relational form. %
% Lower case is used to distinguish them from the constants on GATES.th.     %
%----------------------------------------------------------------------------%

let nor_DEF =
new_definition
 (`nor_DEF`,
  "nor(in1,in2) =  ~(in1 \/ in2)");;

let nand_DEF =
new_definition
 (`nand_DEF`,
  "nand(in1,in2) =  ~(in1 /\ in2)");;

let nand3_DEF =
new_definition
 (`nand3_DEF`,
  "nand3(in1,in2,in3) =  ~(in1 /\ in2 /\ in3)");;

let nand4_DEF =
new_definition
 (`nand4_DEF`,
  "nand4(in1,in2,in3,in4) =  ~(in1 /\ in2 /\ in3 /\ in4)");;

let inv_DEF =
new_definition
 (`inv_DEF`,
  "inv in = ~in");;

let xnor_DEF =
new_definition
  (`xnor_DEF`,
   "xnor(in1:bool,in2:bool) = (in1 = in2)");;

%----------------------------------------------------------------------------%
% To translate ELLA to HOL, we need the functions BIT2 and BIT6 for          %
% the indexing notation, and LISTIFY for the iteration notation.  Note that  %
% indices in ELLA go from 1 to n left to right, and in HOL from (n-1) to 0   %
% left to right -- or 0 to (n-1) from right to left.                         %
% Note also a "let f x = y" expression must become "let f = \x.y".           %
% Also, we can't say "\(x,y)" but need "\ x y".                              %
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% BIT6 5 #abcdef = b                                                         %
% BIT6 1 #abcdef = f                                                         %
%----------------------------------------------------------------------------%

let BIT2_DEF =
new_definition(`BIT2_DEF`,
               "(BIT2:word2 -> num -> bool) w n =
                 EL (n - 1) (BITS2 w)");;

let BIT6_DEF =
new_definition(`BIT6_DEF`,
               "(BIT6:word6 -> num -> bool) w n =
                 EL (n - 1) (BITS6 w)");;

%----------------------------------------------------------------------------%
% LISTIFY 6 f = [f 6;f 5;f 4;f 3;f 2;f 1]                                    %
%----------------------------------------------------------------------------%
```

```
new_prim_rec_definition(`LISTIFY_DEF`,
                        "((LISTIFY:num -> (num -> *) -> * list) 0 f = NIL)/\
                          (LISTIFY(SUC n) f = CONS(f(SUC n))
                                                  (LISTIFY n f))");;

%----------------------------------------------------------------------------%
% The p. 16 definitions are curried here but need not be -- the difference   %
% is in the formulation rather than the proof.                               %
%----------------------------------------------------------------------------%

let MPLEXCIRC_DEF =
new_definition(`MPLEXCIRC_DEF`,
               "MPLEXCIRC (incout:word6) (loadin:word6) (mplxsel:bool) =
                let BITSEL = \ (lbit:bool)(lsel:bool)
                               (incbit:bool)(incsel:bool).
                             nand(nand(lbit,lsel),nand(incbit,incsel))
                in let mplxselbar = inv mplxsel
                in WORD6(V(LISTIFY 6
                           (\k.BITSEL(BIT6 loadin k) mplxselbar
                                    (BIT6 incout k) mplxsel))
                         )");;

let INCCIRC_DEF =
new_definition(`INCCIRC_DEF`,
               "INCCIRC (count:word6) (noinc:bool) =
                let noincbar = inv noinc
                in let ic1 = xnor(BIT6 count 1,noinc)
                in let ic2 = xnor(BIT6 count 2,nand(noincbar,BIT6 count 1))
                in let ic3 = xnor(BIT6 count 3,nand3(noincbar,BIT6 count 1,
                                                     BIT6 count 2))
                in let carry4bar = nand4(noincbar,BIT6 count 1,BIT6 count 2,
                                         BIT6 count 3)
                in let ic4 = xnor(BIT6 count 4, carry4bar)
                in let carry4 = inv carry4bar
                in let ic5 = xnor(BIT6 count 5,nand(carry4,BIT6 count 4))
                in let ic6 = xnor(BIT6 count 6, nand3(carry4,BIT6 count 4,
                                                      BIT6 count 5))
                in V6(ic6,ic5,ic4,ic3,ic2,ic1)" );;

let CONTROLCIR_DEF =
new_definition(`CONTROLCIR_DEF`,
               "CONTROLCIR (node:word2)(func:word2)(double:bool) =
                let inccon = nor(BIT2 node 1,BIT2 node 2)
                in let mplxcon = nand(BIT2 node 1,BIT2 node 2)
                in let common = nand3(inccon,inv(BIT2 func 2),
                                      BIT2 func 1)
                in let nextnode1 = nand(common,nand(inccon,BIT2 func 2))
                in let nextnode2 = nand(common,nand3(double,BIT2 node 1,
                                                     inv(BIT2 node 2)))
                in (inccon,mplxcon,V2(nextnode2,nextnode1))" );;

%----------------------------------------------------------------------------%
% The relation between MPLEXCIRC and MULTIPLEX_IMP:                          %
%----------------------------------------------------------------------------%

let LISTIFY_THM = theorem `-` `LISTIFY_DEF`;;

let MULTIPLEX_EXPAND = EXPAND [definition `GATES` `NAND_DEF`;
                               definition `GATES` `INV_DEF`]
                              (definition `MULTIPLEX_CORRECT`
                                          `MULTIPLEX_IMP_DEF`);;

%----------------------------------------------------------------------------%
% "6 = SUC(SUC(SUC(SUC(SUC(SUC 0)))))"                                       %
% is needed to rewrite "LISTIFY 6"                                           %
%----------------------------------------------------------------------------%

let six_def = REWRITE_RULE[](REDEPTH_CONV num_CONV "6");;

