Technical Report

UCAM-CL-TR-136
ISSN 1476-2986

Number 136

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Formal specification and verification of asynchronous processes in higher-order logic

Jeffrey J. Joyce

June 1988

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
http://www.cl.cam.ac.uk/

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

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic

Jeffrey J. Joyce

University of Cambridge
Computer Laboratory
Cambridge, England

Abstract

We model the interaction of a synchronous process with an asynchronous memory process using a four-phase "handshaking" protocol. This example demonstrates the use of higher-order logic to reason about the behaviour of synchronous systems such as microprocessors which communicate requests to asynchronous devices and then wait for unpredictably long periods until these requests are answered. We also describe how our model could be revised to include some of the detailed timing requirements found in real systems such as the M68000 microprocessor. One enhancement uses non-determinism to model minimum setup times for asynchronous inputs. Experience with this example suggests that higher-order logic may also be a suitable formalism for reasoning about more abstract forms of concurrency.

1 Introduction

This paper describes the use of higher-order logic to model the interaction of a synchronous process with an asynchronous process using a four-phase "handshaking" protocol. Other formalisms, notably temporal logic, have also been used to reason about handshaking protocols [1,6,7]. In our approach, universal and existential quantification are used in approximately the same roles as the operators $\Box$ ("forever") and $\Diamond$ ("eventually"). But unlike temporal logic, our specifications allow explicit references to time; for example, $f(t)$ and $f(t+1)$ denote the values of signal f at two successive points in discrete time. Higher-order functions are used to define predicates expressing conditions such a signal rising or the stability of a signal over a specified interval. The use of explicit time references and higher-order functions leads to convenient mechanisms for deriving abstract views of system behaviour where intervals of time are collapsed into single time steps.

Our use of higher-order logic to verify signalling protocols is based on experience with this formalism in verifying synchronous hardware. We have used

higher-order logic to verify the correctness of a very simple microprocessor where the formal semantics of the instruction set have been formally derived from a switch-level model of MOS circuit behaviour [18]. The current design for this microprocessor assumes that every read and write request to external memory is completed in a single clock cycle. This assumption could be validated by a detailed analysis of timing characteristics for both the microprocessor and the external memory; for example, see [2]. We are now designing a new version of this microprocessor which has wait states where the microprocessor waits one or more clock cycles after issuing a read or write request to the external memory until the request is acknowledged. Wait states are implemented by while-loops in the microcode.

The example described in this paper was developed to investigate how interaction between the microprocessor and the asynchronous memory could be formally specified in higher-order logic. To focus more clearly on this problem, the microprocessor specification is replaced by an abstract state machine which models the interaction of the microprocessor with the asynchronous memory. The abstract state machine implements a synchronous interface which accepts synchronous read and write requests and communicates these asynchronously to the external memory. We develop formal specifications for both the synchronous process and the asynchronous memory and prove that these two processes correctly implement the four-phase handshaking protocol.

2 Related Work

Warren Hunt [15] also considers the formal specification and verification of a microprocessor system which communicates asynchronously with external memory. The absence of existential quantification in quantifier-free Boyer-Moore logic used by Hunt motivates the use of "oracles" to predict asynchronous responses from the memory. An oracle is a list of numbers which specifies the length of wait states for a particular execution sequence; Hunt's microprocessor is shown to be correct for all such oracles.

The VIPER microprocessor also uses four-phase handshaking to communicate asynchronously with external memory [24]. The formal proof described by Avra Cohn in [4] deals with higher level aspects of the VIPER implementation where the details of asynchronous communication are not considered. However, the proof models the possibility of a failed memory interaction which occurs when a wait state exceeds a specified "timeout" period.

The use of higher-order logic to reason about asynchronous behaviour in the verification of latches and flip-flops implemented by fixed delay NOR gates is described by John Herbert in [12]. Herbert reasons about asynchronous behaviour at the nanosecond time scale whereas this paper focuses on the behaviour of an asynchronous memory in terms of a much coarser grain of time corresponding to the behaviour of clocked sequential circuits. However, the main difference in

this paper is that the delayed response of the asynchronous component, *i.e.* the asynchronous memory, is not fixed.

3 Formal Specification in Higher-Order Logic

Functions which accept other functions as arguments or return functions as results are called "higher-order" functions. This property often leads to function definitions which are both simple and direct. Higher-order logic is a formalism for reasoning about functions; this includes higher-order functions as well as relations, *i.e.* functions that return Boolean values.

The behaviour of a hardware device can be described by a relation on input and output signals where signals are modelled by functions from discrete time to sampled values. Because signals are modelled by functions, these will be higher-order relations. This is one reason why higher-order logic is a convenient formalism for reasoning about functions and relations modelling hardware devices.

This approach, inspired by [9], can also be used to describe input-output behaviour at the more abstract level of concurrent processes. The synchronous interface and asynchronous memory correspond to physical devices but the discussion in this paper is only concerned with their external behaviour. Their behaviour is described by relations on a set of signals associated with each process. Some of these signals correspond to wires and busses; other signals represent internal states and do not have physical counterparts.

Thus, the behaviour of a single process is a set of relations, or constraints, on output signals in terms of input signals and internal state signals. The behaviour of a network of concurrent processes is obtained by composing the constraints imposed by each process. This is expressed formally by logical conjunction which serves as a behaviour composition operator. Another operator which appears in specification languages such as CCS [20] and CSP [14] is hiding (or restriction). In our higher-order logic approach, we use existential quantification to hide internal signals, *i.e.* signals which are not external ports of the network.

Much of our notation such as "$\forall$" ("*for all*") and "$\exists$" ("*there exists*") should be familiar from ordinary predicate calculus. Higher-order logic also includes function-denoting terms called λ-expressions as well as Hilbert's ε-operator. For example, the λ-expression "λx. x + 1" denotes the successor function. The term "εx. P x" denotes a value satisfying the predicate P if such a value exists; otherwise, it denotes an arbitrary value of the appropriate type. For example, "εx. x < 10" denotes some natural number less than ten but "εx. x < 0" denotes an arbitrary natural number since the predicate "less than zero" cannot be satisfied in the natural numbers.

In addition to standard logical connectives, the language provides conditional expressions, "b $\Rightarrow$ t1 | t2", which may be read as "*if* b *then* t1 *else* t2" and n-tuples of the form "(t1, ... tn)". Certain features of the language have a special syntactic status for improved readability, *e.g.* the definition of infix func-

tions. The language also includes basic arithmetic functions and relations such as +, -, $<$ and $\leq$.

4 Signal Types

Every term in higher-order logic has a type which is either a primitive type (*e.g.* Booleans, natural numbers) or built-up from existing types using type constructors such as Cartesian product. Every compound term must correctly type-check; for example, a function from numbers to Booleans can only be applied to a number. This section describes function types used to represent various signals in the synchronous interface and asynchronous memory.

A single wire is modelled by a function which maps every point in discrete time to either `Hi` or `Lo`. For instance, the function f models a signal which is low when sampled at times t and $t+3$ and high at times $t+1$ and $t+2$.

$$\ldots\ f(t) = \texttt{Lo},\ f(t+1) = \texttt{Hi},\ f(t+2) = \texttt{Hi},\ f(t+3) = \texttt{Lo},\ \ldots$$

`Hi` and `Lo` are logical constants denoting a pair of distinct values of a type `val`. These constants could be synonyms for true and false in a simple Boolean model of circuit behaviour or they could denote signal values in a more complex circuit model. For simplicity, we assume that `Hi` and `Lo` are the only two values of the type `val`; a similar effect can be achieved for a larger set of values with "well-definedness" assumptions; for example, see [5] or [18].

A simple model is also used for n-bit word values, namely natural numbers, ignoring the fact that n-bit words are finite which is not important in this example. Hence, a bus is modelled by a function from discrete time to the natural numbers. A third type of signal represents a memory state as it varies over time; this is modelled by a function from discrete time to a memory state which, in turn, is modelled by a function from addresses to n-bit word values (this is another use of higher-order functions). Addresses are also modelled by natural numbers ignoring the fact that memories are finite which, once again, is not important in our example. These conventions are summarized by the following type abbreviations where *ty1*→*ty2* denotes the type of all total functions with arguments of type *ty1* and results of type *ty2*.

```
   val = {Hi,Lo}
  wire = time→val
   bus = time→number
memory = time→(number→number)
```

We also introduce a type abbreviation for Boolean signals which are functions from discrete time to Boolean values. These may be thought of as virtual signals testing for specific logical conditions. For example, a Boolean signal could be defined to test when a particular wire is high. In the formal specification of

the four-phase handshaking protocol, Boolean signals are often used to detect synchronization points such as when a memory request is signalled. We sometimes refer to Boolean signals as event signals or sampling functions.

```
boolsig = time→boolean
```

Some predicates such as `Stable` (see next section) are used to describe signals of more than one type. A wire, bus, memory or even a logical condition can all be described as being stable over a specified interval of time. Type variables provide the logic with a limited amount of polymorphism. The following type abbreviation is parameterized by a type variable α to describe the type of a polymorphic signal. Predicates such as `Stable` are defined in terms of this polymorphic type.

```
α sig = time→α
```

5 Temporal Predicates

Reasoning about asynchronous behaviour involves reasoning about the behaviour of signals over intervals of time. This section introduces some higher-order predicates which make it easier to describe conditions which involve one or more instants of time.

`IsHi` and `IsLo` are curried functions, that is, functions which take their arguments 'one at a time'. When applied to a wire signal, *e.g.* "`IsHi w`", these functions return Boolean signals detecting when the wire is high (or low). When applied to both a wire signal and time value, *e.g.* "`IsHi w t`", these functions return Boolean values indicating whether the wire is high (or low) at the specified time.

Definition 5.1:
```
⊢ IsHi (w:wire) (t:time) = (w t = Hi)
```

Definition 5.2:
```
⊢ IsLo (w:wire) (t:time) = (w t = Lo)
```

`Rises` and `Falls` are also curried functions. These functions are used to define Boolean signals which detect transitions from low to high and *vice versa* as shown in Figure 1. For example, a wire rises at time `t` if it is low at time `t` and then high at time `t+1`.

Definition 5.3:
```
⊢ Rises (w:wire) (t:time) = ((w t = Lo) ∧ (w (t+1) = Hi))
```

Definition 5.4:
```
⊢ Falls (w:wire) (t:time) = ((w t = Hi) ∧ (w (t+1) = Lo))
```

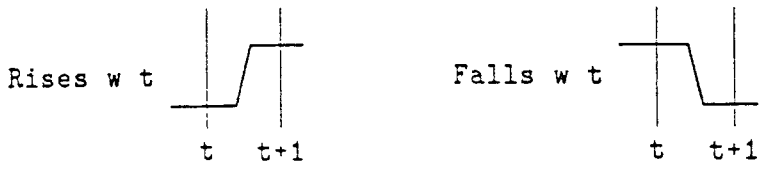

Figure 1: Rising and Falling Signals

Definition 5.5 shows the definition of stability during an interval. This definition takes advantage of polymorphic types provided in the HOL logic to define a predicate which will serve for all types of signal values.

Definition 5.5:

```
⊢ Stable (s:α sig,v:α,t1:time,t2:time) =
    ∀t. t1 ≤ t ∧ t < t2 ⟹ (s t = v)
```

Because asynchronous delays are not fixed, we often need to express the condition that a signal remains stable at a given value until a specified event occurs. If the event never occurs, then the signal remains stable forever. We define two predicates, `StableUntil1` and `StableUntil2`, which capture slightly different interpretations of this informal notion. Polymorphic types are also used here so that these predicates can be used for all types of signals.

