Technical Report

UCAM-CL-TR-124
ISSN 1476-2986

Number 124

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Formal verification of basic memory devices

John Herbert

February 1988

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

Formal Verification of Basic Memory Devices

John Herbert
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG

Abstract: Formal methods have been used recently to verify high-level functional specifications of digital systems. Such formal proofs have used simple models of circuit components. In this article we describe complementary work which uses a more detailed model of components and demonstrates how hardware can be specified and verified at this level.

In this model all circuits can be described as structures of gates, each gate having an independent propagation delay. The behaviour of digital signals in real time is captured closely. The function and timing of asynchronous and synchronous memory elements implemented using gates is derived. Formal proofs of correctness show that, subject to certain constraints on gate delays and signal timing parameters, these devices act as memory elements and exhibit certain timing properties.

All the proofs have been mechanically generated using Gordon's HOL system.

Preface

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

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

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

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

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

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

Contents

1 Introduction

Conventional design tools deal mostly with behaviour modelled at the register transfer level. However it is also necessary to check the timing of the implementation to determine the maximum speed and find timing bugs and critical paths. Timing analysis programs are used for this purpose in a conventional CAD system.

Formal methods are usually applied to the register transfer behavioural level and the more detailed timing of a design is ignored. However timing is an important part of a design, especially in integrated circuits where speed is often a major reason for integration. We have developed formal techniques for reasoning about circuit behaviour at a detailed timing level. In this article we demonstrate the basic techniques and use them to verify the function and timing of simple asynchronous and synchronous memory devices. (Another article [Herbert88b] deals with the problem of verifying efficiently the function and timing of complete designs.)

All specifications and proofs of correctness are mechanically generated using the HOL system [Gordon85a] [Gordon85b]. The HOL language and system are not described in this article; full descriptions are available in the references just cited.

2 Primitive Device Models

Large scale synchronous digital designs have been successfully specified and verified using the HOL system [Camilleri85] [Cohn87] [Herbert88a]. The proofs are based on simple models for the primitive components (combinational devices such as gates and memory elements such as flip-flops). The combinational devices have no delay and memory elements introduce unit delay. This behavioural level is usually called the register transfer level but we will also refer to this level as the *synchronous level* to avoid any unintended association with register transfer languages. The synchronous level is characterised by a time scale where the basic units correspond to the period of an implicit synchronous clock.

The simple models which ignore timing provide a tractable basis for the verification of high-level functional specifications. However the behaviour of the physical component depends on lower level timing constraints. For example, real memory elements do not work if the clock period is too short or the data signal is not stable long enough for the device to store its value. We introduce a more detailed model of components which captures more closely the behaviour of the physical device.

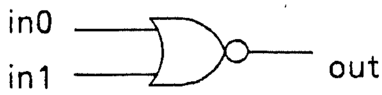

Figure 1: NOR gate

2.1 Primitive Component Model

A new model is introduced where the primitive components are propagation delay gates and signal behaviour over real time is modelled closely. The only primitive components in this model are gates with arbitrary, fixed propagation delay. Other components, such as asynchronous and synchronous memory elements, are built from gates and, in turn, larger circuits can be constructed from these components.

2.1.1 Example: NOR gate

The behaviour of a two input NOR gate (Figure 1) modelled as a propagation delay device is:

```
∀t. out (t + del) = ¬ (in0 t ∨ in1 t)
```

This may be paraphrased as:

> for all times, the output `del` units after a time is the *nor* of the inputs at that time.

A gate takes a fixed time `del` to compute its output, and each individual gate may have a unique delay.

The time scale for this model is a representation of real time. For example, the propagation delay `del` might be 55, representing 5.5 nanoseconds if the units of the time scale correspond to 0.1 nanoseconds of real time. We describe the level of behaviour in the delay model of digital devices as the *timing level*. The *timing level* is characterised by a fine time scale where the basic units can be chosen to represent any interval of real time. (In contrast, the time scale at synchronous level is derived from an implicit clock and the time units correspond to the clock period in real time.)

3 Reasoning at the Timing Level

We describe in this section some of the methods we devised to reason in HOL about digital systems at a detailed timing level.

3.1 Describing Timing Behaviour in HOL

We are concerned with describing and reasoning about timing behaviour. The value of a signal changes in time and some statements about a device may be true at some times and false at others. The natural numbers (of type `num`) are used to represent time in HOL. Statements whose truth-value depends on time can be expressed in HOL as functions of type `num` $\longrightarrow$ `bool`. We model digital signal values as booleans, and signals as functions from time to boolean (*i.e.* of type `num` $\longrightarrow$ `bool`). The natural numbers can represent multiples of the smallest significant unit of real time. This provides a granular time-scale of sufficient precision for the user. For example, if $0.001ns$ is regarded as the smallest significant unit of real time then the number 500 can represent $0.5ns$.

As well as instants of time we also need to reason about intervals of time. We choose to describe behaviour over half-open intervals. The expression

$$\forall t.\ (t1 < t) \wedge (t \leq t2) \Longrightarrow P\ t$$

states that the predicate `P` is true over the half-open interval `(t1, t2]`.

3.2 New higher-order functions

Two new higher-order functions, `ALWAYS` and `DURING`, provide a simple basis for descriptions of timing level behaviour. We deduce theorems involving these higher-order functions which can be used in many proofs.

These are defined as:

$$\text{ALWAYS } P = \forall t.P\ t$$

$$\text{DURING } (t1,\ t2)\ P = \forall t.(t1 < t) \wedge (t \leq t2) \Longrightarrow P\ t$$

`ALWAYS P` simply states that at all times, t, the term `P t` is true.
`DURING (t1, t2) P` states that `P t` is true for all times, t, in the half-open interval `(t1, t2]`.

The type of `P` is the same in both definitions and so similar expressions can correspond to `P` in both definitions. The lambda abstraction mechanism allows arbitrary expressions involving a variable which corresponds to time to be rewritten in the form `P t`.

As stated previously, the behaviour of a NOR gate can be described as:

$$\forall t.\ \ \texttt{out (t + del)} = \neg\texttt{(in0 t} \vee \texttt{in1 t)}$$

This can be expressed using the higher-order function ALWAYS as:

$$\texttt{ALWAYS} (\lambda t.\ \ \texttt{out (t + del)} = \neg\texttt{(in0 t} \vee \texttt{in1 t))}$$

If a signal out is low from time t1 to t2 this can be described as:

$$\forall t.\ \ (t1 < t) \wedge (t \leq t2) \Longrightarrow (\texttt{out t} = F)$$

Using the higher-order function DURING this can be expressed as:

$$\texttt{DURING (t1, t2)} (\lambda t.\ \ \texttt{out t} = F)$$

All descriptions of timing behaviour use the higher-order functions ALWAYS and DURING rather than standard logical form. Special techniques are developed to manipulate terms involving these higher-order functions.

3.3 Special Theorems and Rules

We develop a set of theories on which to base the proofs at the timing level. For example, we have a theory MAX_MIN in which functions MAX and MIN are defined, and theorems stating useful properties of these functions are proved. The theory named DURING contains a number of theorems which allow us to reason about intervals.

Some of these theorems are:

```
simp_extend:
   DURING (t1, t2) P ∧ DURING (t2, t3) P
            ⟹ DURING (t1, t3) P
extend:
   DURING (t1, t2) P ∧ DURING (t3, t4) P ∧ (t3 ≤ t2)
            ⟹ DURING (t1, t4) P
extend_or:
   DURING (t1, t2) P0 ∧ DURING (t3, t4) P1 ∧ (t3 ≤ t2)
            ⟹ DURING (t1, t4) (λt. P0 t ∨ P1 t)
overlap:
   DURING (t1, t2) P0 ∧ DURING (t3, t4) P1
            ⟹ DURING ( MAX(t1,t3), MIN(t2,t4) ) (λt. P0 t ∧ P1 t)
narrow:
   DURING (t1, t2) P ∧  (t1 ≤ t3)  ∧ (t4 ≤ t2)
            ⟹ DURING (t3, t4) P
```

We construct a number of rules which use these derived theorems. For example, OVERLAP_RULE (of type thm ⟶ thm ⟶ thm) can be applied to two theorems

whose conclusions are expressions of the form DURING (t1,t2) P.
This rule

- forms the conjunction of the theorems,
- deduces a new theorem by using theorem overlap, and
- simplifies the final theorem using β-conversion.

Example:

```
thm_1:  DURING(t1,t2)(λt. in0 t = F)
thm_2:  DURING(t3,t4)(λt. out(t + del) = ¬ in0 t)

OVERLAP_RULE  thm_1  thm_2

yields:

 DURING(MAX(t1,t3),MIN(t2,t4))(λt. (out(t + del) = ¬ in0 t) ∧ (in0 t = F))
```

These special purpose inference rules allow us to reason about timing at the level of the higher-order functions ALWAYS and DURING. We do not need to expand ALWAYS and DURING into more standard logical form. This eliminates much tedious manipulation. The rules make the proof efficient — a standard theorem is used by matching rather than repetition of inference steps.

4 An Asynchronous Latch

A model of component delay has been introduced. The behaviour of a simple RS latch is now deduced using this model. The latch is a good example because its correct operation depends on gate propagation delays and the timing of its input signals. A detailed timing model is therefore required to deduce its behaviour accurately. The latch forms a sub-structure of synchronous memory elements built from gates and so its behaviour can be used in proofs of synchronous memory elements.

4.1 NOR Gate behaviour

A latch is built using NOR gates. The model of a NOR gate at a detailed timing level is that of a propagation delay device computing the *nor* function of its inputs. Using the higher-order function ALWAYS, the behaviour can be defined as:

```
NOR2 (in0, in1, del, out) =
```

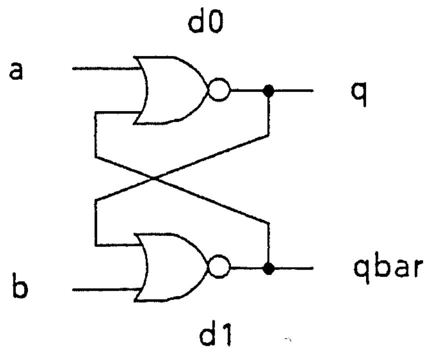

Figure 2: Latch

```
ALWAYS  (λ t. out(t + del) = ¬ (in0 t ∨ in1 t))
```

This definition may be read as follows:

> The predicate NOR2 is true of signals in0,in1,out and delay del if and only if the output out at time t+del is always equal to the *nor* of the inputs in0 and in1 at time t.

4.2 Definition of Latch

A latch is built from two primitive NOR gates as depicted in Figure 2.
The predicate LATCH, defined as

```
LATCH (a, b, q, qbar, d0, d1) =
            (NOR2 (a, qbar, d0, q) ∧ NOR2 (b, q, d1, qbar))
```

describes a latch consisting of cross-coupled NOR gates of delay d0 and d1.

A latch can exhibit different types of behaviour depending on its environment. For example, a narrow input pulse may result in some oscillation on the latch outputs. The behaviour that interests us is when the latch acts as a simple memory element. To permit this desired behaviour the input signals must satisfy constraints which depend on the latch gate delays. Before beginning the formal description and proof of the latch, we form a statement of the required behaviour for the device. The behaviour we wish to demonstrate is:

> If some data and its inverse are presented on the two latch inputs for a certain length of time and both inputs are then low until the next data is presented, then the data and its inverse are available on the latch outputs from a time after the data was presented to the latch until some time after the next data is presented.

The latch acts as a memory because the data is available on its outputs for an indefinite period after the data becomes unavailable on the inputs and while new data has not yet been presented. Note that we regard a low signal as corresponding to the absence of data. We refer to data without stating the particular data value. We could have formed a more detailed description of behaviour which mentioned the data value. The data independent statement of behaviour provides a simpler basis for proofs which use the latch result.

4.3 Proof of Latch Behaviour

The proof of the latch behaviour is chosen to illustrate the methods used when reasoning about behaviour at the detailed timing level. In Appendix 1 we give the full commented code for this proof in HOL.

4.3.1 Derived Rules

In the code listing we describe briefly the special rules used in the proof. Two rules used widely in the proof are PROPAGATE and SHIFT_RULE.