%----------------------------------------------------------------------------%
% ["SUC 0 = 1"; ... ; "SUC(SUC(SUC(SUC(SUC(SUC 0))))) = 6"]                  %
% to restore effects of the above used by LISTIFY                            %
%----------------------------------------------------------------------------%

let one_to_six_def' =
    map(\x.SYM(REWRITE_RULE[](REDEPTH_CONV num_CONV x)))
       ["1";"2";"3";"4";"5";"6"];;

let BITS6_V6_THM = theorem `WORD_WIDTHS` `BITS6_V6_THM`;;
```

```
let [LIST_AX1;WORD6_AX2;V_AX1] = map (axiom `WORD_WIDTHS`)
                                     [`LIST_AX1`;`WORD6_AX2`;`V_AX1`];;

let MULTIPLEX_ELLA_IMP_EQUIV =
  prove_thm(`MULTIPLEX_ELLA_IMP_EQUIV`,
            "MULTIPLEX_IMP(c0,c1,c2,c3,c4,c5,10,11,12,13,14,15,b,
                           c0',c1',c2',c3',c4',c5') =
             (V6(c5',c4',c3',c2',c1',c0') =
              MPLEXCIRC (V6(c5,c4,c3,c2,c1,c0))(V6(15,14,13,12,11,10)) b)",
            REWRITE_TAC[MULTIPLEX_EXPAND;MPLEXCIRC_DEF]
            THEN REWRITE_TAC[inv_DEF;BIT6_DEF;LET_DEF]
            THEN BOOL_CASES_TAC "b:bool"
            THEN REWRITE_TAC [six_def;LISTIFY_THM;BITS6_V6_THM]
            THEN REWRITE_TAC one_to_six_def'
            THEN CONV_TAC(TOP_DEPTH_CONV BETA_CONV THENC DIF_CONV THENC
                          EL_CONV)
            THEN REWRITE_TAC[V6_DEF;nand_DEF;LIST_AX1;WORD6_AX2;V_AX1]
);;

%-----------------------------------------------------------------------------%
% The relation between INCCIRC and INCLOGIC_IMP:                              %
%-----------------------------------------------------------------------------%

let INCLOGIC_EXPAND = EXPAND [definition `GATES` `NAND_DEF`;
                              definition `GATES` `INV_DEF`;
                              definition `GATES` `NAND3_DEF`;
                              definition `GATES` `NAND4_DEF`;
                              definition `GATES` `XNOR_DEF`]
                              (definition `INCLOGIC_CORRECT`
                                          `INCLOGIC_IMP_DEF`);;

let INCLOGIC_ELLA_IMP_EQUIV =
   prove_thm(`INCLOGIC_ELLA_IMP_EQUIV`,
             "INCLOGIC_IMP(c0,c1,c2,c3,c4,c5,b,c0',c1',c2',c3',c4',c5') =
              (V6(c5',c4',c3',c2',c1',c0') =
               INCCIRC (V6(c5,c4,c3,c2,c1,c0)) b )",
             REWRITE_TAC[INCLOGIC_EXPAND;INCCIRC_DEF]
             THEN REWRITE_TAC[inv_DEF;xnor_DEF;nand_DEF;nand3_DEF;nand4_DEF;
                              BIT6_DEF;LET_DEF]
             THEN BOOL_CASES_TAC "b:bool"
             THEN REWRITE_TAC [BITS6_V6_THM]
             THEN CONV_TAC(TOP_DEPTH_CONV BETA_CONV THENC DIF_CONV THENC
                           EL_CONV)
             THEN REWRITE_TAC[V6_DEF;LIST_AX1;WORD6_AX2;V_AX1]
             THEN CONJ_SET_TAC
);;

%-----------------------------------------------------------------------------%
% Relation between NEXTNODE_IMP and CONTROLCIR:                               %
% Note that CONTROLCIR combines NEXTNODE_IMP, MPLXCON_IMP and INCCON_IMP,     %
% just as they are combined in Fig.6.                                         %
%-----------------------------------------------------------------------------%

let BITS2_V2_THM = theorem `WORD_WIDTHS` `BITS2_V2_THM`;;

let NEXTNODE_EXPAND = EXPAND [definition `GATES` `NAND_DEF`;
                              definition `GATES` `INV_DEF`;
                              definition `GATES` `NAND3_DEF`;
                              definition `GATES` `NOR_DEF`]
                              (definition `NEXTNODE_CORRECT`
                                          `NEXTNODE_IMP_DEF`);;

%-----------------------------------------------------------------------------%
% The following two theorems are from WORD_WIDTHS:                            %
%-----------------------------------------------------------------------------%

let WORD2_TH2 = theorem `WORD_WIDTHS` `WORD2_TH2`;;

let V_TH3 = theorem `WORD_WIDTHS` `V_TH3`;;

let NEXTNODE_ELLA_IMP_EQUIV =
  prove_thm(`NEXTNODE_ELLA_IMP_EQUIV`,
            "NEXTNODE_IMP(n0,n1,f0,f1,double,n0',n1') =
             (V2(n1',n0') = SND(SND(CONTROLCIR(V2(n1,n0))(V2(f1,f0))
                                               double)))",
            REWRITE_TAC[NEXTNODE_EXPAND;CONTROLCIR_DEF]
            THEN REWRITE_TAC[inv_DEF;nand_DEF;nand3_DEF;nor_DEF;
                             BIT2_DEF;LET_DEF]
```

```
            THEN REWRITE_TAC [BITS2_V2_THM]
            THEN CONV_TAC(TOP_DEPTH_CONV BETA_CONV THENC DIF_CONV THENC
                          EL_CONV)
            THEN REWRITE_TAC[V2_DEF;SND;LIST_AX1;WORD2_TH2;V_TH3]
            THEN BOOL_CASES_TAC "n0:bool"
            THEN BOOL_CASES_TAC "n1:bool"
            THEN REWRITE_TAC []
            THENL [CONJ_SET_TAC;CONJ_SET_TAC;CONJ_SET_TAC;
                   BOOL_CASES_TAC "f1:bool"
                   THEN REWRITE_TAC[]
                   THEN CONJ_SET_TAC]
);;

