Axioms for Memory Access in Asynchronous Hardware Systems

J. MISRA
The University of Texas at Austin

The problem of concurrent accesses to registers by asynchronous components is considered. A set of axioms about the values in a register during concurrent accesses is proposed. It is shown that if these axioms are met by a register, then concurrent accesses to it may be viewed as nonconcurrent, thus making it possible to analyze asynchronous algorithms without elaborate timing analysis of operations. These axioms are shown, in a certain sense, to be the weakest. Motivation for this work came from analyzing low-level hardware components in a VLSI chip which concurrently accesses a flip-flop.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Design, Verification

Additional Key Words and Phrases: Concurrent access

1. INTRODUCTION

This paper is motivated by issues in hardware design. It addresses the problem of concurrent accesses to registers by asynchronous components. Any system in which concurrent accesses are permitted can be proven correct only if some assumptions are made about the behaviors of the registers, particularly under concurrent reads and writes. Unfortunately, physical behaviors of registers are so complex that a direct analysis of the physical behavior cannot be employed in any reasonable correctness argument. Therefore, we propose an axiomatic basis for the study of registers. We show that if certain axioms are obeyed by a register, then the device may be analyzed as a "serial device" (i.e., its operation is a sequence of actions). Hence any correctness issue involving the register is considerably simplified. We also show that our axioms are the weakest ones enjoying this property. Discussions with hardware designers lead us to believe that devices satisfying these axioms can be physically realized.

In analyzing concurrent systems [7], it has generally been assumed that memory references are nonconcurrent, that is, all accesses to a register are equivalent to those accesses made in some sequence. Thus, if two or more

This work was supported by the IBM Corporation.
Author's address: The University of Texas at Austin, Austin, TX 78712.

processes simultaneously attempt to access a register, then the accesses will be made in some arbitrary sequential order: the effect of concurrent accesses $op1 \parallel op2$ is equivalent to the sequence $op1$; $op2$ or $op2$; $op1$. It follows then that if two write operations are simultaneously attempted, then one of the values appears in the register upon completion of both operations. Also, if a read and a write overlap in time, then the read will either receive the value before the write or the value after the write.

Nonconcurrency of accesses to a register can be established by locking the register such that a write operation is never executed concurrently with any other operation. This paper is motivated by questions of asynchronous hardware design. At the lowest hardware level, a flip-flop is a register which is capable of storing one bit. Locking a flip-flop requires implementing a lock bit, which again has to be implemented as a flip-flop with similar kinds of constraints on accesses. Also, the overhead of locking and consequent loss of concurrency makes such an approach unattractive for very low-level hardware implementation. We consider a different approach: we propose to build the register such that, to an external observer, concurrent accesses appear nonconcurrent and nonconcurrent accesses appear to preserve their order.

It is by no means obvious what value gets deposited in a register if two writers are writing into it simultaneously (see [1, 5, 8, 10] for descriptions of some of the complexities associated with a flip-flop operation). We propose to study registers axiomatically. We postulate a set of axioms about the properties of a register which make intuitive sense, and then show that any register obeying these axioms can be viewed as one for which accesses are nonconcurrent.

Each horizontal line denotes a time interval in which an operation executes. Notation $w_1:0$, $w_3:0$ denote write operations that write a "0" into the register; analogously, $w_2:1$. Notation $r_2:0$ denotes a read operation that returns a value 0; analogously, $r_1:1$. Subscripts are used to distinguish among operations. In Figure 1, operations $w_2:1$, $r_1:1$ are *concurrent* because they overlap in time and so do $w_2:1$ and $w_3:0$. Figure 1 pictorially depicts certain concurrent accesses to a register.

If the register behaves in the manner depicted in Figure 1, its behavior is indistinguishable from a register where the operation takes place one-at-a-time and in the sequence shown in Figure 2.

In this case (Figure 1) processes that access the register can be analyzed as if concurrent memory references were sequentially ordered. Consider, however, the situation depicted in Figure 3, which is identical with Figure 1, except that the last read operation returns a value 1. We show that the operations in Figure 3 cannot be ordered so that nonconcurrent operations preserve their order and the reads receive consistent values.

Assume for this example (Figure 3) that a process P sequentially executes $w_1:0$, $r_1:1$, $w_3:0$ and $r_2:1$ and a process Q executes $w_2:1$. It is then impossible to analyze processes P, Q, assuming that concurrent accesses to the common register are sequentially ordered. Because, if concurrent accesses are sequentially ordered and process P is aware that Q writes a "1" into the register after its first write and before its last read, then process P can guarantee that one of its reads will return value "0", contradicting the actual operation as given in Figure 3. The

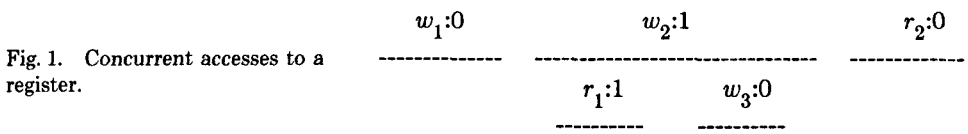

Fig. 1. Concurrent accesses to a register.