The function PROPAGATE is of type (thm ⟶ thm ⟶ thm); it is applied to two theorems. The first theorem has a conclusion which is a DURING expression, the conclusion of the second is a DURING or ALWAYS expression. The first theorem asserts that a signal (the source signal) has some value over an interval; the second theorem relates this signal to another (the destination signal). PROPAGATE deduces a new theorem asserting the value of the destination signal over some interval.

For example,

```
Given
thm_1:    DURING(t1,t2)(λt. in0 t = F)
thm_2:    DURING(t3,t4)(λt. out(t + del) = ¬in0 t)
thm_3:    ALWAYS(λt. out(t + del) = ¬in0 t)

              PROPAGATE  thm_1  thm_2

yields:   DURING(MAX(t1,t3),MIN(t2,t4))(λt. out(t + del))

              PROPAGATE  thm_1  thm_3

yields:   DURING(t1,t2)(λt. out(t + del))
```

The function SHIFT_RULE is of type (thm ⟶ thm). It can be used to "shift" the interval of a DURING expression which forms the conclusion of the theorem. This rule is often used after PROPAGATE to deal with the delay introduced by a gate.

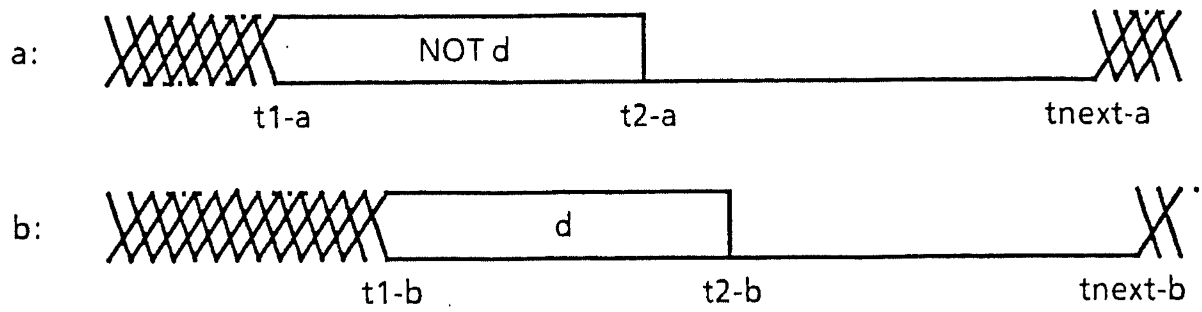

Figure 3: Latch input waveforms

For example,

```
Given
thm_1:    DURING(t1,t2)(λt. out(t + del))

              SHIFT_RULE thm_1

yields:   DURING(t1 + del,t2 + del)(λt. out t)
```

4.3.2 Description of Proof

We now describe the steps taken in the proof of the latch behaviour. (The code for the proof is given in Appendix 1.)

Starting Assumption

We begin by assuming that a predicate LATCH is true for some signals and delays, that data and its inverse are presented on the two latch inputs for some duration, and that both inputs revert to low afterwards.

This assumption is formally stated by the conjunction:

```
LATCH (a, b, q, qbar, d0, d1) ∧
DATA_AVAILABLE (a, (¬ d), t1_a, t2_a, tnext_a) ∧
DATA_AVAILABLE (b, d, t1_b, t2_b, tnext_b)
```

The predicate DATA_AVAILABLE is defined as:

```
DATA_AVAILABLE (signal, data, t1, t2, tnext) =
                        (DURING (t1, t2)    (λt. signal t = data) ∧
                         DURING (t2, tnext) (λt. signal t = F) )
```

(*i.e.* data is available on the line from t1 to t2 and absent from t2 to tnext)
The input waveforms are represented in Figure 3.

Propagation of Input Signals

From the definition of the latch structure and the behaviours of its component NOR gates, the relationships between the inputs and outputs can be deduced. The assumed input waveforms can be propagated to the outputs using these relationships. We do not assume any particular value for the data presented on the latch inputs, just that some data value d and its inverse ¬ d are presented. In practice, we prove the behaviour for the two possible data values T and F and then combine the results. This does not double the proof; the symmetry of the latch allows us to transform the result for one data value into the result for the other.

We assume d = F and propagate the input signal a through the top gate to output q. Since q feeds the other NOR gate, we propagate the resultant signal through this gate. As we propagate through the lower gate the signal is combined with the input signal b. Since the timing parameters of the two external signals are independent, the resultant signal depends on the relative values of these parameters. The behaviour of qbar is deduced as:

```
DURING
  (MAX(t1_a + d0,t1_b),MIN(t2_a + d0,tnext_b))
  (λ t. qbar(t + d1))
```

The comparison of timing parameters results in a number of branches in the proof. These branches correspond to different orderings of the timing parameters and are eventually combined. There is a single main proof path and the other branches follow trivial orderings. For the above expression, the main proof branch follows the condition (t2_a + d0) ≤ tnext_b. Our main concern is to show that data is stored in the latch for an arbitrary long interval until new data arrives. The alternative branch, where tnext_b < (t2_a + d0), follows a trivial case when new data arrives at input b very shortly after data disappears from input a.

Latching of Data

Having propagated the input data through the latch components, the next concern is to demonstrate that the data is stored in the latch when the inputs revert to the no-data value (F). We call the condition that ensures that input data is latched the *latching condition.* Figure 4 presents some waveforms which illustrate the latching condition.

The input a presents a value T from time t1_a to t2_a and this forces signal q to have value F. After t2_a, signal a is low and so from that time the signal qbar determines q. We require that from t2_a the signal qbar should present T and thus

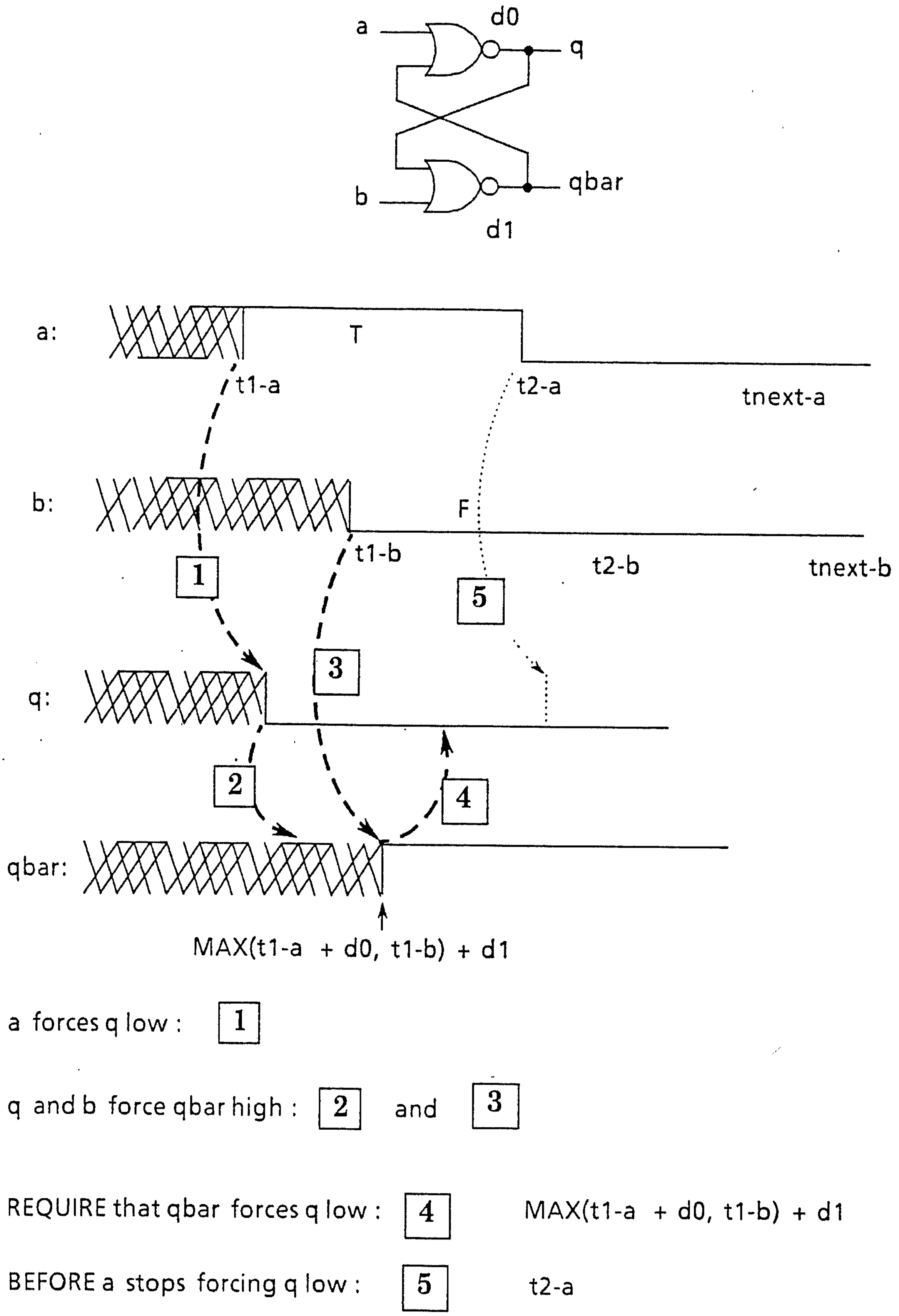

Figure 4: Latching of data

maintain q with value F. We have deduced (by propagation) that qbar goes high initially at time MAX(t1_a + d0,t1_b)+ d1. The condition that the data is latched is therefore:

$$(\text{MAX}(t1\_a + d0, t1\_b) + d1) \leq t2\_a$$

In terms of our formal proof, we can prove that q is high over a certain interval due to input a and also high during another interval due to propagation via signal qbar. To deduce that q retains its value throughout a longer interval including both of these we must prove that the intervals overlap. To do this we need to add the assumption that the lower limit of the second interval is less than or equal to the upper limit of the first interval. This corresponds exactly to the condition mentioned above.

Induction

By introducing the condition for latching of data we have deduced that data is initially latched, *i.e.* that the outputs are maintained immediately after the forcing input signal reverts to low. We need to prove that the output signals are maintained for an arbitrary length of time until new data appears at one of the external inputs; this is done by induction.

We form a suitable statement of the desired behaviour involving the variable n which will be the induction variable. We prove that this statement is true for all n by proving the base case (n = 0) and the step case (if it is true for n then it is true for n+1).

The base case can be proved immediately by the theorem stating the initial latching of data. Proving the step case again requires the latching condition and also necessitates the introduction of a new condition, namely that the gate delay d0 is non-zero. (If we look again at the theorem stating the initial latching of data we see that the upper limit of the deduced interval is greater than t2_a only when d0 or d1 is non-zero. So data is maintained after t2_a only when at least one of the gate delays is non-zero.)

The introduction of a requirement that the latch delays are non-zero is not surprising. Delay is necessary for memory. The need for capacitance, which introduces delay, to allow cross-coupled circuits to store information is discussed in [Seitz80].

The theorem proven by induction is an implication; we specialise n to the maximum value which keeps the antecedent true. This value asserts the longest interval of stability for signal q.

We have proved the desired behaviour for signal q following the main proof branch. We now deduce similar behaviour for the trivial proof branches and combine the theorems of behaviour, eliminating the conditions associated with the branches.

We can deduce the behaviour of signal qbar in a straightforward way, using propagation, from the behaviour deduced for q.

Using Latch Symmetry

We have derived a theorem stating the behaviour of q and qbar for the case when d = F. The symmetry of the latch allows us to prove the d = T case directly from this theorem. To do this we interchange signals a and b, the timing parameters for a and b, signals q and qbar and delays d0 and d1. We also replace d by its inverse ¬ d. The theorem deduced in this way for d = T needs a little manipulation to get it into a form similar to that of the theorem for d = F. For example, we must transform LATCH(b,a,qbar,q,d1,d0) into LATCH(a,b,q,qbar,d0,d1).