%----------------------------------------------------------------------------%
% Relation between MPLXCON_IMP and CONTROLCIR:                               %
%----------------------------------------------------------------------------%

let MPLXCON_IMP_DEF = definition `MPLXCON_CORRECT` `MPLXCON_IMP_DEF`;;

let NAND_DEF = definition `GATES` `NAND_DEF`;;

let MPLXCON_ELLA_IMP_EQUIV =
  prove_thm(`MPLXCON_ELLA_IMP_EQUIV`,
            "MPLXCON_IMP(n0,n1,b) =
             (b = FST(SND(CONTROLCIR (V2(n1,n0)) (V2(f1,f0)) double)))",
            REWRITE_TAC[MPLXCON_IMP_DEF;CONTROLCIR_DEF;NAND_DEF]
            THEN REWRITE_TAC[inv_DEF;nand_DEF;nand3_DEF;nor_DEF;
                             BIT2_DEF;LET_DEF]
            THEN REWRITE_TAC [BITS2_V2_THM]
            THEN CONV_TAC(TOP_DEPTH_CONV BETA_CONV THENC DIF_CONV THENC
                          EL_CONV)
            THEN REWRITE_TAC[V2_DEF;FST;SND;LIST_AX1;WORD2_TH2;V_TH3]
);;

%----------------------------------------------------------------------------%
% Relation between INCCON_IMP and CONTROLCIR:                                %
%----------------------------------------------------------------------------%

let NOR_DEF = definition `GATES` `NOR_DEF`;;

let INCCON_IMP_DEF = definition `INCCON_CORRECT` `INCCON_IMP_DEF`;;

let INCCON_ELLA_IMP_EQUIV =
  prove_thm(`INCCON_ELLA_IMP_EQUIV`,
            "INCCON_IMP(n0,n1,b) =
             (b = FST(CONTROLCIR (V2(n1,n0)) (V2(f1,f0)) double))",
            REWRITE_TAC [INCCON_IMP_DEF;CONTROLCIR_DEF;NOR_DEF]
            THEN REWRITE_TAC[inv_DEF;nand_DEF;nand3_DEF;nor_DEF;
                             BIT2_DEF;LET_DEF]
            THEN REWRITE_TAC [BITS2_V2_THM]
            THEN CONV_TAC(TOP_DEPTH_CONV BETA_CONV THENC DIF_CONV THENC
                          EL_CONV)
            THEN REWRITE_TAC[FST]
);;


close_theory();;
```

```
%============================================================================%
% On this theory, we formulate and prove the correctness and termination of  %
% the counter, with time taken into account.  The main theorems used are:    %
%                                                                            %
%     COUNTLOGIC_DEF_IMP_EQUIV from CIRCUIT_CORRECT.th                       %
%                     (which relates COUNTLOGIC to COUNTLOGIC_IMP)           %
%     HOST_CORRECT from HOST_CORRECT theory                                  %
%                     (which states the correctness of NEXT in terms         %
%                      of COUNTER)                                           %
%     HOST_TERMINATES from HOST_CORRECT theory                               %
%                     (which states the termination of NEXT)                 %
%     NEXT_COUNTLOGIC_EQUIV from DESIGN_CORRECT                              %
%                     (which states the equivalence of NEXT and COUNTLOGIC)  %
%     COUNTLOGIC_CORRECT from this theory                                    %
%                     (which joins HOST_CORRECT and NEXT_COUNTLOGIC_EQUIV    %
%                      to prove the equivalence of COUNTLOGIC and COUNTER)   %
%     COUNTLOGIC_TERMINATES from this theory                                 %
%                     (which joins HOST_TERMINATES and NEXT_COUNTLOGIC_EQUIV %
%                      to prove the termination of COUNTLOGIC)               %
%     COUNTLOGIC_DEF_LEMMA from this theory                                  %
%                     (which relates COUNTLOGIC to the circuit with time,    %
%                      called CIRCUIT_IMP and defined on this theory).       %
%                                                                            %
% The main results, on this theory, are:                                     %
%                                                                            %
%     COUNTER_PCORRECT (which states the partial correctness of the circuit  %
%                       by relating CIRCUIT_IMP and COUNTER), and            %
%   COUNTER_TERMINATES (which states the termination of CIRCUIT_IMP) and     %
%     COUNTER_CORRECT  (which states the total correctness of the circuit).  %
%============================================================================%

new_theory `COUNTER_CORRECT`;;

map new_parent [`CIRCUIT_CORRECT`;`DESIGN_CORRECT`;`HOST_CORRECT`];;

%----------------------------------------------------------------------------%
% We recall the definitions of "V2", "V6", "COUNT" and "NODE".               %
% "VS2" and "VS6" are handy abbreviations.                                   %
%----------------------------------------------------------------------------%

let V2_DEF = definition `WORD_WIDTHS` `V2_DEF`;;

let V6_DEF = definition `WORD_WIDTHS` `V6_DEF`;;

let VS2_DEF =
new_definition(`VS2_DEF`,
               "VS2(s1:num -> bool,s0:num -> bool) t = V2(s1 t,s0 t)");;

let VS6_DEF =
new_definition(`VS6_DEF`,
               "VS6(s5,s4,s3,s2,s1,s0) (t:num) =
                V6(s5 t,s4 t,s3 t,s2 t,s1 t,s0 t)");;

let [COUNT_DEF;NODE_DEF] = map (definition `HOST_CORRECT`)
                               [`COUNT_DEF`;`NODE_DEF`];;