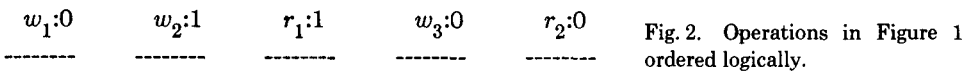

Fig. 2. Operations in Figure 1 ordered logically.

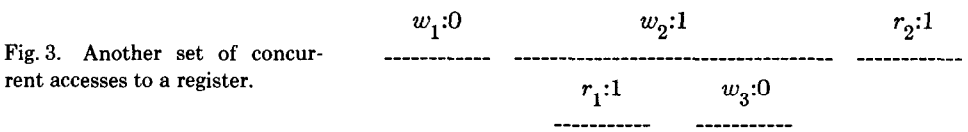

Fig. 3. Another set of concurrent accesses to a register.

question then is: what properties of the register will make it behave as in Figure 1 and not as in Figure 3.

We formally state the problem in the next section. We define *valid schedules*, a sequence of operations whose effect is equivalent to some sequential (nonconcurrent) executions of these operations. We suggest a set of axioms for a register in Section 3. We show, in Section 4, that these axioms define only valid schedules and, in a sense, are the weakest. We also give a simple, necessary, and sufficient condition for the validity of a schedule.

The example in Figure 3 is from Mills and Lounsbery [6]. They show that elaborate timing analysis is needed if a register is indeed to behave in this manner. Work reported here began after reading that paper, and as an attempt to derive conditions under which a register will not display such pathological behavior.

Lamport [2, 3] has considered the problem of concurrent reading and writing on a register which may hold more than one atomic data item; the possibility of an inconsistent read therefore exists. In [2, 3] he proposes protocols for reading and writing which guarantee consistency of every read. We do not consider the problem of receiving an inconsistent value in a read operation. In [4] he treats a problem similar to the one treated here for the special case of one writer (i.e., there are no overlapping write operations). It is then shown that a schedule is valid if and only if a write operation overlaps with at most one read operation. This result holds under very weak assumptions about the change in register value during a write.

A problem similar to the one addressed here appears in concurrent database access and is called the serializability problem [9]. In that problem, a number of transactions—analogous to processes—each having several atomic steps, concurrently read from and write into a database. It is assumed that truly concurrent steps, that is those happening at exactly the same time, are arbitrarily ordered,

and therefore the result of concurrent access is an interleaved sequence of steps accessing the database. It is required to develop conditions that guarantee that any interleaved sequence of transaction steps is equivalent to some serial executions of the transactions. Therefore each transaction may assume that it is executing alone, even if that is not the case, when these conditions are met. Our problem is different in several respects. Our notion of serial execution is much weaker: we merely observe the accesses made to a register, without knowing which process made an access. Our problem is to devise conditions under which concurrency is equivalent to interleaving; database concurrency problems assume this equivalence and ask when interleaving is equivalent to sequential execution.

2. BASIC CONCEPTS

An *operation* is either a *read* or a *write*. Associated with each operation is a *value*. For a read operation, the associated value represents the result of reading the register contents. For a write operation, the associated value is the value that is being written into the register. *We adopt the convention that all values written by write operations are distinct.* This convention may be enforced by encoding the actual value being written and the instance of the write operation together into a single value which is associated with the write operation. This requirement is only for convenience: we can now distinguish different write operations through their associated values and also we can associate a read operation with a specific write operation if their associated values are identical. It should be realized that, in practice, distinct values are not necessarily written by distinct write operations. Even then the axioms that we propose would be meaningful, and their implementation would lead to valid schedules only. However, our claim that the axioms are the weakest ones defining valid schedules would no longer be true.

Each operation, read or write, has an associated *start* and *end* event. We use $|op{:}x$ and $op{:}x|$ to denote start and end, respectively, of operation *op*, with associated value x. We omit x when this value is irrelevant to the discussion.

A *schedule* is a sequence of operation starts and ends such that for every $|op$ there is a unique $op|$ and vice versa.

For a given schedule, *precedes* denotes that some event (start or end of an operation) happens before another event. Thus $|op1{:}x$ *precedes* $|op2{:}y$ means that *op*1 starts before *op*2. We write *op*1 *precedes op*2 to denote that $op1|$ *precedes* $|op2$ (i.e., *op*1 ends before *op*2 starts). Operations *op*1, *op*2 are *concurrent* if neither *precedes* the other; they are *nonconcurrent* otherwise.

A *valid schedule* S is one for which it is possible to rearrange the operations so that all operations are nonconcurrent, all nonconcurrent operations in S preserve their orders, and the values received by the read operations are consistent with the write operations. Formally, S is a valid schedule iff there exists a permutation S' of S satisfying all of the following *validity conditions.*

Validity Conditions

VC1. For every *op*, $|$ *op precedes op* $|$ in S'.
VC2. Every pair of operations in S' is nonconcurrent.
VC3. If *op*1 *precedes op*2 in S, then *op*1 *precedes op*2 in S'.
VC4. In S', every read operation has a preceding write operation, and, if $w{:}x$ is the closest preceding write operation for $r{:}y$ in S', then $y = x$.