We must also prepare both theorems so that they can be easily combined to deduce the behaviour for an unknown value of d. The latching conditions for d high and low are, respectively:

```
    (MAX(t1_b + d1,t1_a)+ d0) ≤ t2_b
and (MAX(t1_a + d0,t1_b)+ d1) ≤ t2_a
```

We devise a more general condition which ensures that either a high or low data value is latched.
This latching condition is:

```
((MAX(t1_a,t1_b)+ d0)+ d1) ≤ MIN(t2_a,t2_b)
```

Deduced Output Behaviour

The behaviour of the outputs for any data value of d is deduced by combining the results deduced for d = T and d = F. The deduced behaviour of q and qbar is:

```
DURING
 (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d0)
 (λ t. q t = d) ∧
 DURING
 (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d1)
 (λ t. qbar t = ¬ d)
```

The outputs q and qbar present the data and its inverse over certain intervals. The theorem of behaviour for the latch is described in detail in the next section.

4.3.3 Deduced Behaviour of Latch

We arrange the theorem of behaviour for the latch into the following form:

```
delay_and_timing_conditions ⟹
          latch_implementation ⟹
                input_output_behaviour
```

This makes clear that under certain delay and timing conditions, the implementation of the latch achieves the desired input-output behaviour. In this form the theorem of behaviour is:

```
0 < d0 ∧
0 < d1 ∧
(((MAX(t1_a,t1_b)) + d0) + d1) ≤ MIN(t2_a,t2_b) ⟹

   LATCH(a,b,q,qbar,d0,d1) ⟹

     (DATA_AVAILABLE(a,(¬ d),t1_a,t2_a,tnext_a) ∧
      DATA_AVAILABLE(b,d,t1_b,t2_b,tnext_b) ⟹
       DURING
        (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d0)
        (λ t. q t = d) ∧
       DURING
        (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d1)
        (λ t. qbar t = ¬ d))
```

The delay and timing conditions are:

```
0 < d0 ∧
0 < d1 ∧
(((MAX(t1_a,t1_b)) + d0) + d1) ≤ MIN(t2_a,t2_b)
```

The gate delays must be non-zero and the constraint which ensures that the input data gets latched (the latching condition) must be satisfied.

The latch implementation is described by:

```
LATCH(a,b,q,qbar,d0,d1)
```

The input-output behaviour is:

```
DATA_AVAILABLE(a,(¬ d),t1_a,t2_a,tnext_a) ∧
DATA_AVAILABLE(b,d,t1_b,t2_b,tnext_b) ⟹
   DURING
     (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d0)
     (λ t. q t = d) ∧
   DURING
     (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d1)
     (λ t. qbar t = ¬ d)
```

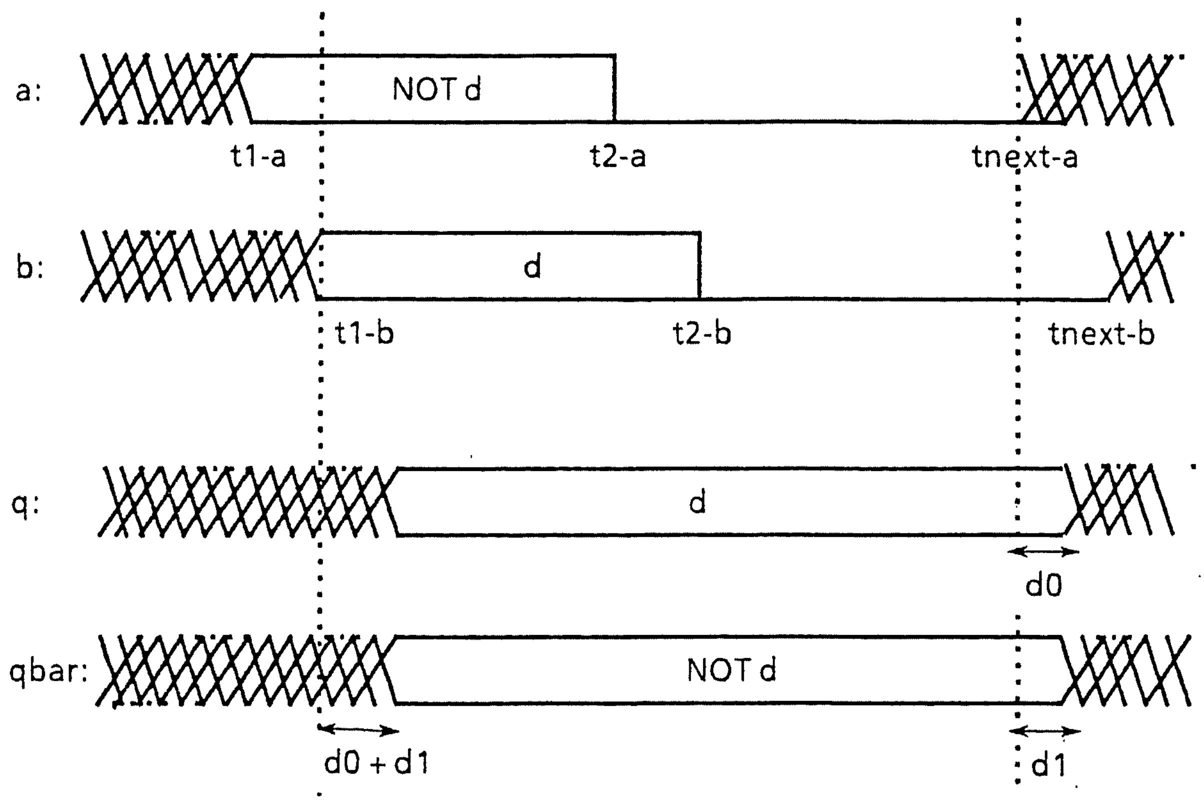

Figure 5: Input-output behaviour of the latch

Figure 5 depicts waveforms which meet the input-output behaviour. Data and its inverse must be presented on the input signals a and b. If this is true and the other conditions hold, we can deduce that the output behaviour is achieved. The data and its inverse are available on the outputs q and qbar over an interval starting d0+d1 after MAX(t1_a,t1_b), the first time when both inputs are presented simultaneously with suitable data, and finishing at, respectively, d0 and d1 time units after MIN(tnext_a,tnext_b), the earliest time when one of the inputs receives some new data.

We can simplify the expression describing the input-output behaviour to get an interval when both q and qbar are stable.
This is:

```
DATA_AVAILABLE(a,(¬ d),t1_a,t2_a,tnext_a) ∧
DATA_AVAILABLE(b,d,t1_b,t2_b,tnext_b) ⟹
  DURING
    (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)))
    (λ t. (q t = d) ∧ (qbar t = ¬ d))
```

This simpler statement of output behaviour is used in the proofs of some higher-level components.

4.3.4 Behaviour as a Memory Element

The latch is described as displaying the behaviour of a memory element because under certain conditions it retains data for an arbitrary length of time after that data is no longer available on any of its inputs. If the latching condition holds for the input signals a and b then the outputs q and qbar retain the latched data values after both inputs have lost the data value and gone low. The outputs continue to retain their values until after the next input change occurs. This is true irrespective of how long it is until the next change.

The latch is an asynchronous memory element because it is sensitive at all times to its inputs, and the stored value can thus be changed at any time.

The behaviour of the latch as a memory element depends on propagation delay and the precise timing of its input signals. We have shown that we can model these concerns in HOL and can deduce this behaviour. Timing constraints have been introduced in the course of proving the correct behaviour.

5 Master-Slave Flip-flop

In the previous section it was established that an RS latch can provide a simple asynchronous memory element. A latch is not sufficient for most applications involving memory; for example, a shift register with a single latch per bit will not work. Edge-triggered, synchronous flip-flops are commonly used memory elements. The behaviour of a positive-edge triggered master-slave flip-flop is now derived.

The master-slave flip-flop is implemented using propagation delay NOR gates and has clock and data inputs and data and inverse data outputs (Figure 6).

5.1 Specification

An informal description of desired behaviour is:

> If the clock signal rises and the data input signal has been stable over an interval of time around the clock rise, then the input data and its inverse will be available on the master-slave outputs some time after the clock rise and will persist on the outputs until some time after the next rising edge.

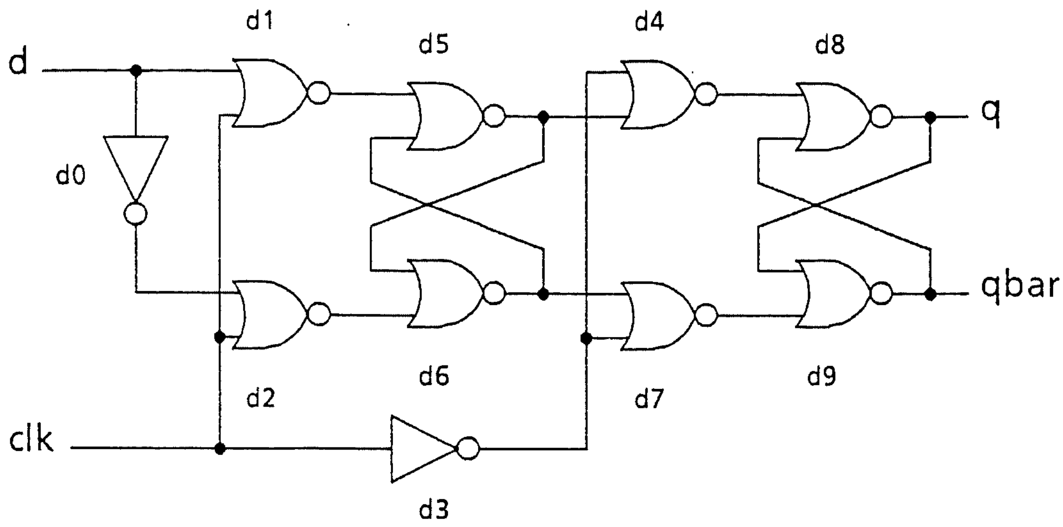

Figure 6: Master-Slave Structure

We do not present a formal specification of the desired behaviour of the flip-flop because we do not devise one before we do the proof. The timing conditions which form part of the behavioural specification of the master-slave are not known and therefore a precise specification is not possible. These conditions are introduced as the proof progresses. The proof of behaviour is a forward one which proceeds from an assumption about the master-slave structure and the form of the input signals. Although the informal specification guides the creation of the proof, it would be misleading to present a formal specification as a starting point.

5.2 Implementation

A master-slave flip-flop is built from NOR gates and inverters (cf Figure 6). The inverter is a primitive component whose behaviour is described by the predicate INV, defined as follows:

```
INV(in, del, out) = ALWAYS(λ t.out(t + del) = ¬ in t)
```

There are two large sub-blocks (the master and the slave) which comprise four gates each. Within each sub-block two gates form a latch and the other two gates are used to control the latch inputs. We define a new predicate GATED_2NOR to describe a structure of two NOR gates which share a common gating signal:

```
GATED_2NOR(a,b,clk,ga,gb,d0,d1) = NOR2(a,clk,d0,ga) ∧ NOR2(b,clk,d1,gb)
```

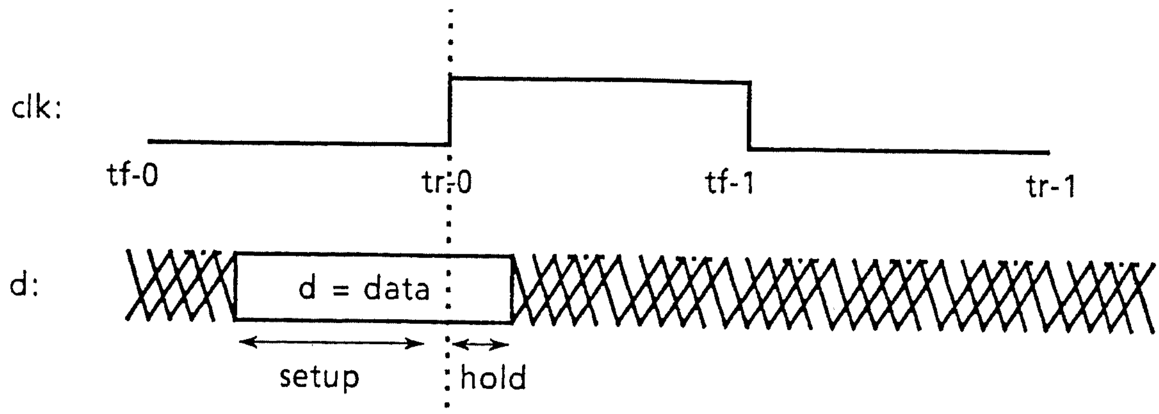

Figure 7: Timing Diagram Example

5.3 Proof of Behaviour

The proof is a forward one which proceeds from an initial assumption to ultimately derive the desired behaviour of the flip-flop. As the proof progresses certain conditions are introduced.