%----------------------------------------------------------------------------%
% We now relate COUNTLOGIC and COUNTER, via NEXT:                            %
%----------------------------------------------------------------------------%

let [HOST_PCORRECT;HOST_TERMINATES] = map (theorem `HOST_CORRECT`)
                                          [`HOST_PCORRECT`;`HOST_TERMINATES`];;

let NEXT_COUNTLOGIC_EQUIV = theorem `DESIGN_CORRECT` `NEXT_COUNTLOGIC_EQUIV`;;

let COUNTLOGIC_PCORRECT =
  prove_thm(`COUNTLOGIC_PCORRECT`,
          "(!t. state_sig(t+1) =
           COUNTLOGIC(state_sig t,loadin_sig t,func_sig t)) /\
           (NODE(state_sig t)=#00)                                          /\
           NEXTTIME (t,t') (\x. NODE(state_sig x)=#00)
           ==>
           (COUNT(state_sig t') =
           COUNTER(COUNT(state_sig t),loadin_sig(t+1),func_sig t))",
           REWRITE_TAC[SYM NEXT_COUNTLOGIC_EQUIV;
                       HOST_PCORRECT] );;

let COUNTLOGIC_TERMINATES =
  prove_thm(`COUNTLOGIC_TERMINATES`,
```

```
   "(!t.
       state_sig(t + 1) =
       COUNTLOGIC(state_sig t,loadin_sig t,func_sig t)) /\
       (NODE(state_sig t) = #00)
       ==>
       ?t'.NEXTTIME(t,t')(\x.NODE(state_sig x)=#00)",
       REWRITE_TAC[SYM NEXT_COUNTLOGIC_EQUIV;
                   HOST_TERMINATES]);;

%----------------------------------------------------------------------------%
% The theorem COUNTER_PCORRECT will tie together the 3 main theorems so far, %
% the two just proved, with this one:                                        %
%----------------------------------------------------------------------------%

let COUNTLOGIC_DEF_IMP_EQUIV = theorem `CIRCUIT_CORRECT`
                                       `COUNTLOGIC_DEF_IMP_EQUIV`;;

%----------------------------------------------------------------------------%
% We first define a boolean latch, so we can represent the whole of Fig. 5,  %
% including the latches with time delay.  (COUNTLOGIC represents the combin- %
% atorial part only of Fig. 5.)                                              %
%----------------------------------------------------------------------------%

let LATCH_DEF =
 new_definition
  (`LATCH_DEF`,
   "LATCH(in:num -> bool,out:num -> bool) = !t. out(t+1) = in t");;

%----------------------------------------------------------------------------%
% COUNTLOGIC_DEV  is like COUNTLOGIC_IMP but with time taken into account.   %
%----------------------------------------------------------------------------%

let COUNTLOGIC_DEV_DEF =
 new_definition
  (`COUNTLOGIC_DEV_DEF`,
   "COUNTLOGIC_DEV
    (c0_sig,c1_sig,c2_sig,c3_sig,c4_sig,c5_sig,n0_sig,n1_sig,
     l0_sig,l1_sig,l2_sig,l3_sig,l4_sig,l5_sig,f0_sig,f1_sig,double_sig,
     c0'_sig,c1'_sig,c2'_sig,c3'_sig,c4'_sig,c5'_sig,
     n0'_sig,n1'_sig,double'_sig) =
!t:num. COUNTLOGIC_IMP
    (c0_sig t,c1_sig t,c2_sig t,c3_sig t,c4_sig t,c5_sig t,
     n0_sig t,n1_sig t,
     l0_sig t,l1_sig t,l2_sig t,l3_sig t,l4_sig t,l5_sig t,
     f0_sig t,f1_sig t,double_sig t,
     c0'_sig t,c1'_sig t,c2'_sig t,c3'_sig t,c4'_sig t,c5'_sig t,
     n0'_sig t,n1'_sig t,double'_sig t)");;

%----------------------------------------------------------------------------%
% CIRCUIT_IMP describes the whole of Fig. 6, with time.                      %
%----------------------------------------------------------------------------%

let CIRCUIT_IMP_DEF =
 new_definition
  (`CIRCUIT_IMP_DEF`,
   "CIRCUIT_IMP
    (((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),
     double_sig,
     (n1_sig,n0_sig)),
     (l5_sig,l4_sig,l3_sig,l2_sig,l1_sig,l0_sig),
     (f1_sig,f0_sig)) =
    ?c0'_sig  c1'_sig  c2'_sig  c3'_sig  c4'_sig  c5'_sig double'_sig
     n0'_sig  n1'_sig.
    (COUNTLOGIC_DEV
     (c0_sig,c1_sig,c2_sig,c3_sig,c4_sig,c5_sig,n0_sig,n1_sig,
      l0_sig,l1_sig,l2_sig,l3_sig,l4_sig,l5_sig,f0_sig,f1_sig,double_sig,
      c0'_sig,c1'_sig,c2'_sig,c3'_sig,c4'_sig,c5'_sig,
      n0'_sig,n1'_sig,double'_sig)          /\
     LATCH(c5'_sig,c5_sig)                  /\
     LATCH(c4'_sig,c4_sig)                  /\
     LATCH(c3'_sig,c3_sig)                  /\
     LATCH(c2'_sig,c2_sig)                  /\
     LATCH(c1'_sig,c1_sig)                  /\
     LATCH(c0'_sig,c0_sig)                  /\
     LATCH(n1'_sig,n1_sig)                  /\
     LATCH(n0'_sig,n0_sig)                  /\
     LATCH(double'_sig,double_sig))");;

%----------------------------------------------------------------------------%
```