Definition 5.6:

```
⊢ StableUntil1 (s:α sig,v:α,t:time,b:boolsig) =
    ∀n. Stable (b,F,t,t+n) ⟹ (s (t+n) = v)
```

Definition 5.7:

```
⊢ StableUntil2 (s:α sig,v;α,t:time,b:boolsig) =
    ∀n. Stable (b,F,t,t+n) ⟹ (s ((t+n)+1) = v)
```

The choice between `StableUntil1` and `StableUntil2` is a matter of fine-tuning when these predicates are used to describe the behaviour of the synchronous interface and the asynchronous memory. The predicate `StableUntil1` refers to an interval which begins immediately and ends exactly when the specified event occurs. `StableUntil2` refers to an interval which begins and ends one time unit later than the interval described by `StableUntil1`. Figure 2 illustrates the intervals described by these two predicates where the specified event occurs at time `t2`. The shaded region indicates the stable interval in each case.

Interaction between the synchronous interface and the asynchronous memory process is often described in terms of the "first" or "next" occurrence of a specified event such as a particular signal rising or falling. The predicate `First` expresses

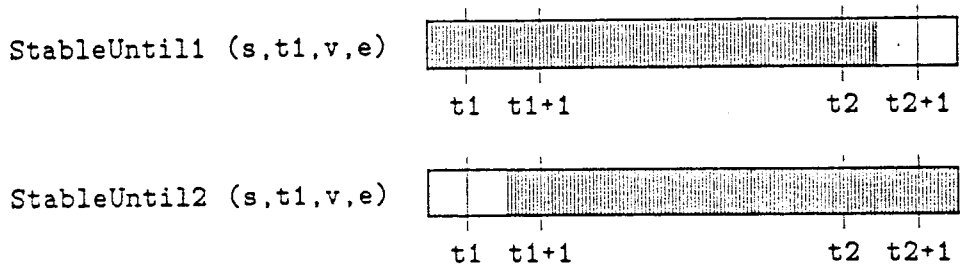

Figure 2: Intervals described by StableUntil1 and StableUntil2

the property that a specified event occurs for the first time after the start of an interval. The predicate Next is similar to the First except that the inequality "<" is used to ensure that the "next" time is strictly after the start of the interval.

Definition 5.8:

$\vdash$ First (t1:time,t2:time) (b:boolsig) =
    (t1 $\leq$ t2) $\wedge$ Stable (b,F,t1,t2) $\wedge$ (b t2)

Definition 5.9:

$\vdash$ Next (t1:time,t2:time) (b:boolsig) =
    (t1 < t2) $\wedge$ Stable (b,F,t1+1,t2) $\wedge$ (b t2)

6 Asynchronous Memory

In this section we develop the formal specification of an asynchronous memory based on the informal specification presented in [15]. As shown in Figure 3, the asynchronous memory has four inputs and two outputs. A read (write) request is signalled by a transition from low to high on the read (write) line. The request is eventually acknowledged by a transition from low to high on the dtack line. A request to end the cycle can then be signalled by a transition from high to low on the read (write) line which is acknowledged by a similar transition from high to low on the dtack line. This four-phase handshaking protocol is shown in Figure 4 for a read request.

Once a read or write request has been signalled, several conditions must be maintained until the asynchronous memory responds or until the end of the cycle. After signalling a read request, the read line must remain high until the request is acknowledged. Furthermore, a write request cannot be signalled during a read cycle; that is, the write line must remain low until the end of the read cycle. Similarly, the address bus must remain stable throughout the cycle. During a write cycle, the write line must remain high until the request is acknowledged and the read line must remain low until the end of the cycle. In this case, the

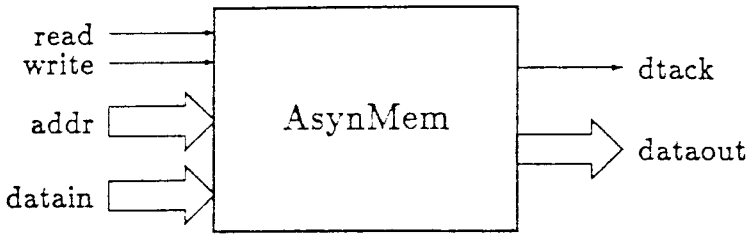

Figure 3: Asynchronous Memory Device

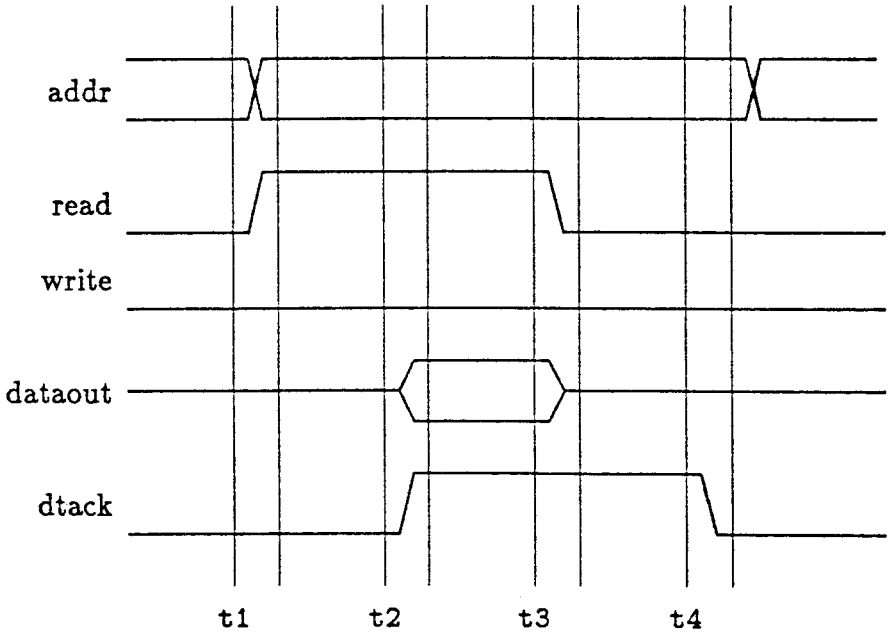

Figure 4: Read Cycle using a Four-Phase Handshaking Protocol. *This simplified timing diagram shows the relative order of events in a read cycle where a request occurs at* `t1` *followed by an acknowledgement at* `t2`. *The acknowledgement may occur in the same interval as the request, i.e.* `t2` *may be equal to* `t1`. *Similarly,* `t4` *may be equal to* `t3`. *However,* `t3` *must occur strictly after* `t1` *and* `t4` *strictly after* `t2` *since signals are not allowed to rise and fall within the same interval.*

data bus as well as the address bus must remain stable throughout the write cycle. For both read and write cycles, a request to end the cycle is signalled by `read` or `write` falling. The next cycle is not allowed to begin until the end of the current cycle is acknowledged by the asynchronous memory; that is, the `read` and `write` lines must remain low until the `dtack` line also falls. Between read and write cycles, when the asynchronous memory is idle, the `dtack` line remains low and the internal state of the memory is stable.

Working from this informal description, the formal specification of the asynchronous memory process can be developed by outlining the overall structure of a memory cycle specification and then filling in precise details. We focus on the behaviour of the asynchronous memory during a read cycle, as illustrated in Figure 4, and begin with a pseudo-formal specification which shows the overall structure of the read cycle specification. Universally quantified time variables denote when the synchronous interface may initiate a request to begin or end a cycle. Existential quantification is used to specify when the asynchronous memory must eventually acknowledge one of these requests.

```
∀t1.
  asynchronous memory idle ∧                              (1)
  read cycle requested at time t1 ∧                       (2)
  ⟹
  ∃t2.
    read request acknowledged at time t2 ∧                (3)
    fetched memory word available as output ∧             (4)
    ∀t3.
      end-of-cycle requested at time t3 ∧                 (5)
      ⟹
      ∃t4.
        end-of-cycle acknowledged at time t4 ∧            (6)
        asynchronous memory returns to idle state ∧       (7)
        internal memory state unchanged during cycle      (8)
```

In the terminology of [21], lines (1) to (2) and (5) specify "domain" constraints which are assumptions about inputs to the asynchronous memory. Lines (3) to (4) and (6) to (8) specify "functional" constraints, that is, outputs and internal state changes which the memory must produce in response to inputs.

Line (1) requires the asynchronous memory to be in an idle state when a read cycle is initiated (it cannot already be in the middle of another memory request). Unfortunately, the asynchronous memory device does not provide a "idle" flag as a physical output signal; while this would simplify the task of writing a formal specification, real memories devices do not usually provide such a signal. Clearly, the value of the `dtack` signal at any particular moment does not indicate when the memory is in an idle state since it can remain low for some time after a read or write cycle begins. For instance, the `dtack` signal is still low at time `t1+1` in

Figure 4 even though a read cycle has already begun and the memory is no longer idle. Instead, one of the conditions indicating that the memory is idle is that the dtack signal is low and will remain low until either read or write rises. The other condition is that the memory state is also stable until the next read or write request is signalled. These two conditions are easily expressed with the predicate StableUntil1 in Definition 6.2. We also define an infix function OrWhen which combines two event signals using logical "or".

Definition 6.1:

```
⊢ (b1:boolsig) OrWhen (b2:boolsig) = λt. b1 t ∨ b2 t
```

Definition 6.2:

```
⊢ MemIdle (read:wire,write:wire,dtack:wire,mem:memory) t =
    StableUntil1 (dtack,Lo,t,(Rises read) OrWhen (Rises write)) ∧
    StableUntil1 (mem,mem t,t,(Rises read) OrWhen (Rises write))
```

Line (2) refers to the initiation of a read cycle. A read request is signalled by a transition from from Lo to Hi on the read line. Once high, the read signal is required to stay high until the request is acknowledged; otherwise the behaviour of the asynchronous memory is undefined. More precisely, the read signal must remain stable up to and including some point when the dtack line is also high. Furthermore, the write signal must remain low and the address bus stable until the end of the read cycle, *i.e.* until dtack falls. These conditions are expressed formally by the following term.

```
Rises read t1 ∧
StableUntil2 (read,Hi,t1,Rises dtack) ∧
StableUntil2 (write,Lo,t1,Falls dtack) ∧
StableUntil2 (addr,address,t1,Falls dtack)
```

The eventual acknowledgement of the read request is signalled as soon as the dtack signal rises, line (3). Once the read request has been acknowledged, the dtack signal will remain high until the read signal falls. The acknowledgement of the request also means that the contents of memory at the given address are now available on dataout until the read signal falls, line (4). The function constant FETCH is used to represent this memory operation. Another constant, STORE, is used to specify the effect of a write cycle on the internal state of the asynchronous memory (see Appendix A for an example of its use). The memory operations denoted by these two constants are not formally defined here because we never need to reason about the effect of storing and later fetching a value from memory.

```
First (t1,t2) (Rises dtack) ∧
StableUntil1 (dtack,Hi,t2+1,Falls read) ∧
StableUntil1 (dataout,FETCH (mem t1) address,t2+1,Falls read)
```

Line (5) refers to when read falls signalling a request to end the read cycle. Once low, the read signal must remain low until the request is acknowledged by the asynchronous memory.

```
First (t2,t3) (Falls read) ∧
StableUntil2 (read,Lo,t3,Falls dtack)
```

Lines (6) and (7) specify that the asynchronous memory will eventually acknowledge this request by resetting the dtack signal and entering the idle state. Finally, line (8) states that the internal state of the memory will be unchanged from its state at the beginning of the cycle.

```
First (t3,t4) (Falls dtack) ∧
MemIdle (read,write,dtack,mem) (t4+1) ∧
(mem (t4+1) = mem t1)
```

The formal specification of the asynchronous memory behaviour during a read cycle is obtained by replacing the numbered lines of the pseudo-formal specification with these precise details. The predicate AsynMemRead is defined in terms of this specification. The variable address, which denotes the stable value of the addr signal during the read cycle, is included in the outermost quantification.