5.3.1 Starting Assumption

The assumption on which we base the proof of behaviour is:

```
INV(d,dbar,d0) ∧
INV(clk,clkbar,d3) ∧
GATED_2NOR(d,dbar,clk,a_0,b_0,d1,d2) ∧
LATCH(a_0,b_0,qa,qb,d5,d6) ∧
GATED_2NOR(qa,qb,clkbar,a_1,b_1,d4,d7) ∧
LATCH(a_1,b_1,q,qbar,d8,d9) ∧
DURING(tf_0, tr_0)(λt. clk t = F) ∧
DURING(tr_0, tf_1)(λt. clk t = T) ∧
DURING(tf_1, tr_1)(λt. clk t = F) ∧
DURING(tr_0 - setup, tr_0 + hold)(λt. d t = data)
```

The first six conjuncts are the predicates which correspond to the master-slave components; the final four describe the assumed behaviour of the clock and data signals. The timing diagram (Figure 7) presents typical clock and data waveforms.

The times tf_0, tf_1, tr_0 and tr_1 correspond to what we would usually think of as times of falling and rising edges. Although the timing diagram conveys accurately our intuitive ideas about the waveforms, it also contains implicit assumptions about relationships between the various instants of time. No ordering of timing parameters is assumed in the formal description of the signals.

The parameters *setup* and *hold* are the lengths of time a data signal must be stable before and after an active clock edge to ensure correct operation of a memory element.

5.3.2 Proof Structure

We base the proof of behaviour on the hierarchical structure of the master-slave device. A repeated sub-block of the master-slave consists of two NOR gates with a common gating signal and a latch (cf. Figure 6). In deducing the behaviour of this sub-block the theorem of behaviour of the latch is used. Since certain timing conditions are included in the latch theorem, the use of the latch result entails the introduction of appropriate conditions.

Two instances of the gated latch sub-block and two inverters, form the highest level of structure in the master-slave. The final part of the proof is to deduce the overall behaviour of the master-slave. More constraints are introduced when we do the final composition of sub-components.

5.4 Deduced Behaviour of Master-slave

We arrange the theorem of behaviour into the form:

```
delay_and_timing_conditions ==>
          master-slave_implementation ==>
                input_output_behaviour
```

The theorem of behaviour of the master-slave is presented in Figure 8.

5.4.1 Input-output Behaviour

The clock is low, high and low during the intervals `(tf_0,tr_0]`, `(tr_0,tf_1]` and `(tf_1,tr_1]` respectively. Data is presented on the input `d` for some interval around time `tr_0`, the first rise time.

If the input signals behave in the above manner then the data is available (along with its inverse) on the outputs `q` and `qbar` from `(d3 + (d8 + (d9 + (MAX(d4,d7)))))` time units after the first rise time, `tr_0`, until `(d3 + (MIN(d4,d7)))` after `tr_1`, the second rise time.

5.4.2 Delay and Timing Conditions

There are a number of conditions relating to the gate delays of the flip-flop. The delays of the gates used in the latches, `d5`, `d6`, `d8` and `d9`, must be non-zero and the condition `d3` $\leq$ `(MIN(d1,d2))` (which is a constraint on the relative delays of internal gates in the flip-flop) must hold.

```
tf_0 < (tr_0 - setup) ∧
(setup = (MAX(d1,d0 + d2)) + (d5 + d6)) ∧
(tr_0 + (d8 + (d9 + ((MAX(d4,d7)) - (MIN(d4,d7)))))) ≤ tf_1 ∧
d3 ≤ (MIN(d1,d2)) ∧
d5 > 0 ∧
d6 > 0 ∧
d8 > 0 ∧
d9 > 0 ⟹

   INV(d,dbar,d0) ∧
   INV(clk,clkbar,d3) ∧
   GATED_2NOR(d,dbar,clk,a_0,b_0,d1,d2) ∧
   LATCH(a_0,b_0,qa,qb,d5,d6) ∧
   GATED_2NOR(qa,qb,clkbar,a_1,b_1,d4,d7) ∧
   LATCH(a_1,b_1,q,qbar,d8,d9) ⟹

     DURING
     (tr_0 - setup,tr_0 + hold)
     (λt. d t = data) ∧
     DURING(tf_0,tr_0)(λt. ¬clk t) ∧
     DURING(tr_0,tf_1)(λt. clk t) ∧
     DURING(tf_1,tr_1)(λt. ¬clk t) ⟹
      DURING
      (tr_0 + (d3 + (d8 + (d9 + (MAX(d4,d7))))),
       tr_1 + (d3 + (MIN(d4,d7))))
         (λt. (q t = data) ∧ (qbar t = ¬data))
```

Figure 8: Theorem of behaviour of master-slave

There are also three conditions which relate to the external signals:

C0: `tf_0 < (tr_0 - setup)`

C1: `(tr_0 + (d8 + (d9 + ((MAX(d4,d7)) - (MIN(d4,d7)))))) ≤ tf_1`

C2: `setup = (MAX(d1,d0 + d2)) + (d5 + d6)`

C0 imposes a restriction on the minimum time the clock must remain low.

C1 is a restriction on the minimum time the clock must remain high.

C2 defines the minimum setup time.

Notice that there is no restriction on the variable `hold`. Therefore the master-slave flip-flop does not have a hold time constraint.

There are two different types of constraint:

- Restriction on some internal parameters. (*e.g.* relative gate delays)

- Restriction on the behaviour of external inputs. (*e.g.* setup time)

A master-slave flip-flop exhibits the desired behaviour of a synchronous memory element if its implementation satisfies the internal constraints and the applied input signals satisfy the external constraints.

6 Six Gate D Flip-flop

An alternative flip-flop to the master-slave is one built from 6 gates. This is also edge-triggered and is the more usual implementation for a flip-flop because it uses 4 fewer components. In this section we deduce that the D flip-flop exhibits the behaviour of a synchronous negative-edge triggered flip-flop.

6.1 Specification

We do not give a formal specification of required behaviour. The informal specification is identical to that for the master-slave flip-flop except that falling rather than rising edges are the active clocking events.

6.2 Implementation

The circuit diagram in Figure 9 shows a D flip-flop built from NOR gates. The structure can be divided into two parts - the front four gates and the final stage, consisting of a simple latch. Although the front section contains two latches, the extra coupling between the latches means that the resultant behaviour is complicated and the behaviour deduced for a simple latch is not applicable.

6.3 Proof of Behaviour

The proof is a forward one which starts with an assumption of a D flip-flop structure and a pattern of input signals. There are no initial restrictions on gate delays or the timing parameters of the input signals. As the proof progresses we introduce these conditions as necessary.

Starting Assumption

We begin by assuming an expression which describes the flip-flop structure and the behaviour of the input signals `ck` and `d`. This is

```
NOR2(s4, s2, d1, s1) ∧
NOR2(s1, ck, d2, s2) ∧
```

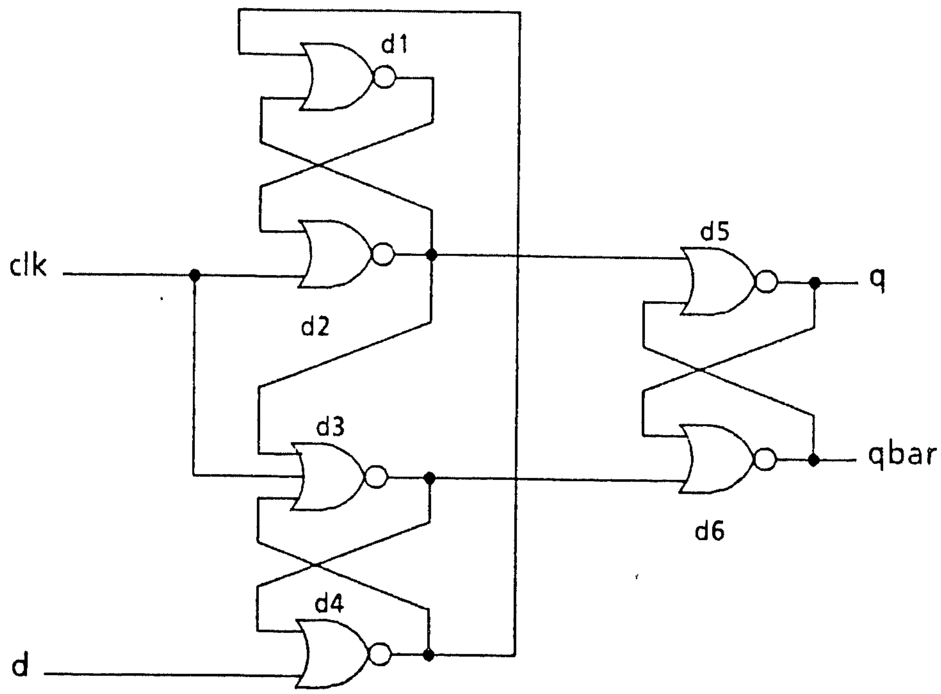

Figure 9: Structure of D flip-flop

```
NOR3(s2, ck, s4, d3, s3) ∧
NOR2(s3, d, d4, s4) ∧
LATCH(s2,s3,q,qbar,d5,d6) ∧
DURING(tr_0, tf_1) (λt. ck t = T) ∧
DURING(tf_1, tr_1) (λt. ck t = F) ∧
DURING(tr_1, tf_2) (λt. ck t = T) ∧
DURING(t1, t2) (λt. d t = data)
```

We assume that the clock signal is high for a length of time then low and then high again. We also assume that data is presented on the d input between certain times.

Proof Outline

The proof is confined to deducing the behaviour of the front four gates; the behaviour of the final latch is that deduced previously.

While the clock signal is high the feedback loops in the front section are inactive and the logic displays a combinational behaviour. We can use the derived rule PROPAGATE to deduce the behaviour of the signals.

When the clock signal goes low different behaviour may occur depending on the relationships between the timing parameters of the input signals and the gate delays and also depending on the relative gate delays. We choose the conditions to generate the behaviour we desire — the latching of the data presented for an interval of time around the falling edge of the clock. These conditions determine the setup and hold times for the data signal.

The value of the data signal and its inverse are stored in the bottom and top latches respectively (cf. Figure 9). We use induction to prove that the data values are stored for an indefinite time until the clock signal goes high. The induction follows a similar pattern to that used in the simple latch. This strategy is described in section 8.1.

The output section of the D flip-flop is a simple latch. The theorem of behaviour of the latch is used to derive the behaviour of the flip-flop outputs.

6.4 Deduced Behaviour of D Flip-flop

The theorem of behaviour deduced for the D flip-flop is presented in Figure 10. The theorem is in the following form:

```
delay and timing conditions ==>
          D-flip-flop implementation ==>
                input-output behaviour
```

6.4.1 Input-output Behaviour

The clock signal is high, low and high during the intervals `(tr_0,tf_1]` `(tf_1,tr_1]` `(tr_1,tf_2]` respectively, and data is presented on the input `d` during the interval `(t1,t2]`.

The outputs `q` and `qbar` present the input data and its inverse for an interval starting `(MAX(d2,d3)) + d5 + d6` after `tf_1`, time of the first negative edge, and ending `MIN(d2,d3)` after `tf_2`, time of the second negative edge. Data which is latched on a clocking edge is available (along with its inverse) for a length of time starting some time after that clocking edge and enduring until some time after the next clocking edge. This behaviour is characteristic of an edge-triggered synchronous memory element.