```
% The following lemma is needed for the main correctness theorem.  It        %
% relates the time-free definition, COUNTLOGIC, over time, to the circuit     %
% description CIRCUIT_IMP.  CIRCUIT_IMP is ultimately in terms of gates.      %
% Note that COUNT, DOUBLE and NODE, in Fig. 5, "feed back" into COUNTLOGIC,   %
% and this is reflected in the relation of the state at time t to the state   %
% at time (t + 1).                                                           %
%---------------------------------------------------------------------------%
%---------------------------------------------------------------------------%
% For convenience, the lemma is proved in a forward way, as follows:         %
%---------------------------------------------------------------------------%
%---------------------------------------------------------------------------%
% We rewrite CIRCUIT_IMP_DEF in terms of COUNTLOGIC_DEF_IMP_EQUIV and so on, %
% to replace COUNTLOGIC_DEV by COUNTLOGIC:                                   %
%                                                                            %
% th1 =                                                                      %
% |- CIRCUIT_IMP                                                            %
%    (((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),double_sig,n1_sig,        %
%      n0_sig),(15_sig,14_sig,13_sig,12_sig,11_sig,10_sig),f1_sig,f0_sig) =  %
%    (?c0'_sig c1'_sig c2'_sig c3'_sig c4'_sig c5'_sig double'_sig           %
%      n0'_sig n1'_sig.                                                      %
%      (!t.                                                                  %
%        V6(c5'_sig t,c4'_sig t,c3'_sig t,c2'_sig t,c1'_sig t,c0'_sig t),    %
%        double'_sig t,V2(n1'_sig t,n0'_sig t) =                             %
%        COUNTLOGIC                                                          %
%        ((V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),        %
%          double_sig t,V2(n1_sig t,n0_sig t)),                              %
%         V6(15_sig t,14_sig t,13_sig t,12_sig t,11_sig t,10_sig t),         %
%         V2(f1_sig t,f0_sig t))) /\                                         %
%      (!t. c5_sig(t + 1) = c5'_sig t) /\                                    %
%      (!t. c4_sig(t + 1) = c4'_sig t) /\                                    %
%      (!t. c3_sig(t + 1) = c3'_sig t) /\                                    %
%      (!t. c2_sig(t + 1) = c2'_sig t) /\                                    %
%      (!t. c1_sig(t + 1) = c1'_sig t) /\                                    %
%      (!t. c0_sig(t + 1) = c0'_sig t) /\                                    %
%      (!t. n1_sig(t + 1) = n1'_sig t) /\                                    %
%      (!t. n0_sig(t + 1) = n0'_sig t) /\                                    %
%      (!t. double_sig(t + 1) = double'_sig t))                              %
%---------------------------------------------------------------------------%

let th1 = REWRITE_RULE [COUNTLOGIC_DEV_DEF;LATCH_DEF;
                        COUNTLOGIC_DEF_IMP_EQUIV] CIRCUIT_IMP_DEF;;

%---------------------------------------------------------------------------%
% We reverse the order of the latch clauses so they can be used as           %
% substitutions in unwinding.  That is, "s(x + 1)=t" becomes "t=s(x + 1)".   %
%                                                                            %
% swap = |- !s' t s. (s(t + 1) = s' t) = (s' t = s(t + 1))                   %
%                                                                            %
% th2 =                                                                      %
% |- CIRCUIT_IMP                                                            %
%   (((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),double_sig,n1_sig,         %
%      n0_sig),(15_sig,14_sig,13_sig,12_sig,11_sig,10_sig),f1_sig,f0_sig) =  %
%    (?c0'_sig c1'_sig c2'_sig c3'_sig c4'_sig c5'_sig double'_sig           %
%      n0'_sig n1'_sig.                                                      %
%      (!t.                                                                  %
%        V6(c5'_sig t,c4'_sig t,c3'_sig t,c2'_sig t,c1'_sig t,c0'_sig t),    %
%        double'_sig t,V2(n1'_sig t,n0'_sig t) =                             %
%        COUNTLOGIC                                                          %
%        ((V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),        %
%          double_sig t,V2(n1_sig t,n0_sig t)),                              %
%         V6(15_sig t,14_sig t,13_sig t,12_sig t,11_sig t,10_sig t),         %
%         V2(f1_sig t,f0_sig t))) /\                                         %
%      (!t. c5'_sig t = c5_sig(t + 1)) /\                                    %
%      (!t. c4'_sig t = c4_sig(t + 1)) /\                                    %
%      (!t. c3'_sig t = c3_sig(t + 1)) /\                                    %
%      (!t. c2'_sig t = c2_sig(t + 1)) /\                                    %
%      (!t. c1'_sig t = c1_sig(t + 1)) /\                                    %
%      (!t. c0'_sig t = c0_sig(t + 1)) /\                                    %
%      (!t. n1'_sig t = n1_sig(t + 1)) /\                                    %
%      (!t. n0'_sig t = n0_sig(t + 1)) /\                                    %
%      (!t. double'_sig t = double_sig(t + 1)))                              %
%---------------------------------------------------------------------------%

let swap =
GEN "s'"(GEN "t"(GEN "s"
(SPEC "(s':num -> bool) t"(SPEC "(s:num -> bool)(t + 1)"
                                (INST_TYPE[":bool",":*"]EQ_SYM_EQ))))));;

let th2 = REWRITE_RULE[swap]th1;;
```

```
%----------------------------------------------------------------------------%
% We unwind, so that the latch clauses disappear, and their substitutions    %
% are made.                                                                  %
%                                                                            %
% th3 =                                                                      %
% |- CIRCUIT_IMP                                                             %
%    (((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),double_sig,n1_sig,        %
%      n0_sig),(15_sig,14_sig,13_sig,12_sig,11_sig,10_sig),f1_sig,f0_sig) =  %
%    (?c0'_sig c1'_sig c2'_sig c3'_sig c4'_sig c5'_sig double'_sig           %
%      n0'_sig n1'_sig.                                                      %
%      (!t. c5'_sig t = c5_sig(t + 1)) /\                                    %
%      (!t. c4'_sig t = c4_sig(t + 1)) /\                                    %
%      (!t. c3'_sig t = c3_sig(t + 1)) /\                                    %
%      (!t. c2'_sig t = c2_sig(t + 1)) /\                                    %
%      (!t. c1'_sig t = c1_sig(t + 1)) /\                                    %
%      (!t. c0'_sig t = c0_sig(t + 1)) /\                                    %
%      (!t. n1'_sig t = n1_sig(t + 1)) /\                                    %
%      (!t. n0'_sig t = n0_sig(t + 1)) /\                                    %
%      (!t. double'_sig t = double_sig(t + 1)) /\                            %
%      (!t.                                                                  %
%        V6                                                                  %
%        (c5_sig(t + 1),c4_sig(t + 1),c3_sig(t + 1),c2_sig(t + 1),           %
%         c1_sig(t + 1),c0_sig(t + 1)),double_sig(t + 1),                    %
%        V2(n1_sig(t + 1),n0_sig(t + 1)) =                                   %
%        COUNTLOGIC                                                          %
%        ((V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),        %
%          double_sig t,V2(n1_sig t,n0_sig t)),                              %
%         V6(15_sig t,14_sig t,13_sig t,12_sig t,11_sig t,10_sig t),         %
%         V2(f1_sig t,f0_sig t))))                                           %
%----------------------------------------------------------------------------%

let th3 = UNWIND_ALL_RULE [] th2;;