Definition 6.3:

```
⊢ AsynMemRead (
    read:wire,write:wire,addr:bus,datain:bus,
    dtack:wire,dataout:bus,
    mem:memory) =
    ∀ t1 address.
      Rises read t1 ∧
      MemIdle (read,write,dtack,mem) t1 ∧
      StableUntil2 (read,Hi,t1,Rises dtack) ∧
      StableUntil2 (write,Lo,t1,Falls dtack) ∧
      StableUntil2 (addr,address,t1,Falls dtack)
      ⟹
      ∃t2.
        First (t1,t2) (Rises dtack) ∧
        StableUntil1 (dtack,Hi,t2+1,Falls read) ∧
        StableUntil1 (
          dataout,FETCH (mem t1) address,t2+1,Falls read) ∧
        ∀t3.
          First (t2,t3) (Falls read) ∧
          StableUntil2 (read,Lo,t3,Falls dtack)
          ⟹
          ∃t4.
            First (t3,t4) (Falls dtack) ∧
            MemIdle (read,write,dtack,mem) (t4+1) ∧
            (mem (t4+1) = mem t1)
```

The formal specification of a write cycle is very similar to the above specification of the read cycle with the roles of `read` and `write` exchanged. There is also the extra condition that the data bus must remain stable throughout the write cycle. The formal specification uses the function constant `STORE` to specified that the memory state is updated as expected by the end of the write cycle. These two specifications are then combined to define a predicate, `AsynMem`, which denotes the behaviour of the asynchronous memory process. The formal specification also states that the memory will be initially idle at time `0`. The complete specification is given in Appendix A.

7 Synchronous Interface

In this section we formally specify the behaviour of a synchronous interface to the memory device described in the previous section. To focus our discussion, we ignore implementation issues, *i.e.* the interconnection of clocked registers and control logic, and specify the behaviour of the synchronous interface in terms of an abstract state machine. The derivation of abstract state machine behaviour from

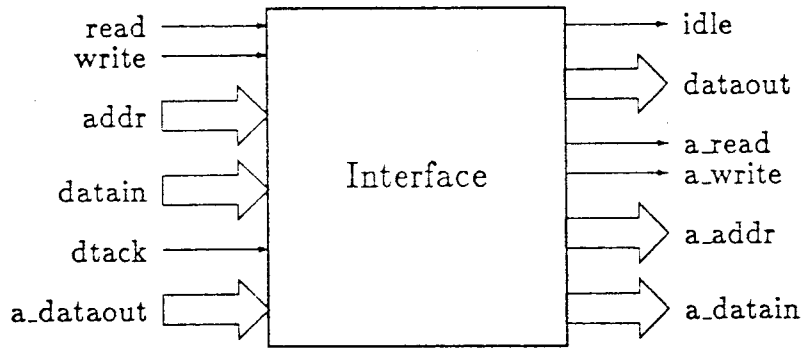

Figure 5: Synchronous Interface Device

an implementation has been demonstrated for several other examples including those described in [3] and [17].

As shown in Figure 5, the interface has four synchronous inputs and two synchronous outputs. The synchronous signals will eventually be the only externally visible signals when the interface is combined with the asynchronous memory to implement a synchronous memory system. The interface receives two asynchronous inputs from the memory and communicates requests and data to the memory through the four asynchronous outputs. To easily distinguish between synchronous and asynchronous signals with a common name, we adopt the naming convention of prefixing asynchronous signals with an "a", *e.g.* `read` and `a_read`.

The synchronous interface waits in an idle state until the `read` or `write` signals becomes high. As soon as this happens, the synchronous address and data busses are latched and the synchronous interface begins either a read or write cycle. After signalling a request to the asynchronous memory, the interface waits some number of clock ticks until the asynchronous memory responds. As soon as the request is acknowledged, the interface immediately resets the `a_read` or `a_write` line and waits one or more clock ticks until the asynchronous memory terminates the memory cycle. In the case of a read cycle, the value of `a_dataout` is latched just before resetting the `a_read` line. At the end of the memory cycle, the interface returns to the idle state.

We begin with a PASCAL-like notation to describe the implementation of the handshaking protocol in the synchronous interface. The wait states are implemented by repeat-until constructs where each iteration corresponds to a single clock tick. Assignment statements are used to specify both current outputs and updates to the internal state of registers in the interface. This algorithm is illustrated by the flow graph of Figure 6.

```
state0: idle := Hi;
        a_read := Lo;
        a_write := Lo;
        repeat_each_clock_tick {
          a_addr := addr;
          a_datain := datain
        } until ((read = Hi) or (write = Hi));
        if (read = Hi) goto state1 else goto state2;

state1: idle := Lo;
        a_read := Hi;
        repeat_each_clock_tick {
          dataout := a_dataout
        } until (dtack = Hi);
        goto state3;

state2: idle := Lo;
        a_write := Hi;
        repeat_each_clock_tick {
        } until (dtack = Hi);
        goto state3;

state3: a_read := Lo;
        a_write := Lo;
        repeat_each_clock_tick {
        } until (dtack = Lo);
        goto state0;
```

The behaviour implied by this algorithm is formally specified in higher-order logic by Definition 7.1. Assignment statements are replaced by output equations, *e.g.* "`idle t = Hi`", and next state equations, *e.g.* "`a_addr (t+1) = addr t`". Unlike assignment statements in the PASCAL-like notation, these equations must explicitly describe current outputs and next state for every clock tick. In the above "program" there is an implicit notion of state which is updated by assignment statements. This contrasts with our style of writing formal specifications in higher-order logic where state is made explicit. We are using the PASCAL-like notation here as an informal description; the formal semantics of this notation or its relationship to our formal specifications are not considered.

At this point we should clarify the distinction between two uses of the word "state". In the more general sense, we use this word to describe the current contents of all storage devices in the implementation of a process including both storage of data, *e.g.* contents of registers, and storage of control information, *e.g.* the program counter in a microprocessor. We also use the word "state" to refer

very specifically to control points in a sequential process such as the four nodes of the flow graph in Figure 6. In the abstract state machine, the sequence of control points is represented by a function from discrete time to the numbers, 0, 1, 2 and 3, used to label the nodes of the flow graph. Even though we already have a type abbreviation for signals of this type, bus, we introduce a new type abbreviation, counter, to emphasis that the state signal does not necessarily have a physical counterpart.

```
counter = time→number
```

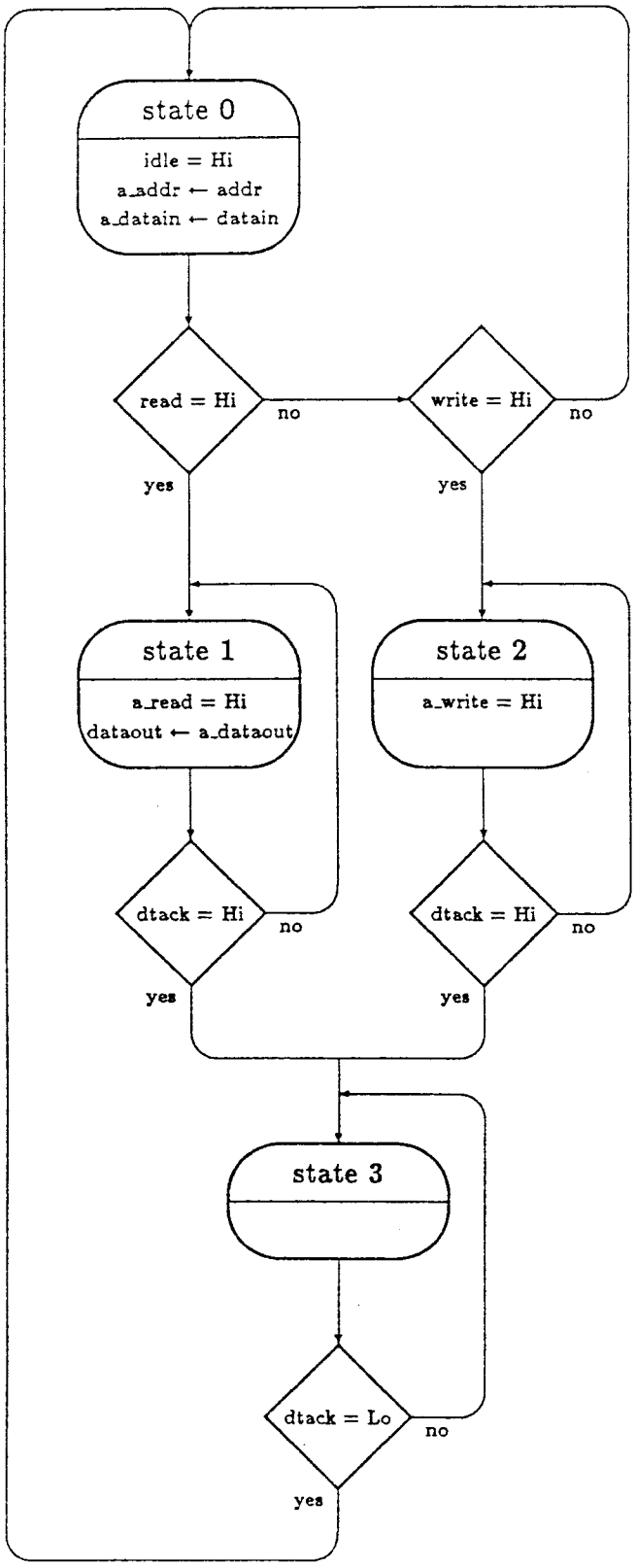

Figure 6: Flow Graph for the Synchronous Interface

Definition 7.1:

```
⊢ Interface (
    read:wire,write:wire,dtack:wire,
    addr:bus,datain:bus,a_dataout:bus,
    idle:wire,a_read:wire,a_write:wire,
    a_addr:bus,a_datain:bus,dataout:bus,
    state:counter) =
    (state 0 = 0) ∧
    ∀t.
      ((state t = 0) ⟹
        (idle t = Hi) ∧
        (a_read t = Lo) ∧
        (a_write t = Lo) ∧
        (a_addr (t+1) = addr t) ∧
        (a_datain (t+1) = datain t) ∧
        (dataout (t+1) = dataout t) ∧
        (state (t+1) =
          ((read t = Hi) ⇒ 1 | (write t = Hi) ⇒ 2 | 0))) ∧
      ((state t = 1) ⟹
        (idle t = Lo) ∧
        (a_read t = Hi) ∧
        (a_write t = Lo) ∧
        (a_addr (t+1) = a_addr t) ∧
        (dataout (t+1) = a_dataout t) ∧
        (state (t+1) = ((dtack t = Hi) ⇒ 3 | 1))) ∧
      ((state t = 2) ⟹
        (idle t = Lo) ∧
        (a_read t = Lo) ∧
        (a_write t = Hi) ∧
        (a_addr (t+1) = a_addr t) ∧
        (a_datain (t+1) = a_datain t) ∧
        (dataout (t+1) = dataout t) ∧
        (state (t+1) = ((dtack t = Hi) ⇒ 3 | 2))) ∧
      ((state t = 3) ⟹
        (idle t = Lo) ∧
        (a_read t = Lo) ∧
        (a_write t = Lo) ∧
        (a_addr (t+1) = a_addr t) ∧
        (a_datain (t+1) = a_datain t) ∧
        (dataout (t+1) = dataout t) ∧
        (state (t+1) = ((dtack t = Lo) ⇒ 0 | 3)))
```

Definition 7.1 also specifies that the initial state of the machine at time 0 is state 0 which is the idle state.

8 Formal Verification in Higher-Order Logic

The main emphasis of this paper is the use of higher-order logic as a specification language for asynchronous processes. However, higher-order logic is also a powerful means of reasoning about these specifications.