6.4.2 Delay and Timing Conditions

We use two predicates, `DELAYS_3_2` and `DELAY_GT`, to simplify the theorem of behaviour. A common rule of thumb used by digital hardware designers is the 3 for 2 law. This states that the delay through any 3 gates is greater than the delay through any 2 gates. A predicate `DELAYS_3_2` is defined which describes this 3 for

```
(0 < d1 ∧
 0 < d2 ∧
 0 < d3 ∧
 0 < d4 ∧
 0 < d5 ∧
 0 < d6 ∧
 DELAYS_3_2(d1,d2,d3,d4)) ∧
 ((tf_1 + x) ≤ tr_1 ∧ DELAY_GT x(d1,d2,d3,d4,d5,d6)) ∧
 (tr_0 + (d3 + (d4 + (d1 + 1)))) < tf_1 ∧
 (t1 + (d4 + d1)) < tf_1 ∧
 (tf_1 + d3) < t2  ==>

    NOR2(s4,s2,d1,s1) ∧
    NOR2(s1,ck,d2,s2) ∧
    NOR3(s2,ck,s4,d3,s3) ∧
    NOR2(s3,d,d4,s4) ∧
    LATCH(s2,s3,q,qbar,d5,d6) ==>

        DURING(tr_0,tf_1)(λt. ck t) ∧
        DURING(tf_1,tr_1)(λt. ¬ck t) ∧
        DURING(tr_1,tf_2)(λt. ck t) ∧
        DURING(t1,t2)(λt. d t = data) ==>
          DURING
          (tf_1 + ((MAX(d2,d3)) + (d5 + d6)),tf_2 + (MIN(d2,d3)))
          (λt. (q t = data) ∧ (qbar t = ¬data))
```

Figure 10: Behaviour of the D flip-flop

2 relationship for the gate delays in its argument. There are a number of similar restrictions on the relationship between tf_1 and tr_1. These can be combined into a single condition (tf_1 + x) ≤ tr_1 if x satisfies the predicate DELAY_GT x(d1,d2,d3,d4,d5,d6). This restricts x to be greater than the sum of any two of these delays plus the difference between d1 and d2.

The delay and timing conditions restrict both the internal delays and the timing parameters of the input signals. The internal restrictions are that gate delays must be non-zero and the 3 for 2 law must hold. The other conditions describe relationships between the timing parameters of the clock and data signals, and internal gate delays. These provide restrictions which must be satisfied by the input signals if the deduced behaviour is to occur.

These restrictions are:

```
C0:      (tf_1 + x) ≤ tr_1 ∧ DELAY_GT x(d1,d2,d3,d4,d5,d6)
C1:            (tr_0 + (d3 + (d4 + (d1 + 1)))) < tf_1
C2:                          (t1 + (d4 + d1)) < tf_1
C3:                               (tf_1 + d3) < t2
```

c0 is the constraint on the length of time the clock signal must remain low.

c1 is the constraint on the length of time the clock signal must remain high. The 1 in the constraint can for practical purposes be ignored. By choosing a sufficiently fine grain of time the 1 becomes insignificant.

c2 is the setup constraint on the data signal with respect to the clocking edge. The setup time is d4 + d1.

c3 is the hold constraint on the data signal with respect to the clocking edge. The hold time is d3.

6.5 Related Proof

A formal derivation of the behaviour of a D flip-flop has also been done by Hanna and Daeche [Hanna85]. A different model of components is used and different constraints are introduced in the proof of correctness. A comparison of that proof and our work is given in [Herbert86].

7 External Timing Parameters

The proofs of the master-slave and D flip-flop ended when the behaviour of synchronous memory elements was established. A lot of detail about structure and internal delays remained in the derived theorems of behaviour. We now use the master-slave flip-flop as an example to demonstrate how the internal details can be hidden and the device characterised by the external timing parameters.

In dealing with clock timing parameters, we call the length of time a clock signal is high the *mark* time, and the length of time it is low the *space* time.

7.1 Specification of Behaviour

The predicate POSITIVE_EDGE_FF is used to specify the behaviour of a positive edge-triggered flip-flop.
POSITIVE_EDGE_FF is defined as follows

```
POSITIVE_EDGE_FF(d,clk,q,qbar,setup,hold,mark,space,start,finish) =
   (∀data tf_0 tr_0 tf_1 tr_1.
     DURING(tr_0 - setup,tr_0 + hold)(λt. d t = data) ∧
     DURING(tf_0,tr_0)(λt. ¬clk t) ∧
     tf_0 < (tr_0 - space) ∧
     DURING(tr_0,tf_1)(λt. clk t) ∧
     (tr_0 + mark) < tf_1 ∧
```

```
DURING(tf_1,tr_1)(λt. ¬clk t) ⟹
   DURING
   (tr_0 + start,tr_1 + finish)
   (λt. (q t = data) ∧ (qbar t = ¬data))
```

The predicate POSITIVE_EDGE_FF is true of signals d,clk,q,qbar and timing parameters setup,hold,mark,space,start,finish if

> for all data values and clock timing parameters,
>
> whenever the data signal d satisfies the setup and hold times, setup and hold, and the clock signal clk satisfies the minimum clock high and low times, mark and space,
>
> then signals q and qbar present the data from time start after the sampling positive edge (tr_0) to time finish after the next sampling edge (tr_1).

The timing parameters setup,hold,mark and space have been defined earlier. Start and finish are associated with the output changes. Data is guaranteed to be stable on q and qbar a time start after the sampling edge, until a time finish after the next sampling edge. Start and finish are sometimes called the *maximum and minimum propagation delays* respectively. We prefer to use the term *propagation delay* solely for gate delay.

7.2 Specification of Master-slave Implementation

We define a predicate MASTER_SLAVE so that all the internal signals, gate delays and internal delay conditions of the master-slave are hidden. The timing parameters of MASTER_SLAVE are related to the internal delays and these relationships are also included in the definition.

The predicate MASTER_SLAVE is defined as follows:

```
MASTER_SLAVE(d,clk,q,qbar,setup,hold,mark,space,start,finish) =
  (∃dbar d0 clkbar d3 a_0 b_0 d1 d2 qa qb d5 d6 a_1 b_1 d4 d7 d8 d9.
    (setup = (MAX(d1,d0 + d2)) + (d5 + d6)) ∧
    (hold = 0) ∧
    (space = setup) ∧
    (mark = d8 + (d9 + ((MAX(d4,d7)) - (MIN(d7,d4))))) ∧
    (start = (MAX(d4,d7)) + (d3 + (d8 + d9))) ∧
    (finish = (MIN(d4,d7)) + d3) ∧
    INV(d,dbar,d0) ∧
    INV(clk,clkbar,d3) ∧
    NOR2(d,clk,d1,a_0) ∧
    NOR2(dbar,clk,d2,b_0) ∧
    NOR2(a_0,qb,d5,qa) ∧
```

```
NOR2(b_0,qa,d6,qb) ∧
NOR2(qa,clkbar,d4,a_1) ∧
NOR2(qb,clkbar,d7,b_1) ∧
NOR2(a_1,qbar,d8,q) ∧
NOR2(b_1,q,d9,qbar) ∧
MS_INTERNAL_CONDS(d1,d2,d3,d5,d6,d8,d9)
```

The predicate `MASTER_SLAVE` is true of signals `d,clk,q,qbar` and timing parameters `setup,hold,mark,space,start` and `finish` if there exist certain internal signals and delays such that

> the required relationships between internal signals hold (*e.g.* `INV(d,dbar,d0)`).
>
> the external parameters are related in a certain manner to the internal ones (*e.g.* `start = (MAX(d4,d7) + d3 + d8 + d9)`).
>
> the internal conditions are fulfilled (`MS_INTERNAL_CONDS(d1,d2,d3,d5,d6,d8,d9)` is true).

The internal conditions are described by the predicate `MS_INTERNAL_CONDS` defined by:

```
MS_INTERNAL_CONDS(d1, d2, d3, d5, d6, d8, d9) =
      d3 ≤ (MIN(d1,d2)) ∧
      d5 > 0 ∧
      d6 > 0 ∧
      d8 > 0 ∧
      d9 > 0
```

7.3 Proof of External Behaviour

We prove that the master-slave implementation achieves the specified behaviour of a positive-edge triggered device.
This theorem is:

```
MASTER_SLAVE(d,clk,q,qbar,setup,hold,mark,space,start,finish) ⟹
    POSITIVE_EDGE_FF(d,clk,q,qbar,setup,hold,mark,space,start,finish)
```

The theorem states that a master-slave implementation for which the predicate `MASTER_SLAVE` holds, achieves the behaviour of a positive-edge triggered device specified by `POSITIVE_EDGE_FF`.

We have formally deduced that a structure of gates with certain delays exhibits the behaviour of a synchronous edge-triggered flip-flop. The internal structure and delays of the implementation can be ignored and its behaviour taken as that of a "black box" synchronous flip-flop. Much unnecessary information is hidden and a simpler statement of behaviour for the device can henceforth be used.

7.4 Example of External Behavioural Parameters

We illustrate the above result by deducing external behavioural parameters for a gate-array implementation of the master-slave flip-flop.

Consider that we can obtain the structure and gate delays of the implementation from some CAD tool. The structure must match that of the master-slave and the internal gate delay constraints must be satisfied by the actual delay values. We can then deduce that the predicate POSITIVE_EDGE_FF holds for the external signals with timing parameters deduced from the internal delay values.

Gate delays are assigned using the bipolar gate-array data given in Appendix 2 and 0.1ns is taken as the basic unit of time.

The following theorem has been proved:

```
INV(d,dbar,d0) ∧
INV(clk,clkbar,d3) ∧
NOR2(d,clk,d1,a_0) ∧
NOR2(dbar,clk,d2,b_0) ∧
NOR2(a_0,qb,d5,qa) ∧
NOR2(b_0,qa,d6,qb) ∧
NOR2(qa,clkbar,d4,a_1) ∧
NOR2(qb,clkbar,d7,b_1) ∧
NOR2(a_1,qbar,d8,q) ∧
NOR2(b_1,q,d9,qbar) ∧
(d0 = 48) ∧
(d1 = 53) ∧
(d2 = 53) ∧
(d3 = 48) ∧
(d4 = 53) ∧
(d5 = 78) ∧
(d6 = 78) ∧
(d7 = 53) ∧
(d8 = 78) ∧
(d9 = 78) ⟹
  POSITIVE_EDGE_FF(d,clk,q,qbar,257,0,156,257,257,101)
```

The implementation acts as a positive-edge triggered flip-flop with a setup time of 25.7ns, 0ns hold time, minimum high and low clock times of 15.6ns and 25.7ns respectively, and start and finish times after the rising edges of 25.7ns and 10.1ns. Having deduced the external behaviour, the internal structure and delays of the device can be ignored.

8 Discussion

We present the main technique used to verify the behaviour of the memory devices and present some conclusions about the work.

8.1 Strategy Used in Proofs

The proof of behaviour of the front section of the D flip-flop follows a similar strategy to that used in the latch proof. The strategy involves using induction to prove that, under certain conditions, while external signals remain stable all signals remain stable. It is sometimes difficult to derive the asynchronous behaviour of structures with feedback. This technique can be applied to structures containing any configuration of feedback loops.

The following is an outline of the strategy.

- The induction variable, n, is chosen to be part of the upper limit of the interval during which signals are proposed to be stable.
- Stability over an interval corresponding to the base case is proven.
- Assuming the behaviour over the n^{th} interval, the rule `PROPAGATE` is used to deduce the behaviour over a later interval.
- This later interval is shown to include $(n+1)^{th}$ interval by proving that:
  1. the lower limit of this later interval is less than or equal to the upper limit of the base interval.
  2. the upper limit of this interval is greater than or equal to the upper limit of the $(n+1)^{th}$ interval.
- Stability over an interval whose upper limit is parameterised on n can then be deduced.

The first condition needed for the step case is fulfilled if the initial interval during which signals are stable is longer than the maximum delay from input to output of any component. The second condition is fulfilled if the component delays are non-zero.

For an arbitrary structure of non-zero delay components, containing any configuration of feedback loops, if one can deduce that all signals are stable over a base interval greater than the delay of any component, then one can prove that no signal will change until an external input changes.

For a digital design the result of the strategy is a proof that the circuit has settled. However, the proof strategy applies equally well to any other system of computing agents.

8.2 Conclusions

We have devised new techniques for reasoning about detailed timing of circuits and have introduced propagation delay gates as the primitive components of all digital circuits. We have derived the behaviour of asynchronous and synchronous memory elements constructed from propagation delay gates. The relationships between external timing parameters and the internal gate delays have been derived.