%----------------------------------------------------------------------------%
% We eliminate the existential quantifier by repeated substitutions.         %
%----------------------------------------------------------------------------%

let th4 = PRUNEF_RULE th3;;

%----------------------------------------------------------------------------%
% th4 IS the following, so we do a nominal tactical proof to get the lemma:  %
%----------------------------------------------------------------------------%

let COUNTLOGIC_DEF_LEMMA =
   prove_thm(`COUNTLOGIC_DEF_LEMMA`,
   "CIRCUIT_IMP(((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),
                 double_sig,
                 (n1_sig,n0_sig)),
                (15_sig,14_sig,13_sig,12_sig,11_sig,10_sig),
                (f1_sig,f0_sig)) =
    !t. (V6(c5_sig(t+1),c4_sig(t+1),c3_sig(t+1),
            c2_sig(t+1),c1_sig(t+1),c0_sig(t+1)),
         double_sig (t+1),
         V2(n1_sig(t+1),n0_sig(t+1))) =
         COUNTLOGIC((V6(c5_sig t,c4_sig t,c3_sig t,
                        c2_sig t,c1_sig t,c0_sig t),
                     double_sig t,
                     V2(n1_sig t,n0_sig t)),
                    V6(15_sig t,14_sig t,13_sig t,12_sig t,11_sig t,10_sig t),
                    V2(f1_sig t,f0_sig t))",
   REWRITE_TAC[th4]
);;

%----------------------------------------------------------------------------%
% To use COUNTLOGIC_PCORRECT to prove COUNTER_PCORRECT, we need the following%
% instance, which requires more sophisticated matching than rewriting does.  %
% (Clearly, a tactic could be written to do this.)                           %
%                                                                            %
% COUNTLOGIC_PCORRECT' =                                                     %
% |- (!t.                                                                   %
%      V6                                                                    %
%      (c5_sig(t + 1),c4_sig(t + 1),c3_sig(t + 1),c2_sig(t + 1),             %
%       c1_sig(t + 1),c0_sig(t + 1)),double_sig(t + 1),                      %
%      V2(n1_sig(t + 1),n0_sig(t + 1)) =                                     %
%      COUNTLOGIC                                                            %
%      ((V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),          %
%        double_sig t,V2(n1_sig t,n0_sig t)),                                %
%       V6(15_sig t,14_sig t,13_sig t,12_sig t,11_sig t,10_sig t),           %
```

```
%       V2(f1_sig t,f0_sig t))) /\                                              %
%    (NODE                                                                      %
%     (V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),               %
%      double_sig t,V2(n1_sig t,n0_sig t)) =                                    %
%     #00) /\                                                                   %
%    NEXTTIME                                                                   %
%    (t,t')                                                                     %
%    (\x.                                                                       %
%      NODE                                                                     %
%      (V6(c5_sig x,c4_sig x,c3_sig x,c2_sig x,c1_sig x,c0_sig x),              %
%       double_sig x,V2(n1_sig x,n0_sig x)) =                                   %
%      #00) ==>                                                                 %
%    (COUNT                                                                     %
%     (V6(c5_sig t',c4_sig t',c3_sig t',c2_sig t',c1_sig t',c0_sig t'),         %
%      double_sig t',V2(n1_sig t',n0_sig t')) =                                %
%     COUNTER                                                                   %
%     (COUNT                                                                    %
%      (V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),              %
%       double_sig t,V2(n1_sig t,n0_sig t)),                                    %
%      V6                                                                       %
%      (l5_sig(t + 1),l4_sig(t + 1),l3_sig(t + 1),l2_sig(t + 1),                %
%       l1_sig(t + 1),l0_sig(t + 1)),V2(f1_sig t,f0_sig t)))                    %
%-------------------------------------------------------------------------------%

let COUNTLOGIC_PCORRECT' = CONV_RULE(DEPTH_CONV BETA_CONV)
                           (INST ["\t:num.(V6(c5_sig t,c4_sig t,c3_sig t,
                                             c2_sig t,c1_sig t,c0_sig t),
                                          (double_sig t):bool,
                                          V2(n1_sig t,n0_sig t))",
                                  "state_sig:num -> (word6 # bool # word2)";
                                  "\t:num. V6(l5_sig t,l4_sig t,l3_sig t,
                                              l2_sig t,l1_sig t,l0_sig t)",
                                  "loadin_sig:num -> word6";
                                  "\t:num. V2(f1_sig t,f0_sig t)",
                                  "func_sig:num -> word2"]
                                 COUNTLOGIC_PCORRECT);;

%-------------------------------------------------------------------------------%
% We can now use COUNTLOGIC_DEF_LEMMA to substitute into                        %
% COUNTLOGIC_PCORRECT':                                                         %
%-------------------------------------------------------------------------------%

let th5 = REWRITE_RULE [SYM COUNTLOGIC_DEF_LEMMA;COUNT_DEF;NODE_DEF]
                       COUNTLOGIC_PCORRECT';;

%-------------------------------------------------------------------------------%
% The preliminary correctness statement is in terms of tuples of objects  of %
% type ":num -> bool", i.e. functions from time to booleans.                    %
% This IS the preliminary correctness statement, so we do a nominal tactical %
% proof:                                                                        %
%-------------------------------------------------------------------------------%

let TUPLE_COUNTER_PCORRECT =
 prove_thm
  (`TUPLE_COUNTER_PCORRECT`,
   "CIRCUIT_IMP
    (((c5_sig,c4_sig,c3_sig,c2_sig,c1_sig,c0_sig),
       double_sig,
       (n1_sig,n0_sig)),
     (l5_sig,l4_sig,l3_sig,l2_sig,l1_sig,l0_sig),
     (f1_sig,f0_sig))                    /\
   (V2(n1_sig t,n0_sig t) = #00)         /\
   (NEXTTIME (t,t') (\x. V2(n1_sig x,n0_sig x)  = #00))
   ==>
   (V6(c5_sig t',c4_sig t',c3_sig t',c2_sig t',c1_sig t',c0_sig t') =
     COUNTER(V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),
             V6(l5_sig(t+1),l4_sig(t+1),l3_sig(t+1),
                l2_sig(t+1),l1_sig(t+1),l0_sig(t+1)),
             V2(f1_sig t,f0_sig t)))",
   REWRITE_TAC[th5]
);;