An interactive theorem-proving environment for higher-order logic is provided by the HOL system [10], a descendent of Edinburgh LCF [8]. A user of this system manipulates theorems as data objects in the meta-language. The meta-language is an interactive functional programming language called ML. Initially, only the axioms of higher-order logic exist as data objects. New theorems are generated by a small set of built-in ML functions which correspond to primitive inference rules of higher-order logic. Derived inference rules and powerful proof strategies can be programmed as ML functions to automate most of the minor steps, leaving the user to supply only the main steps in a proof [23]. Several large proofs involving more than a million primitive inference steps have been constructed in this system (for instance, see [4] or [18]).

The direct use of inference rules to generate new theorems from existing theorems is called "forward" proof. However, it is often easier to state the desired theorem as a goal and systematically reduce this goal to simpler and simpler sub-goals until all the sub-goals are proved as theorems. This approach, called "backward" or "goal-directed" proof indirectly uses the inference rules to generate the desired theorem. Both forward and backward proof are supported in the HOL system.

The next three sections describe how higher-order logic can be used to reason about the specifications of the synchronous interface and the asynchronous memory, in particular, the interaction of these two processes. Our discussion focuses on theorems which highlight this process; we ignore details of their formal proof (although they have been generated as theorems in the HOL system).

9 Collapsing Wait States into Single Steps

States 1, 2 and 3 of the state machine implement wait states which occur during read and write cycles. In states 1 and 2 the machine waits until the `dtack` signal becomes high. In state 3 the machine waits until the `dtack` signal becomes low. During waits states, the outputs and internal state of the machine remain stable; hence, waits states can be viewed as single steps between synchronization points in the four-phase handshaking protocol.

Using induction on the length of a wait state, we can derive the behaviour of the machine in these wait states expressed in terms of `StableUntil1`. For example, Theorem 9.1 shows that the signals `state`, `idle`, `a_read`, `a_write` and

a_addr remain stable while waiting in state 1 for an acknowledgement from the asynchronous memory.

Theorem 9.1:

```
⊢ Interface (
    read,write,dtack,addr,datain,a_dataout,
    idle,a_read,a_write,a_addr,a_datain,dataout,state)
    ⟹
    ∀t.
      (state t = 1)
      ⟹
      StableUntil1 (state,1,t,IsHi dtack) ∧
      StableUntil1 (idle,Lo,t,IsHi dtack) ∧
      StableUntil1 (a_read,Hi,t,IsHi dtack) ∧
      StableUntil1 (a_write,Lo,t,IsHi dtack) ∧
      StableUntil1 (a_addr,a_addr t,t,IsHi dtack)
```

Similar results can be derived for the behaviour of the machine in state 2 during a write cycle and for its behaviour in state 3 while waiting for the dtack signal to be reset (see Appendix B). These theorems are used to reason about the interaction of the state machine with the asynchronous memory without further use of induction. Thus, we have factored out the inductive aspects of the verification task at this early stage in the formal proof.

10 Symbolic Simulation

This section explains how a proof technique called "symbolic simulation" is used to derive a more abstract view of interaction between the synchronous interface and the asynchronous memory where synchronization details of the handshaking protocol are hidden.

Figure 7 shows the interaction of the abstract state machine with the asynchronous memory during a read cycle. This timing diagram is consistent with the generalized read cycle shown in Figure 4 but shows how some of the unknowns in the generalized read cycle become fixed when the behaviour of the interacting process is known. In particular, Figure 7 shows that time t3 occurs immediately after time t2 (*i.e.* at time t2+1). Figure 7 also shows related activity in the synchronous interface such as the current state, the idle signal, and the latching of data from the asynchronous memory.

To formally reason about the interaction of the abstract state machine with the asynchronous memory process, we assume that a memory cycle is initiated at an arbitrary point in time, t1, and then use inference rules to derive the state of the two processes at each of the subsequent synchronization points, t2, t3 and t4. The asynchronous memory specification and the derived behaviour of the abstract state

machine during wait states, *e.g.* Theorem 9.1, allows us to regard the intervals between synchronization points as single time steps. That is, the interaction of the two processes is deterministic for this abstract view of time where wait states are collapsed into single time steps.

This proof technique, called "symbolic simulation", has also been used to formally verify higher level aspects of microprocessor systems in [4] and [16], *e.g.* symbolic execution of microcode. The simulation takes the form of a sequence of inference in higher-order logic. It is symbolic because variables are used in place of real data and because all possible asynchronous delays are considered at once. Even though we use the descriptive term "simulation", we emphasis that this technique is formal proof based on the inference rules of higher-order logic.

Symbolic simulation is used to reason about read and write cycles as well as idle cycles (idle cycles occur when there is no interaction between the two processes). Details on symbolic simulation are given in Appendix C; the results of this major proof step are summarized in three theorems corresponding to the three types of memory cycles. Each of these theorems relates the state of the synchronous machine at the end of a memory cycle to its initial state at the beginning of the

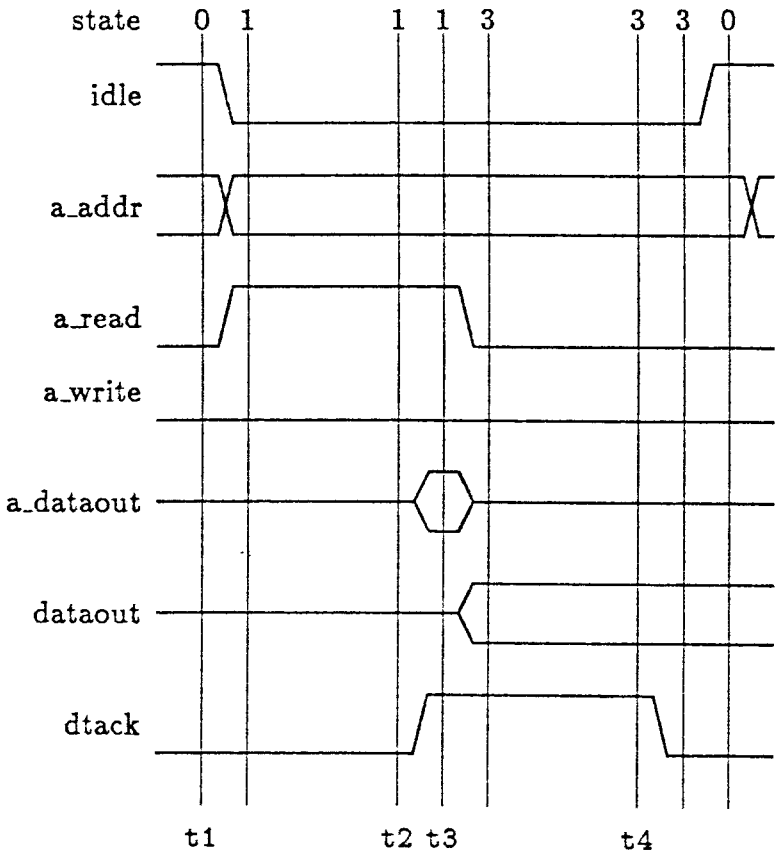

Figure 7: Interaction of State Machine with Asynchronous Memory

cycle. This is illustrated below for a read cycle using pseudo-formal notation.

```
combined behaviour of interface and asynchronous memory      (1)
⟹
∀t1.
  synchronous read request at time t1 ∧                       (2)
  asynchronous memory idle                                    (3)
  ⟹
  ∃t2.
    read cycle ends at t2 ∧                                   (4)
    asynchronous memory returns to idle state ∧               (5)
    fetched memory word available as output ∧                 (6)
    internal memory state unchanged during cycle              (7)
```

As shown above, the overall structure of these three theorems is an implication where the results of symbolic simulation are implied by the combined behaviours of the abstract state machine and the asynchronous memory. The behaviours of these two processes are combined by replacing the formal parameters of `Interface` and `AsynMem` with the names of interconnecting wires and busses and composing the two terms by logical conjunction, line (1).

```
Interface (
  read,write,dtack,addr,datain,a_dataout,
  idle,a_read,a_write,a_addr,a_datain,dataout,state) ∧
AsynMem (a_read,a_write,a_addr,a_datain,dtack,a_dataout,mem)
```

The right-hand side of the outermost implication, lines (2) to (7), states that a read cycle initiated at time `t1` will eventually complete at time `t2`. The existentially quantified variable `t2` denotes the end of the read cycle and should not be confused with time `t2` in Figure 7.

Lines (2) and (3) refer to the initiation of a read cycle. This occurs when the synchronous read line is high, the abstract state machine is in state 0 and the asynchronous memory is idle.

```
(state t1 = 0) ∧
IsHi read t1 ∧
MemIdle (a_read,a_write,dtack,mem) t1
```

If these conditions are satisfied, symbolic simulation shows that read cycle will eventually be completed at time `t2`. The end of the cycle is formally specified as the time when the `idle` signal of the state machine next becomes high after the start of the cycle at time `t1`, line (4). The predicate `Next` is used to formally state this condition.

```
Next (t1,t2) (IsHi idle)
```

Symbolic simulation also shows that the asynchronous memory will return to its idle state by the end of the read cycle and that the fetched memory word will be available as output, lines (5) and (6). Finally, line (7) states that the internal state of the memory will be unchanged from the start of the read cycle.

```
MemIdle (a_read,a_write,dtack,mem) t2 ∧
(dataout t2 = FETCH (mem t1) (addr t1)) ∧
(mem t2 = mem t1)
```

Thus, the results of symbolic simulation for a read cycle are formally expressed by the following theorem.

Theorem 10.1:

```
⊢ ∀ read write dtack addr datain a_dataout
    idle a_read a_write a_addr a_datain dataout state mem.
    Interface (
      read,write,dtack,addr,datain,a_dataout,
      idle,a_read,a_write,a_addr,a_datain,dataout,state) ∧
    AsynMem (a_read,a_write,a_addr,a_datain,dtack,a_dataout,mem)
    ⟹
    ∀t1.
      (state t1 = 0) ∧
      IsHi read t1 ∧
      MemIdle (a_read,a_write,dtack,mem) t1
      ⟹
      ∃t2.
        Next (t1,t2) (IsHi idle) ∧
        MemIdle (a_read,a_write,dtack,mem) t2 ∧
        (dataout t2 = FETCH (mem t1) (addr t1)) ∧
        (mem t2 = mem t1)
```

The above theorem states that the synchronous interface and asynchronous memory correctly implement the four-phase handshaking protocol in the case of a read cycle. Similar theorems can be derived for a write cycle and for a idle cycle (see Appendix C). These three theorems are relatively compact because they do not contain details of synchronization during memory cycles. This is desirable because we wish to derive a more abstract view of interaction between the synchronous interface and the asynchronous memory where synchronization details are hidden.

11 A Synchronous Memory System

Figure 8 shows a synchronous memory system implemented by the synchronous interface and the asynchronous memory device. This implementation is formally specified in Definition 11.1 using logical conjunction to compose behaviours and existential quantification to hide the internal asynchronous signals.

Definition 11.1:

```
⊢ SynMem_Imp (
    read:wire,write:wire,addr:bus,datain:bus,
    idle:wire,dataout:bus,mem:memory) =
    ∃ dtack a_dataout a_read a_write a_addr a_datain state.
      Interface (
        read,write,dtack,addr,datain,a_dataout,
        idle,a_read,a_write,a_addr,a_datain,dataout,state) ∧
      AsynMem (a_read,a_write,a_addr,a_datain,dtack,a_dataout,mem)
```

The behaviour of this implementation is described at an abstract time scale where a memory cycle has a uniform length of one time unit. Thus, read and write requests issued at time `t` are completed by time `t+1`. The following definition specifies the behaviour of the synchronous memory system with respect to this abstract time scale.