The formal proofs about timing level behaviour may seem difficult in comparison to simulation of similar devices. The formal proofs can be tedious, but the results obtained have a wider application than the results of simulation. The formal theorems of behaviour provide precise, unambiguous statements which can be manipulated and related to other behavioural descriptions. The use of variables for the delays and signals means that a formal theorem applies to a whole class of circuits.

For example, the theorem relating the behaviour of a master-slave structure to an external flip-flop specification is proved once, and is valid for all signal and delay values. One can simulate a particular master-slave implementation and determine its timing parameters. However, without knowing the symbolic relationship between timing parameters and gate delays, any implementation with different delays will again require simulation. While a formal proof can be more difficult than simulating a particular flip-flop, the results are applicable to all master-slaves and need never be repeated.

References

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

[Cohn87] A. Cohn, "A Proof of Correctness of the Viper Microprocessor: The First Level", Technical Report No. 104, Computer Laboratory, University of Cambridge, U.K., 1987.

[Gordon85a] M. J. C. Gordon, "HOL A Machine Oriented Formulation of

Higher Order Logic", Technical Report No. 68, Computer Laboratory, University of Cambridge, Cambridge, U.K., 1985.

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

[Hanna85] F. K. Hanna and N. Daeche, "Specification and Verification using Higher-Order Logic: A Case Study", Internal Report, University of Kent, U.K., 1985.

[Herbert86] J. M. J. Herbert, "Application of Formal Methods to Digital System Design", *Ph D Thesis*, University of Cambridge, U.K., 1986.

[Herbert88a] J. M. J. Herbert, "Case Study of the Cambridge Fast Ring ECL Chip using HOL", Technical Report, Computer Laboratory, University of Cambridge, U.K., 1988.

[Herbert88b] J. M. J. Herbert, "Temporal Abstraction of Digital Designs", Technical Report, Computer Laboratory, University of Cambridge, U.K., 1988.

[Seitz80] "System Timing", C. L. Seitz, *Chapter 7 of Introduction to VLSI Systems, C. Mead and L. Conway*, 1980.

Appendix 1

Source Code for Latch Proof

```
% The  percent symbol encloses comments such as this one %

% ***********************************************************************
  COMMENTS are enclosed in boxes like this.
  *********************************************************************** %


% ----------------------------------------------------------------------
   Some intermediate results of the proof are presented in boxes like this
   The general form of a theorem is:

              ... |-  thm_A

   This means that thm_A has a three hypotheses which for clarity are
     not printed.

   For example,
        q_is_low = .. |- DURING(t1_a + d0,t2_a + d0)(\t. ~q t)
  ---------------------------------------------------------------------- %


new_theory `latch`;;
new_parent `shift`;;

% ***********************************************************************
             We create a theory called latch.
  The immediate ancestor of this theory is theory shift.


                   Theory hierarchy:


     jh_basics
         |
         |
      MAX_MIN
         |
         |
       time_t                             time_bools
         |                                     |
         -------------------------------------
                        |
                        |
                    time_rules
                        |
                        |
                      DURING
                        |
                        |
                      shift
                        |
                        |
                      latch

  *********************************************************************** %
```

```
loadt `mk_latch_start`;;

% **********************************************************************
   The file latch_start loads various theorems and defines rules which are
                     used in the proof.
   The following rules and functions are not part of the basic HOL system
                     and are used in the proof.


 LEQ_TRANS          : (thm -> thm -> thm)
                    From  |- x <= y  and |- y <= z
                    Deduces |- x <- z

 MIN_of_MIN_simp    : (thm -> thm)
                    Simpifies terms involving MIN
                       such as MIN(a. MIN(a,b))

 CASES_RULE         : (thm -> thm -> thm)
                    From thms of form  |- a ==> c and |- (~a) ==> c
                    Deduces  |- c

 POST_ADD           : (term -> thm -> thm)
                    Adds the term to each side of an inequality in the theorem.
                    From  "p" and  |- n <= m
                    Deduce  |- (n + p) <= (m + p)

 SIMP_EXTEND_RULE : (thm -> thm -> thm)
                    From  |- DURING (t1,t2) P  and  |- DURING (t2,t3) P
                    Deduce   |- DURING (t1,t3) P

 USE_INEQUAL        : (thm -> thm -> thm)
                    Use the inequality in the first theorem
                     to simplify the second theorem.
                    The inequality can be of the form:
                      "a < b" "~(a < b)"  "a <= b" "~(a <= b)"

 SHIFT_RULE         : (thm -> thm)
                     Transforms a theorem of the form
                      "DURING (t1, t2) (\t. P(t + delta))"
                     into
                      "DURING (t1+delta, t2+delta) (\t. P t)"

 NARROW_RULE_TOP    : (thm -> thm -> thm)
                    From  |- DURING (t1,t2) P  and  |- t3 <= t2
                    Deduce   |- DURING (t1,t3) P

 NARROW_RULE_BOT    : (thm -> thm -> thm)
                    From  |- DURING (t1,t2) P  and  |- t1 <= t3
                    Deduce   |- DURING (t3,t2) P

 swap_x             : (term -> term -> thm -> thm)
                    Swaps the two terms if they occur in any
                    MAX subterms of the theorem.

 swap_n             : (term -> term -> thm -> thm)
                    Swaps the two terms if they occur in any
                    MIN subterms of the theorem.

 PROPAGATE          : (thm -> thm -> thm)
                    From a theorem which asserts that a signal
                         has a certain value over an interval
                    and a theorem which asserts the relationship
                        between a second signal and the first one
                    deduce a theorem for the behaviour of the second signal.
                    The second theorem can be an "ALWAYS" or "DURING"
                                 predicate.
                    This can be used to. in effect, propagate signals.

   ********************************************************************** %
```

```
% ********************************************************************
                      DEFINITIONS
  We define the predicates describing a NOR gate and a latch.
  We also define  "DATA_AVAILABLE" which is used to describe data being
   presented over an interval followed by the absence of data over a
   succeeding interval.
 ******************************************************************** %

let signal = ":num -> bool";;

let NOR2 = new_definition (`NOR2`,
      "NOR2 (in0:^signal, in1:^signal, del:num, out:^signal)
         =  ALWAYS (\t. out (t + del) = ~(in0 t \/ in1 t) )" );;

let LATCH = new_definition (`LATCH`,
 "LATCH (r:^signal, s:^signal, q:^signal, qbar:^signal, del0:num, del1:num)
          =   NOR2 (r, qbar, del0, q)
              /\  NOR2 (s, q, del1, qbar)");;


let DATA_AVAILABLE  = new_definition (`DATA_AVAILABLE`,
 "DATA_AVAILABLE (s:^signal, data:bool, t1:num, t2:num, tnext:num )
          =  DURING (t1, t2) (\t. s t = data)
            /\ DURING (t2, tnext) (\t. s t = F)");;


% ********************************************************************

                   STARTING ASSUMPTION

 The basic assumption that we work from is a latch is presented with some
  data and its inverse on its two inputs over certain intervals.
 No assumptions are made about the relationship between the timing
  parameters of a signal nor between the parameters of the two signals.
 ******************************************************************** %

let basic_ass =
  ASSUME "LATCH (a, b, q, qbar, d0, d1)
          /\ DATA_AVAILABLE (a, (~d), t1_a, t2_a, tnext_a)
          /\ DATA_AVAILABLE (b, d, t1_b, t2_b, tnext_b)";;


% ********************************************************************
 Expand out the definitions in the basic assumption and
        access the individual parts
 ******************************************************************** %

let [q_output; qbar_output; a_is_NOTd; a_is_low; b_is_d; b_is_low] =
  let ass_expanded = REWRITE_RULE [LATCH; NOR2; DATA_AVAILABLE]  basic_ass
    in
  map (\n. CONJUNCT n ass_expanded) (upto 1 6);;


% ********************************************************************
       ASSUME  that the data has value F
          and Simplify  the  basic theorems
 ******************************************************************** %

let [a_is_T; a_is_F; b_is_F_0; b_is_F_1] =
   let ass_d_F = ASSUME "d  = F"
     in
    map (REWRITE_RULE [ass_d_F]) [a_is_NOTd; a_is_low; b_is_d; b_is_low];;


% ********************************************************************
  The signal "b" has value F in its active and inactive regions
       Therefore we can concatenate into a single extended region
 ******************************************************************** %

let b_low = SIMP_EXTEND_RULE b_is_F_0 b_is_F_1;;
```

```
% ********************************************************************

                     PROPAGATE INPUTS

   We now generate new assertions for the "q" and "qbar" outputs by using
     the assumptions about the inputs "a" and "b".
   We are in effect propagating the signals through the gates.
   The three theorems generated state:
            when "q" is low because of input signal "a"
            how the signal "qbar" follows signal "q"
            how the signal "q" follows signal "qbar"
   These theorems will be used many times in the proof.
  ******************************************************************** %

let q_is_low   = SHIFT_RULE (PROPAGATE  a_is_T  q_output )::

% --------------------------------------------------------------------
   q_is_low = .. |- DURING(t1_a + d0.t2_a + d0)(\t. ~q t)
  -------------------------------------------------------------------- %

let qbar_fn_q  = PROPAGATE  b_low  qbar_output::
% --------------------------------------------------------------------
    qbar_fn_q = .. |- DURING(t1_b.tnext_b)(\t. qbar(t + d1) = ~q t)
  -------------------------------------------------------------------- %

let q_fn_qbar  = PROPAGATE  a_is_F  q_output::
% --------------------------------------------------------------------
    q_fn_qbar = . |- DURING(t2_a.tnext_a)(\t. q(t + d0) = ~qbar t)
  -------------------------------------------------------------------- %


%  ********************************************************************
   Deduce the behaviour for "qbar" from the assertion of "q" being low and
       the relationship between "q" and "qbar".
  ******************************************************************** %

let qbar_T_0  = PROPAGATE  q_is_low   qbar_fn_q::

% --------------------------------------------------------------------
 .. |- DURING
      (MAX(t1_a + d0,t1_b),MIN(t2_a + d0.tnext_b))
      (\t. qbar(t + d1))
  -------------------------------------------------------------------- %


% ********************************************************************
      Since the timing parameters of the input signals are independent
         we do not know their relative magnitudes.
      This results in a number of branches in the proof.
      We will follow the main path at each branch and follow the trivial
                     paths later on.
  ******************************************************************** %

% ********************************************************************
   Follow first main branch corresponding to the assumption that:
                      "(t2_a + d0) <= tnext_b"
  ******************************************************************** %

let main_branch_1_ass = ASSUME "(t2_a + d0) <= tnext_b"::

let qbar_T_1 =
   let t4 = USE_INEQUAL (main_branch_1_ass) qbar_T_0
    in
   SHIFT_RULE t4::
% --------------------------------------------------------------------
  ... |- DURING((MAX(t1_a + d0,t1_b)) + d1.(t2_a + d0) + d1)(\t. qbar t)
  -------------------------------------------------------------------- %
```

```
% ********************************************************************

                 LATCHING OF DATA

   Deduce a theorem which states the initial latching of data.
   Use latching assumption    "((MAX(t1_a + d0,t1_b)) + d1) <= t2_a".
   Follow second main branch corresponding to condition:
              "((t2_a + d0) + d1) <= MIN(tnext_a,tnext_b)"
   Reduce upper limit of interval of the theorem
     from   "((t2_a + d0) + d1) + d0"   to   "(t2_a + d0) - d1"
   to fit in with the form of the inductive proof later on.
  ******************************************************************** %

let latching_ass =  ASSUME  "((MAX(t1_a + d0,t1_b)) + d1) <= t2_a";;

let latching_thm =
 let q_F_0  = PROPAGATE   q_fn_qbar  qbar_T_1
  in
 let q_F_1  = USE_INEQUAL  latching_ass  q_F_0
  in
 let main_branch_2_tm  = "((t2_a + d0) + d1) <= MIN(tnext_a,tnext_b)"
  in
 let main_branch_2_ass = ASSUME  main_branch_2_tm
 and MIN_next_a    = SPECL ["tnext_a:num"; "tnext_b:num"] (GEN_ALL MIN_less_1)
  in
let leq_MIN  = LEQ_TRANS  main_branch_2_ass  MIN_next_a
  in
 let latch_0 = USE_INEQUAL  leq_MIN   q_F_1
  in
 let latch_1 = SIMP_EXTEND_RULE q_is_low (SHIFT_RULE latch_0)
  in
 let th_top  = SPECL ["(t2_a + d0) + d1"; "d0"] LESS_EQ_ADD
  in
 let latch_2 = NARROW_RULE_TOP  latch_1   th_top
  in
 DISCH  main_branch_2_tm   latch_2;;

% --------------------------------------------------------------------
    latching_thm =
.... |- ((t2_a + d0) + d1) <= (MIN(tnext_a,tnext_b)) ==>
        DURING(t1_a + d0,(t2_a + d0) + d1)(\t. ~q t)

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

```
% ***********************************************************************
   To allow us to do a proof by induction, we need to prove the step case.
   (i.e. P n ==> P (n+1) )
   We firstly assume some behaviour for "q" parameterised on "n" and then
      deduce some  resultant behaviour  parameterised on  "SUC n".
   We assume the proposition is true for n:
     "(((t2_a + d0) + d1) + n) <=  MIN(tnext_a,tnext_b) ==>
       DURING(t1_a + d0,((t2_a + d0) + d1) + n)(\t. ~q t)";;
   and deduce
     "DURING(t1_a + d0,((t2_a + d0) + d1) + (SUC n))(\t. ~q t)" .