%-------------------------------------------------------------------------------%
% To put this correctness statement in terms of single variables rather than %
% tuples, we instantiate as follows -- a tactic could be written to do it:   %
%-------------------------------------------------------------------------------%

let th6 =            REWRITE_RULE [SYM VS2_DEF;SYM VS6_DEF]
                     (let c = "count_sigs:(num->bool)#(num->bool)#(num->bool)#
```

```
                                (num->bool)#(num->bool)#(num->bool)"
                    and l ="loadin_sigs:(num->bool)#(num->bool)#(num->bool)#
                                (num->bool)#(num->bool)#(num->bool)"
                    and n = "node_sigs: (num->bool)#(num->bool)"
                    and f = "func_sigs: (num->bool)#(num->bool)" in
                    INST["FST `c`,"c5_sig";
                         "FST(SND `c)","c4_sig";
                         "FST(SND(SND `c))","c3_sig";
                         "FST(SND(SND(SND `c)))","c2_sig";
                         "FST(SND(SND(SND(SND `c))))","c1_sig";
                         "SND(SND(SND(SND(SND `c))))","c0_sig";
                         "FST `n`,"n1_sig";
                         "SND `n`,"n0_sig";
                         "FST `l`,"l5_sig";
                         "FST(SND `l)","l4_sig";
                         "FST(SND(SND `l))","l3_sig";
                         "FST(SND(SND(SND `l)))","l2_sig";
                         "FST(SND(SND(SND(SND `l))))","l1_sig";
                         "SND(SND(SND(SND(SND `l))))","l0_sig";
                         "FST `f`,"f1_sig";
                         "SND `f`,"f0_sig"]
                        TUPLE_COUNTER_PCORRECT);;

%----------------------------------------------------------------------------%
% That IS the desired correctness result in the desired terms, so we do a    %
% nominal tactical proof again:                                              %
%----------------------------------------------------------------------------%

let COUNTER_PCORRECT =
 prove_thm
  (`COUNTER_PCORRECT`,
   "CIRCUIT_IMP((count_sigs,double_sig,node_sigs),loadin_sigs,func_sigs) /\
   (VS2 node_sigs t = #00)                                                  /\
   (NEXTTIME (t,t') (\x. VS2 node_sigs x  = #00))
   ==>
   (VS6 count_sigs t' =
     COUNTER(VS6 count_sigs t,
             VS6 loadin_sigs (t + 1),
             VS2 func_sigs t))",
   REWRITE_TAC[th6]
);;

%----------------------------------------------------------------------------%
% Now we must prove termination, and we are finished.                        %
% We proceed exactly as before, when using COUNTLOGIC_PCORRECT to get        %
% COUNTER_PCORRECT.                                                          %
% To use COUNTLOGIC_TERMINATES to prove COUNTER_TERMINATES, we need the      %
% following instance, which requires more sophisticated matching than        %
% rewriting provides:                                                        %
%                                                                            %
% COUNTLOGIC_TERMINATES' =                                                   %
% |- (!t.                                                                   %
%      V6                                                                    %
%      (c5_sig(t + 1),c4_sig(t + 1),c3_sig(t + 1),c2_sig(t + 1),             %
%       c1_sig(t + 1),c0_sig(t + 1)),double_sig(t + 1),                      %
%      V2(n1_sig(t + 1),n0_sig(t + 1)) =                                     %
%      COUNTLOGIC                                                            %
%      ((V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),          %
%        double_sig t,V2(n1_sig t,n0_sig t)),                                %
%       V6(l5_sig t,l4_sig t,l3_sig t,l2_sig t,l1_sig t,l0_sig t),           %
%       V2(f1_sig t,f0_sig t))) /\                                           %
%    (NODE                                                                   %
%     (V6(c5_sig t,c4_sig t,c3_sig t,c2_sig t,c1_sig t,c0_sig t),           %
%      double_sig t,V2(n1_sig t,n0_sig t)) =                                 %
%     #00) ==>                                                               %
%    (?t'.                                                                   %
%      NEXTTIME                                                              %
%      (t,t')                                                                %
%      (\x.                                                                  %
%        NODE                                                                %
%        (V6(c5_sig x,c4_sig x,c3_sig x,c2_sig x,c1_sig x,c0_sig x),         %
%         double_sig x,V2(n1_sig x,n0_sig x)) =                              %
%        #00))                                                               %
%----------------------------------------------------------------------------%

let COUNTLOGIC_TERMINATES' = CONV_RULE(DEPTH_CONV BETA_CONV)
                             (INST ["\t:num.(V6(c5_sig t,c4_sig t,c3_sig t,
                                                c2_sig t,c1_sig t,c0_sig t),
                                            (double_sig t):bool,
```

```
                                        V2(n1_sig t,n0_sig t))",
                                   "state_sig:num -> (word6 # bool # word2)";
                                   "\t:num. V6(l5_sig t,l4_sig t,l3_sig t,
                                              l2_sig t,l1_sig t,l0_sig t)",
                                   "loadin_sig:num -> word6";
                                   "\t:num. V2(f1_sig t,f0_sig t)",
                                   "func_sig:num -> word2"]
                                  COUNTLOGIC_TERMINATES);;

%----------------------------------------------------------------------------%
% We now use the lemma relating COUNTLOGIC to CIRCUIT_IMP, to substitute     %
% into COUNTLOGIC_TERMINATES', just as before:                               %
%----------------------------------------------------------------------------%

let th7 = REWRITE_RULE [SYM COUNTLOGIC_DEF_LEMMA;NODE_DEF]
                       COUNTLOGIC_TERMINATES';;