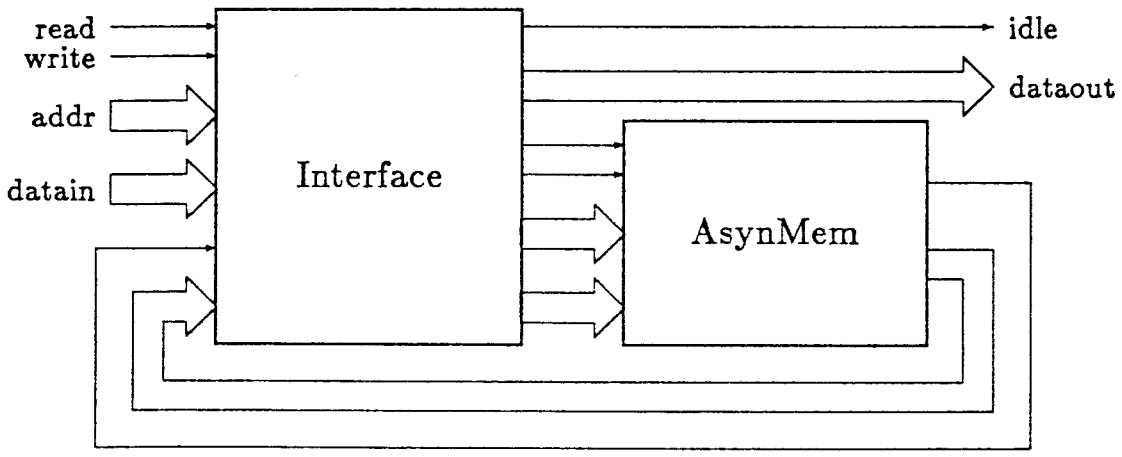

Figure 8: Synchronous Memory System Implementation

Definition 11.2:

```
⊢ SynMem_Beh (
    read:wire,write:wire,addr:bus,datain:bus,
    dataout:bus,mem:memory) =
    ∀t.
      (dataout (t+1),mem (t+1)) =
        IsHi read t  ⇒ (FETCH (mem t) (addr t),mem t) |
        IsHi write t ⇒ (dataout t,
                        STORE (mem t) (addr t) (datain t)) |
                       (dataout t,mem t)
```

The above specification describes the synchronous memory system using an equation for the state of the system at time `t+1` in terms of its inputs and state at time `t`. The right hand of the equation is a nested conditional expression testing first whether the `read` signal is high and then whether the `write` signal is high. A simultaneous request on the `read` and `write` signals defaults to a read request. The state is represented as a pair consisting of `dataout` and the internal memory state `mem`.

The behaviours of the synchronous interface and the asynchronous memory are stated with respect to the clocking of synchronous components in the interface. To prove that the combined behaviour of the interface and the asynchronous memory correctly implement the synchronous memory system, a formal relationship between the concrete time scale and the abstract time scale needs to be established.

Both [12] and [22] describe the use of higher-order functions to relate different time scales. [22] describes a higher-order function `TimeOf` which is used to construct a temporal abstraction from a concrete time scale to an abstract time scale in terms of a sampling function. `TimeOf` can be defined in terms of the predicates `First` and `Next` using primitive recursion and Hilbert's ε-operator. This definition is equivalent to the definition given in [22] under the validity condition discussed shortly.

Definition 11.3:

```
⊢ (TimeOf (b:boolsig) 0 = εt. First (0,t) b) ∧
  (TimeOf (b:boolsig) (n+1) = εt. Next (TimeOf b n,t))
```

The first point on the abstract time scale corresponds to the first time that the sampling function `b` is true with respect to the concrete time scale. Subsequent points on the abstract time scale are defined recursively. The n+1th point on the abstract time scale corresponds to the next time that the sampling function becomes true after the nth time point.

In this example, `TimeOf` is applied to the sampling function "`IsHi idle`" to produce an abstraction which maps the nth point on the abstract time scale to the nth occurrence of "`IsHi idle`" on the concrete time scale. The use of `idle`

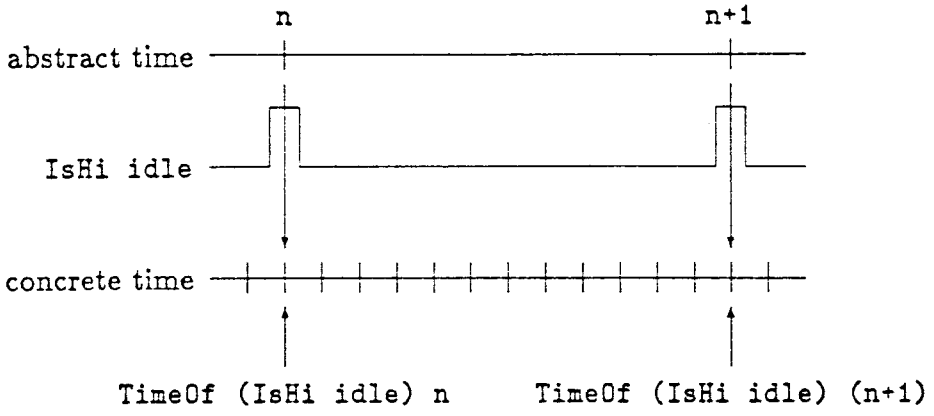

Figure 9: Relationship between Abstract and Concrete Time Scales

to define the formal relationship between the abstract and concrete time scales reflects the fact that this signal is high exactly at the beginning and end of every memory cycle. Figure 9 illustrates the relationship between these two time scales based on the sampling function "IsHi idle".

The validity of the temporal abstraction relating abstract and concrete time scales depends upon showing that the sampling function "IsHi idle" is true infinitely often. The predicate Inf is defined to express this condition.

Definition 11.4:
$\vdash$ Inf (b:boolsig) = $\forall$t1. $\exists$t2. t1 < t2 $\wedge$ (b t2)

For some examples of temporal abstraction such the one described in [22], the external environment is assumed to satisfy the validity condition that the sampling function is true infinitely often. In other examples such as the verification of the Tamarack microprocessor [16], this validity condition is satisfied by the implementation. In the synchronous memory system, the validity of the temporal abstraction is also satisfied by the implementation. The following theorem states general conditions for showing that a sampling function is true infinitely often.

Theorem 11.1:
$\vdash \forall$ g h.
(∃t. g t ∧ h t) ∧
(∀t1. g t1 ∧ h t1 ⟹ ∃t2. Next (t1,t2) g ∧ h t2)
⟹
Inf g

In Theorem 11.1, the variable g is a sampling function and h represents any other conditions which must be propagated from every abstract point in time to

the next. In some cases, there are no such conditions and h can be eliminated from Theorem 11.1. In other cases, the successful completion of an abstract interval of time depends on some conditions holding at the start of the interval. For example, to show that the sampling function "IsHi idle" is true infinitely often, we need to show that the asynchronous memory is always idle at the beginning of an abstract interval. Thus, to show that the sampling function "IsHi idle" is true infinitely often, we first prove the following theorems.

Theorem 11.2:

```
⊢ ∃t. IsHi idle t ∧ MemIdle (a_read,a_write,dtack,mem) t
```

Theorem 11.3:

```
⊢ ∀t1.
    IsHi idle t1 ∧ MemIdle (a_read,a_write,dtack,mem) t1
    ⟹
    ∃t2.
      Next (t1,t2) (IsHi idle) ∧
      MemIdle (a_read,a_write,dtack,mem) t2)
```

The first theorem, asserting that there exists some time when the interface and asynchronous are both idle, is satisfied at time 0. The other condition, that memory cycles always end with the interface and asynchronous memory in an idle state, follows from the results of symbolic simulation described in the previous section. By Theorem 11.1, the sampling function "IsHi idle" is true infinitely often.

Theorem 11.4:

```
⊢ ∀ read write dtack addr datain a_dataout
    idle a_read a_write a_addr a_datain dataout state mem.
    Interface (
      read,write,dtack,addr,datain,a_dataout,
      idle,a_read,a_write,a_addr,a_datain,dataout,state) ∧
    AsynMem (a_read,a_write,a_addr,a_datain,dtack,a_dataout,mem)
    ⟹
    Inf (IsHi idle)
```

The logical significance of Theorem 11.4 is that there is always a "next" abstract time point to be selected by the ε-operator in the definition of `TimeOf`. This is described as a validity condition for temporal abstraction since many useful theorems about `TimeOf` depend on this result. Theorem 11.4 is also significant because it states a "liveness" property for the concurrent system implemented by the two processes.

Using Theorem 11.4 and some of the temporal abstraction theorems described

in [22], we can prove that the asynchronous memory will be idle at every point on the abstract time scale. Therefore, every abstract time point corresponds to a concrete time when a read or write cycle can begin.

Theorem 11.5:

```
⊢ ∀ read write dtack addr datain a_dataout
    idle a_read a_write a_addr a_datain dataout state mem.
    Interface (
      read,write,dtack,addr,datain,a_dataout,
      idle,a_read,a_write,a_addr,a_datain,dataout,state) ∧
    AsynMem (a_read,a_write,a_addr,a_datain,dtack,a_dataout,mem)
    ⟹
    ∀n. MemIdle (a_read,a_write,dtack,mem) (TimeOf (IsHi idle) n)
```

The temporal abstraction relating the abstract time scale to the concrete time scale results in a relationship between the implementation signals which are defined in terms of the concrete time scale and a corresponding set of abstract signals which are defined in terms of the abstract time scale. An abstract signal is the sequence of values obtained when the corresponding concrete signal is sampled at abstract time points. [22] defines an infix function which produces the abstract signal corresponding to a concrete signal.

Definition 11.5:

```
⊢ (s:α sig) when (b:boolsig) = λn. s (TimeOf b n)
```

At this point we are in a position to formulate the top-level statement of correctness for the synchronous memory system. This will state that the constraints imposed on the externally visible signals by the implementation will imply those constraints expressed by the behavioural specification when the implementation signals are sampled according the sampling function "IsHi idle". Recall that SynMem_Imp and SynMem_Beh are the implementation and behavioural specifications of the synchronous memory system. SynMem_Imp imposes constraints on implementation signals which are defined in terms of the concrete time scale whereas SynMem_Beh imposes constraints on signals defined in terms of the abstract time scale. We introduce a higher-order function SamplingWhen which produces constraints on concrete signals corresponding to the constraints imposed by SynMem_Beh on the abstract signals.

Definition 11.6:

```
⊢ SamplingWhen (b:boolsig) behaviour_predicate
    (s1:wire,s2:wire,s3:bus,s4:bus,s5:bus,s6:memory) =
    behaviour_predicate (
      s1 when b,s2 when b,s3 when b,s4 when b,s5 when b,s6 when b)
```

Using this function, the top-level correctness statement for the synchronous memory system is expressed by Theorem 11.6.

Theorem 11.6:

```
⊢ ∀ read write addr datain idle dataout mem.
    SynMem_Imp (read,write,addr,datain,idle,dataout,mem)
    ⟹
    SamplingWhen (IsHi idle)
      SynMem_Beh (read,write,addr,datain,dataout,mem)
```

Until now, we have not described proof details for any of the theorems presented in this paper. However, it is worthwhile outlining the formal proof of Theorem 11.6 since it is a good example of one way in which the HOL system is used to prove theorems. Using backward proof, the conclusion of Theorem 11.6 (*i.e.* the term following the "⊢") is set up as the top-level goal. After expanding the top-level goal with the definitions of SynMem_Imp, SynMem_Beh, SamplingWhen and when, case analysis is used to split the goal into three sub-goals corresponding to the three types of memory cycles, *i.e.* read, write and idle cycles. The results of symbolic simulation, *e.g.* Theorem 10.1, are then used to show that the implementation of the synchronous memory system satisfies its behavioural specification for each of the three cases.

12 Independent Process Specification

One of the main differences between the specifications described in this paper and the approach taken in [15] is that here each process has an independent specification. In particular, the asynchronous memory specification is written as an independent definition in higher-order logic. Thus, it is possible to reason independently about each process; it is only necessary to compose specifications in order to reason about their interaction. This contrasts with Hunt's functional specification style where "the characterization of external devices are wrapped up in the same function which specifies the microprocessor" [1].