  First deduce "DURING(t1_a + d0,(((t2_a + d0) + d1) + n) + d0)(\t. ~q t)"
  We need "SUC n" rather than  "n+d0".
  Must assume  "0 < d0"  to allow us to deduce the proposition for  n+1.
  (i.e. The behaviour as a memory from cycle n to n+1  requires that the
       gate delays are non-zero.)
  *********************************************************************** %

let SUC_n_thm =

 let ass_n = ASSUME "(((t2_a + d0) + d1) + n) <=  MIN(tnext_a,tnext_b) ==>
                     DURING(t1_a + d0,((t2_a + d0) + d1) + n)(\t. ~q t)"
  in
 let qbar_unw0  = PROPAGATE  (UNDISCH ass_n)   qbar_fn_q
 and n_cond     = ASSUME "(((t2_a + d0) + d1) + n) <= (MIN(tnext_a,tnext_b))"
 and MIN_next_b = SPECL ["tnext_a:num"; "tnext_b:num"] (GEN_ALL MIN_less_2)
  in
 let b_next     = LEQ_TRANS n_cond  MIN_next_b
  in
 let qbar_unw1 = USE_INEQUAL  b_next  qbar_unw0
  in
 let q_unw0   = PROPAGATE  (SHIFT_RULE qbar_unw1) q_fn_qbar
  in
 let q_unw1   = USE_INEQUAL  latching_ass  q_unw0
 and top_n     =
      SPECL ["(((t2_a + d0) + d1) + n)"; "tnext_a"; "d1"] (GEN_ALL MIN_RULE_1)
  in
 let q_unw2   = NARROW_RULE_TOP q_unw1  top_n
  in
 let amin       = ASSUME "(((t2_a + d0) + d1) + n) <= (MIN(tnext_a,tnext_b))"
 and MIN_next_a = SPECL ["tnext_a:num"; "tnext_b:num"] (GEN_ALL MIN_less_1)
  in
 let l_n  = LEQ_TRANS  amin  MIN_next_a
  in
 let q_unw3  = USE_INEQUAL l_n  q_unw2
  in
 let thm_n_plus_d0 =  SIMP_EXTEND_RULE  q_is_low   (SHIFT_RULE q_unw3)
  in
 let ad     = ASSUME "0 < d0"
  in
 let ad_1  = REWRITE_RULE [LESS_EQ; ADD1; ADD_CLAUSES] ad
  in
 let get_n = (SPECL ["1":"d0";"((t2_a + d0) + d1) + n" ] o Sym o CONJUNCT 2)
                  LE_CLAUSES
  in
 let ineq  = EQ_MP get_n ad_1
  in
 let thm_n_1   = NARROW_RULE_TOP  thm_n_plus_d0  ineq
  in
 let thm_SUCn  = REWRITE_RULE [Sym ADD_ASSOC; Sym ADD1] thm_n_1
   in
REWRITE_RULE [ADD_ASSOC] thm_SUCn;;
% -----------------------------------------------------------------------
  SUC_n_thm =
...... |- DURING(t1_a + d0,((t2_a + d0) + d1) + (SUC n))(\t. ~q t)

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

```
% **************************************************************************
    Get into the form   P n ==> P(n+1)
  ************************************************************************** %

let n_imp_SUCn =
 let SUC_imp = IMP_TRANS (SPEC_ALL OR_LESS) (SPEC_ALL LESS_IMP_LESS_OR_EQ)
 and n_tm    = "(((t2_a + d0) + d1) + n) <= (MIN(tnext_a,tnext_b))"
 and np1_tm  = "(((t2_a + d0) + d1) + (SUC n)) <= (MIN(tnext_a,tnext_b))"
  in
 let a_n1 = ASSUME np1_tm
  in
 let a_n2 = REWRITE_RULE [ADD_CLAUSES] a_n1
  in
 let a_n3 = MATCH_MP  SUC_imp  a_n2
  in
 let as11 = REWRITE_RULE [a_n3] (DISCH n_tm  SUC_n_thm)
  in
 let as12 = DISCH  np1_tm  as11
  in
DISCH (element 4 (hyp as12)) as12::
% -------------------------------------------------------------------------
  n_imp_SUCn =
.... |- ((((t2_a + d0) + d1) + n) <= (MIN(tnext_a,tnext_b)) ==>
         DURING(t1_a + d0,((t2_a + d0) + d1) + n)(\t. ~q t)) ==>
        (((t2_a + d0) + d1) + (SUC n)) <= (MIN(tnext_a,tnext_b)) ==>
        DURING(t1_a + d0,((t2_a + d0) + d1) + (SUC n))(\t. ~q t)

  ------------------------------------------------------------------------- %


% **************************************************************************
    Use the initial latching theorem to prove  the base case, n = 0.
  ************************************************************************** %

set_goal ([], "((((t2_a + d0) + d1) + 0) <= (MIN(tnext_a,tnext_b)) ==>
         DURING(t1_a + d0,((t2_a + d0) + d1) + 0)(\t. ~q t)) ");;
expandf (REWRITE_TAC [ADD_CLAUSES]);;
expandf (ACCEPT_TAC latching_thm);;

let base_case = save_top_thm `base_case`;;


% **************************************************************************
    Prove by Induction that "q" retains its value for an arbitrary length
     of time until the new input data arrives.
  ************************************************************************** %

set_goal ([], "!n. ((((t2_a + d0) + d1) + n) <= (MIN(tnext_a,tnext_b)) ==>
         DURING(t1_a + d0,((t2_a + d0) + d1) + n)(\t. ~q t)) ");;
expandf (INDUCT_TAC );;
expandf (ACCEPT_TAC base_case);;

expandf (IMP_RES_TAC  n_imp_SUCn);;

let induct_thm = save_top_thm `induct_thm`;;
```

```
% ***********************************************************************
    Use the maximum value of n which generates a true antecedent to get
     the desired theorem of behaviour of "q" over an interval.

   Can extend the upper limit of this interval during which "q" retains its
   value by  propagating the through both gates and using the latching
   assumption.
  *********************************************************************** %

let main_result =
 let n_swap = SPECL ["((t2_a + d0) + d1)"; "n"]  ADD_SYM
  in
 let thm_swap = (GEN "n:num" o SUBS[n_swap] o SPEC_ALL) induct_thm
  in
 let as = ASSUME "((t2_a + d0) + d1) <= (MIN(tnext_a,tnext_b))"
  in
 let simp = MATCH_MP  SUB_ADD  as
  in
 let num_term = "(MIN(tnext_a,tnext_b)) - ((t2_a + d0) + d1)"
  in
 let spec_thm = SPEC num_term  thm_swap
  in
 let res = REWRITE_RULE [simp;LESS_EQ_REFL] spec_thm
  in
 let prop_th1 = PROPAGATE  res  qbar_fn_q
  in
 let prop_th2 = SHIFT_RULE prop_th1
  in
 let res_ex_0 = PROPAGATE  prop_th2  q_fn_qbar
 and chop_d1  = (SPECL ["(MIN(MIN(tnext_a,tnext_b),tnext_b))";"tnext_a";"d1"]
                 o GEN_ALL)  MIN_RULE_1
  in
 let res_ex_1 = NARROW_RULE_TOP  res_ex_0  chop_d1
   in
 let res_ex_2 = MIN_of_MIN_simp  res_ex_1
   in
 let res_ex_3 = USE_INEQUAL  latching_ass   res_ex_2
   in
 SIMP_EXTEND_RULE q_is_low (SHIFT_RULE res_ex_3);;

% ---------------------------------------------------------------------
  main_result =
......  |- DURING(t1_a + d0,(MIN(tnext_a,tnext_b)) + d0)(\t. ~q t)

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

```
% **********************************************************************
   First trivial branch corresponding to the condition:
                    "~ ((t2_a + d0) <= tnext_b)"
  ********************************************************************** %

let triv_branch_1 =
 let branch_2_ass = ASSUME "~ ((t2_a + d0) <= tnext_b)"
  in
 let qbar_0      = USE_INEQUAL branch_2_ass  qbar_T_0
 and MIN_next_b = SPECL ["tnext_a:num": "tnext_b:num"] (GEN_ALL MIN_less_2)
  in
 let qbar_1 = NARROW_RULE_TOP  qbar_0   MIN_next_b
  in
 let q_0     = PROPAGATE  q_fn_qbar  (SHIFT_RULE qbar_1)
 and min_elim =
     let xm = (SPECL ["tnext_a":"MIN(tnext_a.tnext_b)":"d1"] o GEN_ALL)
                   MIN_RULE_2
     and min_of_min = GEN_ALL (MATCH_MP leq_imp_min MIN_less_eq)
        in
     REWRITE_RULE [min_of_min]  xm
  in
 let q_1 = NARROW_RULE_TOP   q_0   min_elim
  in
 let q_2 = USE_INEQUAL latching_ass  q_1
  in
 SIMP_EXTEND_RULE q_is_low (SHIFT_RULE q_2)::

% **********************************************************************
   Second trivial branch corresponding to the condition:
            "~ (((t2_a + d0) + d1) <= MIN(tnext_a, tnext_b))"
  ********************************************************************** %

let triv_branch_2 =
 let aF_M  = ASSUME "~ (((t2_a + d0) + d1) <= MIN(tnext_a, tnext_b))"
  in
 let aF_M1 = REWRITE_RULE [Sym LESS_eq_NOT] aF_M
  in
 let aF_M2 = MATCH_MP LESS_IMP_LESS_OR_EQ   aF_M1
  in
 let qbar_M0 = NARROW_RULE_TOP  qbar_T_1   aF_M2
  in
 let q_M0     = PROPAGATE  q_fn_qbar   qbar_M0
 and MIN_next_a = SPECL ["tnext_a:num": "tnext_b:num"] (GEN_ALL MIN_less_1)
  in
 let q_M1  = USE_INEQUAL  MIN_next_a  q_M0
  in
 let M1 = USE_INEQUAL (ASSUME "((MAX(t1_a + d0.t1_b)) + d1) <= t2_a") q_M1
  in
 SIMP_EXTEND_RULE q_is_low (SHIFT_RULE M1)::

% ***********************************************************************
  Combine the results for the trivial branches with the main result.
  *********************************************************************** %

let q_thm  =
 let resT = DISCH "((t2_a + d0) + d1) <= (MIN(tnext_a.tnext_b))"  main_result
 and resF = DISCH "~((t2_a + d0) + d1) <= (MIN(tnext_a.tnext_b))"
                          triv_branch_2
  in
 let thm_0 = CASES_RULE  resT resF
  in
 let thm_0_T = DISCH "(t2_a + d0) <= tnext_b"   thm_0
 and thm_0_F = DISCH "~(t2_a + d0) <= tnext_b"  triv_branch_1
  in
 CASES_RULE  thm_0_T   thm_0_F ::

% ---------------------------------------------------------------------
  q_thm = .... |- DURING(t1_a + d0.(MIN(tnext_a.tnext_b)) + d0)(\t. ~q t)
  --------------------------------------------------------------------- %
```