%----------------------------------------------------------------------------%
% This is the desired result so we do a tactical proof:                      %
%----------------------------------------------------------------------------%

let TUPLE_COUNTER_TERMINATES =
   prove_thm(`TUPLE_COUNTER_TERMINATES`,
             "CIRCUIT_IMP(((c5_sig,c4_sig,c3_sig,c2_sig,
                            c1_sig,c0_sig),double_sig,n1_sig,n0_sig),
                           (l5_sig,l4_sig,l3_sig,l2_sig,l1_sig,l0_sig),
                           f1_sig,f0_sig) /\
             (V2(n1_sig t,n0_sig t) = #00) ==>
             (?t'. NEXTTIME(t,t')(\x. V2(n1_sig x,n0_sig x) = #00))",
             REWRITE_TAC[th7]);;

%----------------------------------------------------------------------------%
% We now instantiate, to cope with non-tuple terms:                          %
%----------------------------------------------------------------------------%

let th8 =             REWRITE_RULE [SYM VS2_DEF;SYM VS6_DEF]
                      (let c = "count_sigs:(num->bool)#(num->bool)#(num->bool)#
                                           (num->bool)#(num->bool)#(num->bool)"
                       and l ="loadin_sigs:(num->bool)#(num->bool)#(num->bool)#
                                           (num->bool)#(num->bool)#(num->bool)"
                       and n = "node_sigs: (num->bool)#(num->bool)"
                       and f = "func_sigs: (num->bool)#(num->bool)" in
                       INST["FST `c`","c5_sig";
                            "FST(SND `c)","c4_sig";
                            "FST(SND(SND `c))","c3_sig";
                            "FST(SND(SND(SND `c)))","c2_sig";
                            "FST(SND(SND(SND(SND `c))))","c1_sig";
                            "SND(SND(SND(SND(SND `c))))","c0_sig";
                            "FST `n`","n1_sig";
                            "SND `n`","n0_sig";
                            "FST `l`","l5_sig";
                            "FST(SND `l)","l4_sig";
                            "FST(SND(SND `l))","l3_sig";
                            "FST(SND(SND(SND `l)))","l2_sig";
                            "FST(SND(SND(SND(SND `l))))","l1_sig";
                            "SND(SND(SND(SND(SND `l))))","l0_sig";
                            "FST `f`","f1_sig";
                            "SND `f`","f0_sig"]
                           TUPLE_COUNTER_TERMINATES);;

%----------------------------------------------------------------------------%
% Again, this is the desired result, so we do a tactical proof:              %
%----------------------------------------------------------------------------%

let COUNTER_TERMINATES =
 prove_thm
  (`COUNTER_TERMINATES`,
   "CIRCUIT_IMP((count_sigs,double_sig,node_sigs),loadin_sigs,func_sigs) /\
   (VS2 node_sigs t = #00)
   ==>
   (?t'. NEXTTIME(t,t')(\x. VS2 node_sigs x = #00))",
   REWRITE_TAC[th8]
);;

%----------------------------------------------------------------------------%
% Combining COUNTER_TERMINATES and COUNTER_PCORRECT we can prove the         %
% theorem COUNTER_CORRECT below which expresses the total correctness of the %
% COUNTER. This parallels the total correctness proof of the host, on        %
% HOST_CORRECT.th.                                                           %
```

```
%----------------------------------------------------------------------------%

%----------------------------------------------------------------------------%
% The derived inference rule EXISTS_AND_RULE is useful for combining         %
% termination and partial correctness to get total correctness.              %
%                                                                            %
% EXISTS_AND_RULE : (thm # thm) -> thm                                       %
%                                                                            %
%            A1 |- ?x. tm[x]        A2 |- !x. tm[x] ==> tm'[x]              %
%            ------------------------------------------------                %
%                     A1 u A2 |- ?x. tm[x] /\ tm'[x]                         %
%                                                                            %
%                                                                            %
%----------------------------------------------------------------------------%

let EXISTS_AND_RULE(th1,th2) =
 let x,t1 = dest_exists(concl th1)
 and t2   = snd(dest_imp(snd(dest_forall(concl th2))))
 in
 let etm = mk_select(x,t1)
 in
 let th3 = SELECT_RULE th1
 in
 let th4 = CONJ th3 (MP(SPEC etm th2)th3)
 in
 EXISTS ("?'x.'t1/\'t2",etm) th4;;

%----------------------------------------------------------------------------%
% By forward proof we can now deduce:                                        %
%                                                                            %
%   COUNTER_CORRECT_LEMMA =                                                  %
% .. |- ?t'.                                                                 %
%        NEXTTIME(t,t')(\x. VS2 node_sigs x = #00) /\                        %
%       (VS6 count_sigs t' =                                                 %
%        COUNTER(VS6 count_sigs t,VS6 loadin_sigs(t + 1),VS2 func_sigs t))   %
%----------------------------------------------------------------------------%

let COUNTER_CORRECT_LEMMA =
 EXISTS_AND_RULE
  (UNDISCH_ALL(hd(IMP_CANON COUNTER_TERMINATES)),
   GEN_ALL(UNDISCH(UNDISCH(hd(IMP_CANON COUNTER_PCORRECT)))));;


let COUNTER_CORRECT =
 prove_thm
  (`COUNTER_CORRECT`,
   "CIRCUIT_IMP((count_sigs,double_sig,node_sigs),loadin_sigs,func_sigs) /\
    (VS2 node_sigs t = #00)
    ==>
    ?t'. NEXTTIME (t,t') (\x. VS2 node_sigs x =#00) /\
         (VS6 count_sigs t' =
          COUNTER(VS6 count_sigs t,
                  VS6 loadin_sigs (t + 1),
                  VS2 func_sigs t))",
   REPEAT STRIP_TAC
   THEN IMP_RES_TAC(DISCH_ALL COUNTER_CORRECT_LEMMA));;


%============================================================================%
%                              Q.E.D.                                        %
% That completes the overall correctness proof of the counter.              %
%============================================================================%

close_theory();;
```