The relational style of specification used in our approach makes it easy to write independent specifications for each process and compose them using logical conjunction. The main problem with the functional style of specification is modelling bi-directional communication between two processes. [11] and [13] describe functional specification styles which solve this problem using lazy evaluation techniques.

With independent specifications for each process, it is possible to prove that particular implementations of each process correctly implement these specifications. To illustrate this point, Definition 12.1 describes a particular implementa-

[1] [15], page 113.

tion of the asynchronous memory process. The behaviour of this implementation is described by an abstract state machine using a specification style similar to the specification of the synchronous interface. For simplicity, we assume that this specification shares a common time scale with the implementation of the synchronous interface.

Definition 12.1:

```
⊢ Memory (
    read:wire,write:wire,addr:bus,datain:bus,
    dtack:wire,dataout:bus,
    mem:memory,state:counter) =
     (state 0 = 0) ∧
     ∀t.
       ((state t = 0) ⟹
         (dtack t = Lo) ∧
         (mem (t+1) = mem t) ∧
         (state (t+1) =
           ((read t = Hi) ⇒ 1 | (write t = Hi) ⇒ 2 | 0))) ∧
       ((state t = 1) ⟹
         (dtack t = Hi) ∧
         (dataout t = FETCH (mem t) (addr t)) ∧
         (mem (t+1) = mem t) ∧
         (state (t+1) = ((read t = Hi) ⇒ 1 | 0))) ∧
       ((state t = 2) ⟹
         (dtack t = Hi) ∧
         (mem (t+1) = STORE (mem t) (addr t) (datain t)) ∧
         (state (t+1) = ((write t = Hi) ⇒ 2 | 0)))
```

13 Independent Time Scales

The formal specifications of the asynchronous memory process and the synchronous interface are expressed in terms of a common time scale, namely, the clocking rate of sequential components in the synchronous interface. This common time scale is an implicit form of synchronization which underlies the asynchronous interaction implemented by the four-phase handshaking protocol.

To clarify this situation, we distinguish between the time scale of the formal specification and the use of clocked components (if any) in an implementation of the asynchronous memory process. The formal specification describes the behaviour of the asynchronous memory process when observed in terms of the clocking rate of the synchronous interface. It is possible that the asynchronous memory is implemented by components clocked by the same clock used in the synchronous interface (such as the implementation described in the previous section). However, the formal specification of the asynchronous memory process could also specify the

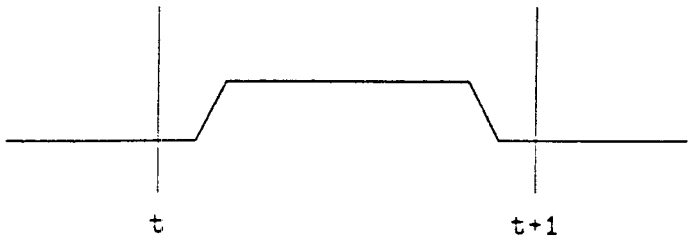

Figure 10: A Signal which is Not "Well-Behaved"

observed behaviour of a memory with an independent clock or a memory implemented by unclocked circuits.

Hence, our specifications use a common time scale even though the corresponding devices may involve independent clocks. We have not used formal methods to relate specifications involving independent time scales. However, we have attempted to write the asynchronous memory specification in a "speed-independent" style where the underlying time scale is unimportant. This specification describes the behaviour of the asynchronous memory process at any granularity of time for which inputs to the memory process are "well-behaved".

A signal is well-behaved with respect to a particular sampling rate if the signal, when viewed continuously, does not rise and fall or vice versa within a single interval of discrete time. For example, Figure 10 shows an input signal which is not well-behaved because it momentarily rises and then falls without the event being detected at the selected sampling rate. This constraint is meant to prevent asynchronous inputs from switching faster than the sampling rate; it also excludes "glitches", pulses with a duration shorter than a technology-dependent threshold, which may have an unknown effect on the asynchronous memory. Both [5] and [19] describe similar conditions constraining how signals may change with respect to discrete time.

When inputs are well-behaved, the formal specification of the asynchronous memory will be satisfied even for a relatively coarse grain of time. We do not assume, for example, that the acknowledgement by the memory must occur strictly after a request since this might be untrue even when the sampling rate is slow relative to the operation of the asynchronous memory. In such a case, the sampled value of the dtack signal may appear to rise simultaneously with the rise of the read signal as shown in Figure 11 even though it really occurs slightly later than the request on the continuous time scale. This is why the specification requires the inequality "$\leq$" to be used in the definition of First instead of "$<$".

The formal specification of the asynchronous memory is also satisfied at an extremely fine grain of time. Even at the granularity of nanoseconds, the formal specification requires that the read and write signals remain stable until the request is recognized by the memory device even if tens of nanoseconds elapse before this happens.

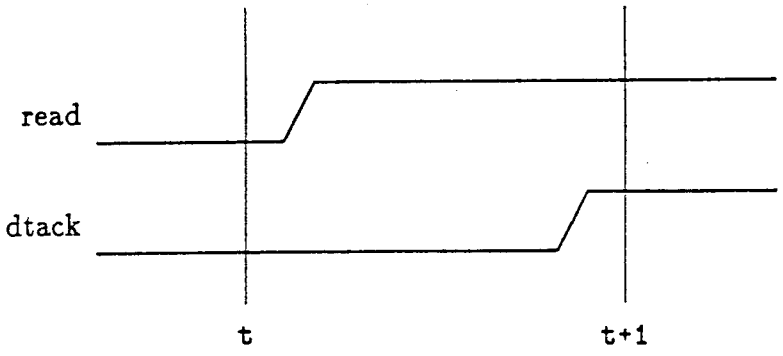

Figure 11: A Relatively Coarse Time Scale

Hence, the time scale in the formal specification of the asynchronous memory is unimportant. Our use of the term "speed-independent" to describe this style of specification alludes to the use of the four-phase handshaking protocol (also called Muller signalling) in self-timed systems [21]. While the scale of time in the asynchronous memmory specification is not important, the specification depends on "open-loop" relations which assume that address and data values are sent, not just concurrently, but in parallel with the request and acknowledgement signals.

14 Real Systems

The four-phase handshaking protocol described in this paper is also the basis for asynchronous data transfers between the M68000 microprocessor and memory or other peripheral devices. Instead of separate signals for read and write requests, the M68000 has an address strobe AS* which signals requests to asynchronous memory. Read requests are distinguished from write requests by another signal, R/W, which is set high or low before a request is signalled by AS*. Data is transferred over a bi-directional bus instead of separate datain and dataout busses.

A simplified timing diagram for a M68000 read cycle is shown in Figure 12. An important difference between this diagram and our model of an asynchronous memory is that the address must be valid before the memory request is signalled by AS*. More detailed timing information for the M68000 states the address must be valid at least 30 nanoseconds before AS* is asserted. This ensures that memory is not written at the wrong address before the correct address has had sufficient time to become stable within the memory. This contrasts with our formal specification of the asynchronous memory which only requires the address to become valid simultaneously with the memory request. We could revise this formal specification to include this more detailed timing requirement by rewriting the specification in terms of a specific time scale, *i.e.* the nanosecond time scale. The specification would explicitly require the address bus to be stable from at least 30 nanoseconds before a read or write request.

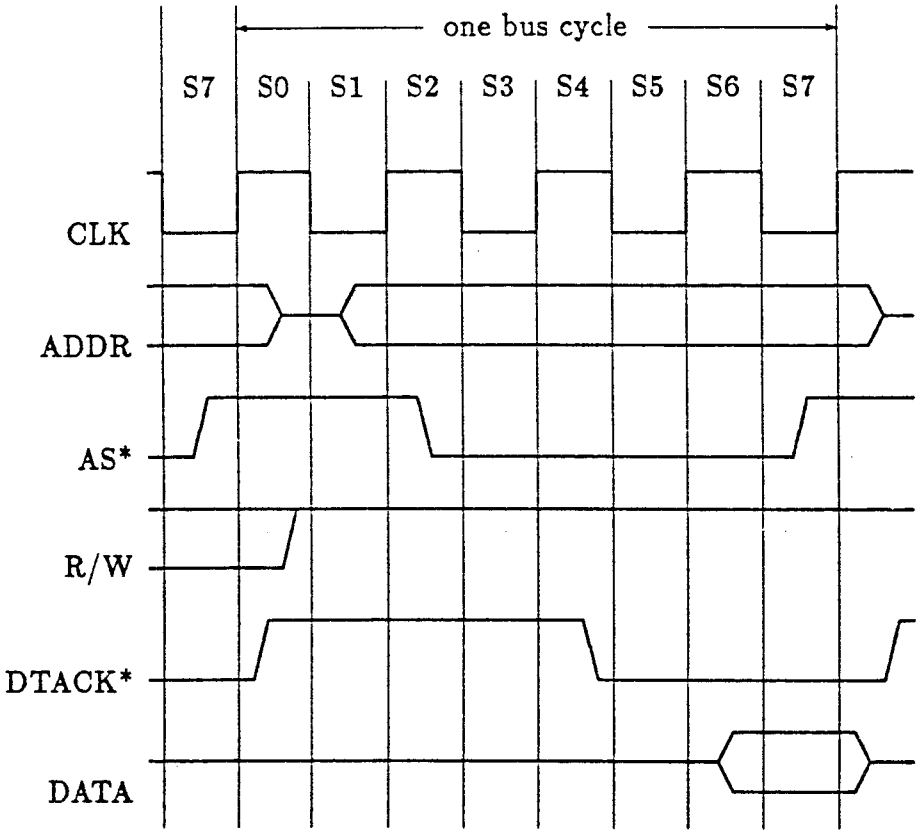

Figure 12: Simplified M68000 Read Cycle. *The synchronous time scale for the M68000 is divided into half cycle intervals because events can occur on both rising and falling edges. Only the address strobe* AS* *is mentioned here. Asterisks indicate "active-low" signals. This timing diagram is adapted from* [2], *page* 121.

To reason about the interaction of the asynchronous memory with the synchronous interface using this revised specification, we need to relate this specification to a synchronous level specification. The timing diagram in Figure 12 shows how the detailed timing level behaviour of the M68000 can be viewed in terms of a time scale divided into half-cycle intervals. Assuming that the latest point at which the address can become stable occurs at least 30 nanoseconds before the earliest possible rise of AS* in the following interval, then the detailed timing requirement is satisfied. Hence, requiring the address to become stable in the preceding half-cycle interval is a synchronous level condition which satisfies the detailed timing requirement that the address becomes stable at least 30 nanoseconds before a read or write request. This synchronous level condition could be used to rewrite the formal specification of the asynchronous memory in terms of the synchronous time scale.

Detailed timing information for the M68000 also states: "If DTACK* is does not go low (active low) at least 20 nanoseconds before the end of state S4, wait states are introduced between S4 and S5 until DTACK* is asserted". In fact, this description, taken directly from [2], is unrealistically precise. The minimum setup time for the acknowledgement signal is a value for which the M68000 is certain to behave in a predictable way. However, if the acknowledgement is asserted some very small fraction of a nanosecond under the minimum setup time, it is likely that the microprocessor will still detect the acknowledgement in this interval and not introduce a wait state. Therefore, the above description is not entirely accurate because the assertion of DTACK* less than 20 nanoseconds before the falling edge of the clock does not necessarily result in a wait state.

If DTACK* was a synchronous input to the M68000, then it would be reasonable to disallow the possibility of an acknowledgement occurring under the minimum setup time; in such a case the behaviour of the M68000 would be left undefined. However, asynchronous inputs such as DTACK* should not be expected to satisfy such conditions. Furthermore, a fixed minimum setup time of 20 nanoseconds is not the problem; more complicated conditions would eventually become impractical and at some point even theoretical physics could not be precise.