```
% ************************************************************************
   Can deduce the equivalent thm for "qbar" by using the relationship
     between "q" and "qbar" in "qbar_fn_q".
   We narrow the  resultant interval so that the resultant theorem does not
     reflect dependency on the value of "d".
  ************************************************************************ %

let qbar_thm =
 let qbar_res  = PROPAGATE  q_thm   qbar_fn_q
 and chop_d0   = (SPECL ["MIN(tnext_a.tnext_b)";"tnext_b";"d0"] o
                           GEN_ALL)  MIN_RULE_1
  in
 let qbar_res0 = NARROW_RULE_TOP  qbar_res  chop_d0
  in
 let qbar_res1 = MIN_of_MIN_simp  qbar_res0
 and bot       = (SPECL ["t1_a"; "d0"; "t1_b"] o GEN_ALL) MAX_RULE_1
  in
 let qbar_res2 = NARROW_RULE_BOT   qbar_res1  bot
  in
 SHIFT_RULE qbar_res2;;

% ----------------------------------------------------------------------
  qbar_thm =
.... |- DURING
       (((MAX(t1_a.t1_b)) + d0) + d1,(MIN(tnext_a.tnext_b)) + d1)
       (\t. qbar t)
   -------------------------------------------------------------------- %


% ************************************************************************
   We now form a theorem stating the behaviour of "q" and "qbar" for "d = F"
   We introduce "d" and "~d" for the data values "T" and "F" on the outputs
                         "q" and "qbar".
   We change the lower limit of the interval for "q" so that it will yield
       a suitable value under symmetry.
  ************************************************************************ %

let q_and_qbar0 =

 let q_d_thm, qbar_d_thm =
  let qbar_T = (SYM o CONJUNCT 2 o SPEC "(qbar:^signal) (t:num)") EQ_CLAUSES
  and q_F    = (SYM o CONJUNCT 4 o SPEC "(q:^signal) (t:num)") EQ_CLAUSES
  and d_F    = ASSUME "d = F"
   in
  let T_is_NOTd = (SYM o PURE_REWRITE_RULE[NOT_CLAUSES] o all_BETA_RULE o
                       AP_TERM "\t.~ t ") d_F
  and  F_is_d   = SYM  d_F
   in
  let q_fn_d    = PURE_REWRITE_RULE[F_is_d] q_F
  and qbar_fn_d = PURE_REWRITE_RULE[T_is_NOTd] qbar_T
   in
  (PURE_REWRITE_RULE_1 [q_fn_d]  q_thm,
   PURE_REWRITE_RULE_1 [qbar_fn_d]  qbar_thm)
  in
 let t1_a0 = (SPECL["t1_a"; "t1_b"] o GEN_ALL) MAX_great_eq
  in
 let t1_a1 = POST_ADD  "d0"  t1_a0
 and add_th = SPECL["((MAX(t1_a.t1_b)) + d0)": "d1"] LESS_EQ_ADD
  in
 let nr_th  = LEQ_TRANS  t1_a1  add_th
  in
 let q_d_thm1 = NARROW_RULE_BOT  q_d_thm  nr_th
  in
 CONJ  q_d_thm1  qbar_d_thm::

% ----------------------------------------------------------------------
  q_and_qbar0 =
.... |- DURING
       (((MAX(t1_a.t1_b)) + d0) + d1,(MIN(tnext_a.tnext_b)) + d0)
       (\t. q t = d) /\
       DURING
       (((MAX(t1_a.t1_b)) + d0) + d1,(MIN(tnext_a.tnext_b)) + d1)
       (\t. qbar t = ~d)
  -------------------------------------------------------------------- %
```

```
% ************************************************************************

                    USE LATCH SYMMETRY

   We use the symmetry of the latch to deduce the converse theorem
                    for "d = T" .
   This involves swapping "a" and "b", "q" and "qbar", "d0" and "d1",
                    "t1_a" for "t1_b"  etc.
 ************************************************************************ %

let converse_thm =
 let converse_thm0 =
   (INST [("b:num->bool","a:num->bool");("a:num->bool","b:num->bool")] o
    SPECL["t1_b";"t2_b";"tnext_b";"t1_a";"t2_a";"tnext_a"] o
    GENL ["t1_a";"t2_a";"tnext_a";"t1_b";"t2_b";"tnext_b"] o
    SPECL["qbar";"q";"d1";"d0"] o
    GENL  ["q";"qbar";"d0";"d1"] o
    SPEC "~d" o
    GEN "d")  (DISCH_ALL q_and_qbar0)
  in
 let latch_eq =
   let c = CONJUNCTS_CONV ("NOR2(a,qbar,d0,q)  /\ NOR2(b,q,d1,qbar)",
                           "NOR2(b,q,d1,qbar)  /\ NOR2(a,qbar,d0,q)" )
    in
   REWRITE_RULE[Sym LATCH] c
  in
 let th = SUBS [SYM latch_eq] converse_thm0
   in
 PURE_REWRITE_RULE [NOT_CLAUSES] th::

% ----------------------------------------------------------------------
  converse_thm =
|- 0 < d1 ==>
   ((MAX(t1_b + d1,t1_a)) + d0) <= t2_b ==>
   LATCH(a,b,q,qbar,d0,d1) /\
   DATA_AVAILABLE(b,d,t1_b,t2_b,tnext_b) /\
   DATA_AVAILABLE(a,~d,t1_a,t2_a,tnext_a) ==>
   (~d = F) ==>
   DURING
   (((MAX(t1_b,t1_a)) + d1) + d0,(MIN(tnext_b,tnext_a)) + d1)
   (\t. qbar t = ~d) /\
   DURING
   (((MAX(t1_b,t1_a)) + d1) + d0,(MIN(tnext_b,tnext_a)) + d0)
   (\t. q t = d)

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

```
% ************************************************************************
    We now deduce theorems for both "d = T" and "d = F".
    The form of the latching assumption differs for "d = T" and "d = F"
             "((MAX(t1_a + d0,t1_b)) + d1) <= t2_a"
         and  "((MAX(t1_b + d1,t1_a)) + d0) <= t2_b" .
    By assuming a more general condition:
           "(((MAX(t1_a,t1_b)) + d0) + d1) <= MIN(t2_a,t2_b)"
    we can have the same latching assumption in both theorems.
    We also rearrange some terms in the theorem for "d = T" (deduced using
                       circuit symmetry)
  ************************************************************************ %

let d_eq_T, d_eq_F =

 let cont_dT, cont_dF =
  let mx_mn_0 = ASSUME "(((MAX(t1_a,t1_b)) + d0) + d1) <= MIN(t2_a,t2_b)"
  and min_a   = (SPECL["t2_a"; "t2_b"] o GEN_ALL) MIN_less_1
  and max_a   = (SPECL["t1_a"; "d0"; "t1_b"] o GEN_ALL)  MAX_RULE_1
   in
  let max_a0 = POST_ADD "d1" max_a
   in
  let min_b  = (SPECL["t2_a"; "t2_b"] o GEN_ALL) MIN_less_2
  and max_b  = (SPECL["t1_b"; "d1"; "t1_a"] o GEN_ALL)  MAX_RULE_1
   in
  let max_b0 = POST_ADD "d0" max_b
   in
  let max_b1 = (PURE_REWRITE_RULE[ADD_ASSOC] o
                SUBS [ SPECL["d1"; "d0"] ADD_SYM ] o
                PURE_REWRITE_RULE[Sym ADD_ASSOC] o
                swap_x "t1_b" "t1_a")  max_b0
   in
 (LEQ_TRANS (LEQ_TRANS max_b1 mx_mn_0) min_b,
  LEQ_TRANS (LEQ_TRANS max_a0 mx_mn_0) min_a)
 in
let Tthm_1 = PURE_REWRITE_RULE [cont_dT;IMP_CLAUSES] converse_thm
and Fthm_1 = PURE_REWRITE_RULE [cont_dF;IMP_CLAUSES] (DISCH_ALL q_and_qbar0)
 in
let  Tthm_2  = (PURE_REWRITE_RULE[ADD_ASSOC] o
                SUBS [ SPECL["d1"; "d0"] ADD_SYM ] o
                PURE_REWRITE_RULE[Sym ADD_ASSOC] o
                swap_x "t1_b" "t1_a" o
                swap_n "tnext_b" "tnext_a" )  Tthm_1
 in
(Tthm_2, Fthm_1);;

% ---------------------------------------------------------------------
  d_eq_T =
 . |- 0 < d1 ==>
     LATCH(a,b,q,qbar,d0,d1) /\
     DATA_AVAILABLE(b,d,t1_b,t2_b,tnext_b) /\
     DATA_AVAILABLE(a,~d,t1_a,t2_a,tnext_a) ==>
     (~d = F) ==>
     DURING
     (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d1)
     (\t. qbar t = ~d) /\
     DURING
     (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d0)
     (\t. q t = d)
 d_eq_F =
 . |- 0 < d0 ==>
     LATCH(a,b,q,qbar,d0,d1) /\
     DATA_AVAILABLE(a,~d,t1_a,t2_a,tnext_a) /\
     DATA_AVAILABLE(b,d,t1_b,t2_b,tnext_b) ==>
     (d = F) ==>
     DURING
     (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d0)
     (\t. q t = d) /\
     DURING
     (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d1)
     (\t. qbar t = ~d)

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

```
% ***************************************************************************

               DEDUCED BEHAVIOUR OF LATCH

   We combine the theorems deduced for "d = T" and "d = F" to prove
    a theorem giving the behaviour of "q" and "qbar" for any data value "d"
   We put the theorem into the form:
           delay and timing conds ==>
                 latch implementation ==>
                      input-output behaviour
  *************************************************************************** %


let final_thm =

 let conds = "0 < d0 /\
              0 < d1 /\
              (((MAX(t1_a,t1_b)) + d0) + d1) <= (MIN(t2_a,t2_b))"
 and lat = "LATCH (a,b,q,qbar,d0,d1)"
 and ins = " DATA_AVAILABLE(a,(~ d),t1_a,t2_a,tnext_a) /\
                   DATA_AVAILABLE(b,d,t1_b,t2_b,tnext_b)"
  in
 let dT = (REWRITE_RULE (map ASSUME [conds; lat; ins]) o DISCH_ALL) d_eq_T
 and switch = (\thm. CONJ (CONJUNCT2 thm) (CONJUNCT1 thm))
 and dF = (REWRITE_RULE (map ASSUME [conds; lat; ins])  o DISCH_ALL) d_eq_F
  in
 let dT0 = (DISCH "d" o switch o UNDISCH) dT
  in
 let thm = CASES_RULE dT0 dF
  in
(DISCH conds o DISCH lat o DISCH ins)  thm;;

% --------------------------------------------------------------------------

final_thm =
|- 0 < d0 /\
   0 < d1 /\
   (((MAX(t1_a,t1_b)) + d0) + d1) <= (MIN(t2_a,t2_b)) ==>
   LATCH(a,b,q,qbar,d0,d1) ==>
   DATA_AVAILABLE(a,~d,t1_a,t2_a,tnext_a) /\
   DATA_AVAILABLE(b,d,t1_b,t2_b,tnext_b) ==>
   DURING
   (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d0)
   (\t. q t = d) /\
   DURING
   (((MAX(t1_a,t1_b)) + d0) + d1,(MIN(tnext_a,tnext_b)) + d1)
   (\t. qbar t = ~d)

  -------------------------------------------------------------------------- %

save_thm (`final_thm`, final_thm);;

close_theory();;
quit();;
```

Appendix 2

Delay of Gates in Bipolar Gate Array

Typical gate delay in nanoseconds as a function of fan-in and fan-out.

Fan-out

Fan-in	1	2	3	4	5	6
1	4.8	7.3	9.8	12.3	14.9	17.4
2	5.3	7.8	10.3	12.9	15.4	17.9
3	8.1	10.7	13.2	15.7	18.2	20.7
4	8.7	11.2	13.7	16.2	18.7	21.3
5	11.5	14.0	16.5	19.1	21.6	24.1
6	12.0	14.5	17.1	19.6	22.1	24.6
7	14.9	17.4	19.9	22.4	24.9	27.5
8	15.4	17.9	20.4	22.9	25.5	28.0