Instead of attempting to be more precise about the timing requirements, we can simply state that when DTACK* is asserted less than 20 nanoseconds before a clock edge, the microprocessor may or may not introduce a wait state. In other words, the behaviour of the microprocessor is not deterministic in this case, but is it constrained to either introduce a wait state or continue with the rest of the memory cycle.

To illustrate this approach, we revise the formal specification of synchronous interface which models the role of the M68000 in its interaction with asynchronous memory. We could rewrite the formal specification in terms of the nanosecond time scale stating that the behaviour of the interface is only deterministic when the acknowledgement is asserted 20 nanoseconds before the clock edge; however, we would then lose the advantage of being able to reason about the behaviour

of the interface in terms of the synchronous time scale. Instead, we continue to express the behaviour of the interface in terms of the synchronous time scale by ignoring the fact that the behaviour is deterministic when the minimum setup time is satisfied. We state that the behaviour of the interface is non-deterministic when dtack is first asserted in an interval. We only state that the behaviour of the interface is deterministic when the dtack signal is asserted throughout the interval in which case the minimum setup time is clearly satisfied. This behaviour is expressed by the following definition of a function which computes the next value of state at time t+1.

Definition 14.1:

```
⊢ Either (v1:α,v2:α) = εv. (v = v1) ∨ (v = v2)
```

Definition 14.2:

```
⊢ Continuous (s:α sig,t:time) = (s (t-1) = s t)
```

Definition 14.3:

```
⊢ InterfaceStateFun (read:wire,write:wire,dtack:wire,state) t =
    (state t = 0) ⇒ (
      (read t = Hi) ⇒ 1 | (write t = Hi) ⇒ 2 | 0) |
    (state t = 1) ⇒ (
      Continuous (dtack,t) ∧ (dtack t = Hi) ⇒ 3 |
      Continuous (dtack,t) ∧ (dtack t = Lo) ⇒ 1 |
      Either (3,1)) |
    (state t = 2) ⇒ (
      Continuous (dtack,t) ∧ (dtack t = Hi) ⇒ 2 |
      Continuous (dtack,t) ∧ (dtack t = Lo) ⇒ 1 |
      Either (2,1)) |
      Continuous (dtack,t) ∧ (dtack t = Lo) ⇒ 0 |
      Continuous (dtack,t) ∧ (dtack t = Hi) ⇒ 3 |
      Either (0,3)) |
```

The predicate Continuous tests whether a signal is stable throughout an interval with respect to continuous time assuming that the signal is well-behaved. We can assume that the dtack signal is well-behaved with respect to the synchronous time scale because it is constrained by the formal specification of the asynchronous memory to change its value only in response to a change from the synchronous interface; hence, it impossible for the dtack to change its value more than once during a single interval.

Testing whether dtack is stable throughout an interval of synchronous time satisfies the minimum setup time for an acknowledgement since the the length of a synchronous interval is likely to be much longer than 20 nanoseconds. For example, if the interface is in state 1 and dtack is stable throughout the interval at high, then we know for certain that the next state will be 3. If dtack is low throughout

the interval, then we know for certain that the next state will be 1. Otherwise, we cannot determined by synchronous sampling alone whether the minimum setup time was satisfied; therefore, we can only conclude that the next state is either 3 or 1. This non-deterministic choice is specified using the ε-operator in the definition of `Either`. It is significant that only the asynchronous inputs to the interface require this special attention; the signals `read` and `write` are synchronous inputs which are assumed to satisfy minimum setup times. The derivation of synchronous behaviour in this regard is considered in [12].

This non-deterministic specification of the next state function does not allow us to predict the exact length of a wait state. However, it is still possible to prove that the synchronous interface and asynchronous memory correctly implement the synchronous memory system because the behaviour of the synchronous memory system does not depend on the actual length of the wait state.

In practice, microprocessor system designers exploit detailed timing characteristics to interface a microprocessor such as the M68000 to asynchronous memory without the full generality of the four-phase handshaking protocol. For example, [2] describes the use of a HM6116P static RAM with an maximum access time that ensures the completion of a M68000 memory cycle without the introduction of wait states. A similar practice is described for the safety-critical VIPER microprocessor which also uses the four-phase handshaking protocol [24]. In these cases, where is the maximum access time is calculated to be sufficient for a memory cycle to complete without the introduction of wait states, the acknowledgement can be automatically asserted by miscellaneous logic not controlled by the asynchronous memory, *e.g.* wiring `DTACK*` to a voltage source.

15 Future Work

The formal specifications of the synchronous interface and the asynchronous memory are currently expressed in terms of the concrete time scale corresponding to the clocking of sequential circuits. Temporal predicates such as `StableUntil1` and `StableUntil2` are used to describe the behaviour of signals over variable length intervals. These specifications implicitly define an abstract time scale where wait states are collapsed into single time steps. This abstract time scale is an intermediate level of timing; it is coarser than the underlying concrete time scale but finer than the abstract time scale formally defined in Section 11.

In future work, we plan to write specifications directly in terms of this abstract time scale using the temporal abstraction function `TimeOf` to establish a direct relationship between this abstract time scale and the concrete time scale. In this case, the sampling function would detect whenever a relevant signal changes value. This guarantees that these signals are stable during every abstract interval since the interval terminates as soon as any signal changes value.

Writing specifications directly in terms of this abstract time scale should result in considerably "smaller" and more direct specifications. For example, the

term "StableUntil2 (read,Hi,t1,Rises dtack)" is used to express the condition that the read signal must remain high until dtack rises in the current specification of the asynchronous memory. When expressed in terms of the abstract time scale, this condition would simply be "read t1 = Hi"; if some event other than the rise of dtack terminates the abstract interval, then the resulting behaviour of the asynchronous memory is undefined. In addition, to the aesthetic qualities of a smaller and more direct specification, formal reasoning about the interaction of the synchronous interface with the asynchronous memory should be much easier because single steps in the symbolic simulation will correspond to unit intervals on the abstract time scale.

16 Summary and Conclusion

We have modelled the interaction of a synchronous device with an asynchronous memory using a four-phase handshaking protocol. This example demonstrates the use of higher-order logic to reason about the behaviour of synchronous systems such as microprocessors which communicate requests to an asynchronous device and then wait for unpredictably long periods until these requests are answered. We also showed how this behaviour can be formally related to an abstract time scale where wait states correspond to single abstract intervals. When viewed in terms of this abstract time scale, the behaviour of the synchronous device is entirely deterministic. We can then reason about higher level aspects of the synchronous behaviour, *e.g.* the correctness of microcode, without the extra complication of non-determinism. The main features of our approach are summarized in the following points:

- Existential quantification provides a straightforward means of specifying the finite but unknown length of a wait state.
- The use of explicit time references and higher-order functions leads to relatively simple and direct specifications.
- The relational specification style makes it easy to write independent specifications for processes.

In reasoning about the interaction of the synchronous interface with the asynchronous memory, inductive aspects were factored out of the verification procedure at an early stage; we obtained properties about the synchronous interface expressed in terms of the StableUntil1 predicate. We then used forward inference rules to symbolically simulate the interaction of the synchronous interface with the asynchronous memory. The results of symbolic simulation were used to prove that the synchronous interface and asynchronous memory correctly implement a synchronous memory system. The behaviour of the synchronous memory system is expressed at an abstract time scale where read and write requests are completed in single intervals of time.

We also compared our model to real systems such as the M68000 microprocessor. We identified some of the detailed timing requirements in these real systems and suggested how our model could be revised to include these details. In particular, we suggested how non-determinism could be used to specify the effect of asynchronous inputs on synchronous devices.

Although the example described in this paper is strongly oriented towards hardware, we believe that higher-order logic is also a suitable formalism for more abstract forms of concurrency. Furthermore, the expressive power of higher-order logic can be used to represent other formalisms such as temporal logic which are often used to specify concurrent systems. The readability of our specifications may be improved by borrowing notation from other formalisms and our proof strategies made more efficient by deriving special-purpose inference rules based on these formalisms.

Acknowledgements

Mike Gordon and Avra Cohn suggested many improvements to earlier versions of this paper. I am also grateful to Miriam Leeser for helpful discussions on handshaking protocols and their formal specification.

This research has been funded by the Cambridge Commonwealth Trust, the Canada Centennial Scholarship Fund, the Government of Alberta Heritage Fund, the Natural Sciences and Engineering Research Council of Canada and the UK Overseas Research Student Awards Scheme.

References

[1] G. Bochman, "Hardware Specification with Temporal Logic", IEEE Transactions on Computers, C-31 (3), March 1982.

[2] A. Clements, *Microprocessor Systems Design*, PWS Publishers, Boston, 1987.

[3] A. Cohn and M. Gordon, "A Mechanized Proof of Correctness of a Simple Counter", Technical Report No. 94, Computer Laboratory, Cambridge University, July 1986.

[4] A. Cohn, "A Proof of Correctness of the Viper Microprocessor: The First Level", *VLSI Specification, Verification and Synthesis*, Proceedings of the Workshop on Hardware Verification, Calgary, Canada, 12-16 January 1987, G. Birtwistle and P. Subrahmanyam, eds., Kluwer Academic Publishers, Boston, 1988.

[5] I. Dhingra, "Formal Validation of an Integrated Circuit Design Style", *VLSI Specification, Verification and Synthesis*, Proceedings of the Workshop on Hardware Verification, Calgary, Canada, 12-16 January 1987, G. Birtwistle and P. Subrahmanyam, eds., Kluwer Academic Publishers, Boston, 1988.

[6] D. Dill and E. Clarke, "Automatic Verification of Asynchronous Circuits using Temporal Logic", IEE Proceedings, Vol. 133, Pt. E, No. 5, September 1986.

[7] M. Fujita, H. Tanaka and T. Moto-oka., "Temporal Logic Based Hardware Description and Its Verification with Prolog", New Generation Computing, No. 1, 1983.

[8] M. Gordon, R. Milner and C. Wadsworth. *Edinburgh LCF: An Mechanised Logic of Computation*, Lecture Notes in Computer Science, Springer-Verlag, 1979.

[9] M. Gordon, "Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware", *Formal Aspects of VLSI Design*, Proceedings of the 1985 Edinburgh Conference on VLSI, G.J. Milne and P. Subrahmanyam, eds., North-Holland, Amsterdam, 1986.

[10] M. Gordon, "A Proof Generating System for Higher-Order Logic", *VLSI Specification, Verification and Synthesis*, Proceedings of the Workshop on Hardware Verification, Calgary, Canada, 12-16 January 1987, G. Birtwistle and P. Subrahmanyam, eds., Kluwer Academic Publishers, Boston, 1988.

[11] P. Henderson, *Functional Programming*, Prentice-Hall, 1980.

[12] J. Herbert, "Application of Formal Methods to Digital System Design", Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1986.

[13] S. Hill, "Simulating Digital Circuits in Miranda", University of Kent, 1986.

[14] C. Hoare, *Communicating Sequential Processes* Prentice-Hall, 1985.

[15] W. Hunt, "FM8501: A Verified Microprocessor", PhD Thesis, Institute for Computer Science, University of Texas at Austin, 1986.

[16] J. Joyce, G. Birtwistle, and M. Gordon, "Proving a Computer Correct in Higher Order Logic", Technical Report No. 100, Computer Laboratory, Cambridge University, December 1986.

[17] J. Joyce, "Multi-Level Verification of a Simple Microprocessor", Ph.D. Research Progress Report, Computer Laboratory, Cambridge University, December 1987.

[18] J. Joyce, "Formal Specification and Verification of Microprocessor Systems", *EUROMICRO 88*, Proceedings of the 14th Symposium on Microprocessing and Microprogramming, Zurich, Switzerland, 29 August - 1 September, 1988, S. Winter and H. Schumny, eds., North-Holland, Amsterdam, 1988.

[19] M. Leeser, "Reasoning about the Function and Timing of Integrated Circuits with Prolog and Temporal Logic", Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1987.

[20] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Springer-Verlag, 1980.

[21] C. Seitz, "System Timing", Chapter 7 in *Introduction to VLSI Systems*, C. Mead and L. Conway, Addison-Wesley, Reading, Massachusetts, 1980.

[22] T. Melham, "Abstraction Mechanisms for Hardware Verification", *VLSI Specification, Verification and Synthesis*, Proceedings of the Workshop on Hardware Verification, Calgary, Canada, 12-16 January 1987, G. Birtwistle and P. Subrahmanyam, eds., Kluwer Academic Publishers, Boston, 1988.

[23] Paulson, L., *Logic and Computation*, Cambridge University Press, Cambridge, 1987.

[24] C. Pygott, "Electrical, Environmental and Timing Specification of the Viper Microprocessor", Memorandum No. 3753 (Unclassified), RSRE (Royal Signals and Radar Establishment), British Ministry of Defense, December 1984.

Appendix A - Memory Specification

The definition of AsynMemRead in Section 6 specifies the behaviour of the asynchronous memory process during a read cycle. The corresponding specification for a write cycle is shown below. These two specifications are then combined to define AsynMem. In addition to the constraints imposed by AsynMemRead and AsynMemWrite during read and write cycles, AsynMem states that the memory is initially idle at time 0.

```
⊢ AsynMemWrite (
    read:wire,write:wire,addr:bus,datain:bus,
    dtack:wire,dataout:bus,
    mem:memory) =
    ∀ t1 address value.
      Rises write t1 ∧
      MemIdle (read,write,dtack,mem) t1 ∧
      StableUntil2 (read,Lo,t1,Falls dtack) ∧
      StableUntil2 (write,Hi,t1,Rises dtack) ∧
      StableUntil2 (addr,address,t1,Falls dtack) ∧
      StableUntil2 (datain,value,t1,Falls dtack)
      ⟹
      ∃t2.
        First (t1,t2) (Rises dtack) ∧
        StableUntil1 (dtack,Hi,t2+1,Falls write) ∧
        ∀t3.
          First (t2,t3) (Falls write) ∧
          StableUntil2 (write,Lo,t3,Falls dtack)
          ⟹
          ∃t4.
            First (t3,t4) (Falls dtack) ∧
            MemIdle (read,write,dtack,mem) (t4+1) ∧
            (mem (t4+1) = STORE (mem t1) address value))

⊢ AsynMem (
    read:wire,write:wire,addr:bus,datain:bus,
    dtack:wire,dataout:bus,
    mem:memory) =
    MemIdle (read,write,dtack,mem) 0 ∧
    AsynMemRead (read,write,addr,datain,dtack,dataout,mem) ∧
    AsynMemWrite (read,write,addr,datain,dtack,dataout,mem)
```

Appendix B - Wait State Theorems

Theorem 9.1 uses the predicate StableUntil1 to describe the behaviour of the synchronous interface while waiting in state 1. The corresponding theorems for states 2 and 3 are shown below.

```
⊢ Interface (
    read,write,dtack,addr,datain,a_dataout,
    idle,a_read,a_write,a_addr,a_datain,dataout,state)
    ⟹
    ∀t.
      (state t = 2)
      ⟹
      StableUntil1 (state,2,t,IsHi dtack) ∧
      StableUntil1 (idle,Lo,t,IsHi dtack) ∧
      StableUntil1 (a_read,Lo,t,IsHi dtack) ∧
      StableUntil1 (a_write,Hi,t,IsHi dtack) ∧
      StableUntil1 (a_addr,a_addr t,t,IsHi dtack) ∧
      StableUntil1 (a_datain,a_datain t,t,IsHi dtack) ∧
      StableUntil1 (dataout,dataout t,t,IsHi dtack)

⊢ Interface (
    read,write,dtack,addr,datain,a_dataout,
    idle,a_read,a_write,a_addr,a_datain,dataout,state)
    ⟹
    ∀t.
      (state t = 3)
      ⟹
      StableUntil1 (state,3,t,IsLo dtack) ∧
      StableUntil1 (idle,Lo,t,IsLo dtack) ∧
      StableUntil1 (a_read,Lo,t,IsLo dtack) ∧
      StableUntil1 (a_write,Lo,t,IsLo dtack) ∧
      StableUntil1 (a_addr,a_addr t,t,IsLo dtack) ∧
      StableUntil1 (a_datain,a_datain t,t,IsLo dtack) ∧
      StableUntil1 (dataout,dataout t,t,IsLo dtack)
```

Appendix C - Symbolic Simulation

Symbolic simulation is used to investigate the interaction of the synchronous interface with the asynchronous memory process. Each step in the simulation is formally derived from initial assumptions or from previous steps using inference rules of higher-order logic. To illustrate this proof technique, we describe the symbolic simulation of a read cycle beginning at time t1. The time variables, t1, t2, t3 and t4, correspond to the timing diagram in Figure 7 which shows the interaction of these two processes during a read cycle.

We begin by assuming that the state machine is in state 0 at time t1 and that the read signal is high. We also assume that the asynchronous memory is idle at time t1.

```
state t1 = 0

read t1 = high

MemIdle (a_read,a_write,dtack,mem) t1
```

At time t1+1, the state machine enters state 1 and a_read becomes high signalling a read request to the memory.

```
state (t1+1) = 1

Rises a_read t1
```

In state 1, the interface waits for the acknowledgement from the memory. Theorem 9.1 shows that a_read will remain high until dtack becomes high.

```
StableUntil1 (a_read,Hi,t1+1,IsHi dtack)
```

This condition may be rewritten in the form required by the formal specification of the asynchronous memory.

```
StableUntil2 (a_read,Hi,t1,Rises dtack)
```

Next, we need to show that a_write and a_addr remain stable throughout the read cycle, *i.e.* until dtack falls. This follows from the behaviour of the interface and does not depend on responses from the asynchronous memory; if the dtack never rises or rises but never falls, then the following conditions will still be satisfied.

```
StableUntil2 (a_write,Lo,t1,Falls dtack)

StableUntil2 (a_addr,addr t1,t1,Falls dtack)
```

At this point in the symbolic simulation we have established all the conditions required by the formal specification of the asynchronous memory to ensure that the memory will eventually acknowledge the read request at some time t2.

```
First (t1,t2) (Rises dtack)
```

As shown in Figure 7, the interface will detect that the dtack signal is high at time t2+1 and will enter state 3 at time (t2+1)+1. However, before leaving state 1, the interface will latch the fetched memory word which becomes available on a_dataout by at least time t2+1. Note that t2+1 is labelled by t3 in Figure 7.

```
dataout ((t2+1)+1) = FETCH (mem t1) (addr t1)

state ((t2+1)+1) = 3

First (t2,t2+1) (Falls a_read)
```

In state 3 the interface waits until the memory acknowledges the request to end the cycle by resetting the dtack signal. The derived behaviour of the state machine while waiting in state 3 shows that the a_read signal will remain low until the dtack becomes low (see Appendix B). This condition can be rewritten in the form required by the formal specification of the asynchronous memory.

```
StableUntil2 (a_read,Lo,t2+1,Falls dtack)
```

This condition ensures that the asynchronous memory will eventually signal the end of the read cycle by resetting dtack at some time t4.

```
First (t2+1,t4) (Falls dtack)
```

The derived behaviour of the state machine while waiting in state 3 also shows that the dataout signal remains stable. Therefore, at the end of this wait state, the value of dataout will still be the fetched memory word.

```
dataout ((t4+1)+1) = FETCH (mem t1) (addr t1)
```

The formal specification of the memory states that the internal state of the asynchronous memory will remain unchanged at the end of the read cycle and the memory will return to an idle state.

```
mem (t4+1) = mem t1

MemIdle (a_read,a_write,dtack,mem) (t4+1)
```

The last step in the symbolic simulation allows the interface to return to state 0, its idle state, at time (t4+1)+1. Because the signals a_read and a_write will remain low during (t4+1)+1, we can show that the state of the asynchronous memory remains unchanged and idle at time (t4+1)+1.

```
mem ((t4+1)+1) = mem t1

MemIdle (a_read,a_write,dtack,mem) ((t4+1)+1)
```

The idle signal generated by the state machine provides an output signal which indicates when the synchronous interface is between memory requests. This signal is only high in state 0 and remains low during read and write cycles. Hence, the next time that idle becomes high after t1 is time (t4+1)+1 when the state machine returns to state 0.

```
Next (t1,(t4+1)+1) (IsHi idle)
```

We summarize the results of symbolic simulation in the following theorem (also given in Section 10) which expresses relevant facts about the simulation at the end of read cycle in relation to its initial state at time t1. The existentially quantified variable t2 stands for when the state machine returns to state 0, *i.e.* time (t4+1)+1, and it should not be confused with time t2 in the above simulation.

```
⊢ ∀ read write dtack addr datain a_dataout
    idle a_read a_write a_addr a_datain dataout state mem.
    Interface (
      read,write,dtack,addr,datain,a_dataout,
      idle,a_read,a_write,a_addr,a_datain,dataout,state) ∧
    AsynMem (a_read,a_write,a_addr,a_datain,dtack,a_dataout,mem)
    ⟹
    ∀t1.
      (state t1 = 0) ∧
      IsHi read t1 ∧
      MemIdle (a_read,a_write,dtack,mem) t1
      ⟹
      ∃t2.
        Next (t1,t2) (IsHi idle) ∧
        MemIdle (a_read,a_write,dtack,mem) t2 ∧
        (dataout t2 = FETCH (mem t1) (addr t1)) ∧
        (mem t2 = mem t1)
```

Symbolic simulation is also used to derive a similar theorem summarizing the interaction of the synchronous interface with the asynchronous memory during a

write cycle. In this case, the contents of the dataout register remain unchanged but the memory has a new state defined in terms of the STORE operation.

```
⊢ ∀ read write dtack addr datain a_dataout
    idle a_read a_write a_addr a_datain dataout state mem.
    Interface (
      read,write,dtack,addr,datain,a_dataout,
      idle,a_read,a_write,a_addr,a_datain,dataout,state) ∧
    AsynMem (a_read,a_write,a_addr,a_datain,dtack,a_dataout,mem)
    ⟹
    ∀t1.
      (state t1 = 0) ∧
      IsLo read t1 ∧
      IsHi write t1 ∧
      MemIdle (a_read,a_write,dtack,mem) t1
      ⟹
      ∃t2.
        Next (t1,t2) (IsHi idle) ∧
        MemIdle (a_read,a_write,dtack,mem) t2 ∧
        (dataout t2 = dataout t1) ∧
        (mem t2 = STORE (mem t1) (addr t1) (datain t1))
```

Finally, we derive a theorem for an idle cycle, that is, a single step by the state machine beginning and ending in state 0 which leaves dataout and mem unchanged.

```
⊢ ∀ read write dtack addr datain a_dataout
    idle a_read a_write a_addr a_datain dataout state mem.
    Interface (
      read,write,dtack,addr,datain,a_dataout,
      idle,a_read,a_write,a_addr,a_datain,dataout,state) ∧
    AsynMem (a_read,a_write,a_addr,a_datain,dtack,a_dataout,mem)
    ⟹
    ∀t1.
      (state t1 = 0) ∧
      IsLo read t1 ∧
      IsLo write t1 ∧
      MemIdle (a_read,a_write,dtack,mem) t1
      ⟹
      ∃t2.
        Next (t1,t2) (IsHi idle) ∧
        MemIdle (a_read,a_write,dtack,mem) t2 ∧
        (dataout t2 = dataout t1) ∧
        (mem t2 = mem t1)
```