Figure 1 depicts a valid schedule because the permutation of it given by Figure 2 satisfies the four validity conditions. Figure 3 depicts an invalid schedule, that is, a schedule that is not valid; we prove this assertion in the next section.

If a set of processes access a register in such a manner that their access schedule S is valid, then we can analyze these processes using the familiar assumption that memory references are nonconcurrent. We can then imagine that the register is being accessed in a sequential manner as given by S'. Conversely, if an access schedule is invalid, then such an assumption cannot be made, in general, and therefore analysis of the corresponding concurrent program becomes far more formidable.

Valid schedules constitute a rich class. For instance, $|w:x\,|w:y\ w:x|\ |r:y\,|r:x\ w:y|\ r:y|\ r:x|$ is valid, even though the two read operations $r:x$ and $r:y$ may start and end nearly simultaneously and still return different values.

3. AXIOMS FOR REGISTER OPERATION

The following axioms state some facts about the value held by a register, how it is manipulated by writes, and how it affects the results of reads. We assume that a register has a unique value at some time instants and that its value is undefined at other time instants. The undefined value may correspond to the point where a component, such as a flip-flop, may be undergoing state transition.

We first present the axioms and then provide the rationale. In the following, *point* is a time instant. If $t1$, $t2$ are points, then $t1 < t2$ denotes that $t1$ is a time instant before $t2$. When there can be no confusion, we use $|op$ and $op|$ to denote the start and end times of operation op. Point t is *within* op means $|op \leq t \leq op|$. As usual, $r:x$, $w:x$ denote read operation and write operation with value x, respectively.

3.1 Register Axioms

A1. For every $r:x$ there is some point within r where the register value is x.

A2. If the register value is x at point t and a write operation, $w:y$, starts after t and ends before t', $t < t'$, then the register value at t' is different from x (it could be undefined).

A3. If register value is x at point t', then there exists t, $t \leq t'$ such that t is within $w:x$ and the register value at t is x.

A4. If the register value is x at t and t', then it is x at all points between t and t'.

3.2 Rationale for the Axioms

Axiom A1 says that is impossible to return a value x if the register never has the value x during a read. Note that the register may have several different values during a read, and the axiom does not specify which value will be returned by the read.

Axiom A2 says that a write operation will have some effect on the value of the register. Upon completion of a write operation, $w:y$, it may be asserted that the register no longer has its former value. Note, however, that we do not require the register value to be y upon completion of $w:y$. It is likely that the register never has value y if $w:y$ is concurrent with other write operations.

Axiom A3 says that any register value x at t' must be written by a write operation that starts at or before t' (i.e., the register cannot change its value spontaneously). In this case the write operation $w:x$ deposits the value x at some point within w, at or before t'.

Axiom A4 implies that $w:x$ writes the value only once and does not try to rewrite if the value changes, presumably as a result of another write. Therefore, once the value changes, the register never regains its former value. This requirement is realistic in the way hardware typically changes values in a register. Multiple writing of the same value by one write will only overwrite the value of a competing write; in a pathological situation, this could lead to nonterminating operations if, for instance, two writes alternatively overwrite each other's values.

Notation. We use (x) within a schedule to denote that the register value is x at that point and remains x until some other (y) appears in the schedule.

Example 1. It can be verified that the following schedule is valid and satisfies all the register axioms.

$$|w:x\ |w:y\ (y)\ w:x|\ |r:y\ w:y|\ r:y|$$

The only read operation returns a value of y. However, a modification of this schedule, in which there is an extra read operation that returns value x, is invalid; axiom (A4) is being violated in this case:

$$|w:x\ |w:y\ w:x|\ |r:y\ w:y|\ |r:x\ r:y|\ r:x|$$

3.3 Notes on the Axioms

(1) It follows from A1 and A3 that a read operation receives value x only if there is a $w:x$ in the schedule.

(2) It is possible to combine A3 and A4: if the register value is x at t', then there exists $t \leq t'$ such that t is within $w:x$ and the register value is x at all t'', $t \leq t'' \leq t'$. We have kept the two axioms separate to emphasize the distinct assumptions embodied in each one.

(3) Axiom A4 may be weakened to read: if register value is x at t and t', then, for all t'', $t \leq t'' \leq t'$, the value is either x or undefined. All the theorems in this paper are satisfied when this weak version is substituted for A4. It should be noted, however, that this weakening is without much merit; under the assumption that every read returns some value as its result, no set of experiments can establish whether a register obeys A4 or this weak version of A4.

4. PROPERTIES OF VALID SCHEDULES

We now show that all schedules on a register satisfying these axioms are valid. We also show that these axioms are, in a sense, the weakest axioms guaranteeing validity. We first give a necessary and sufficient condition for the validity of a schedule.

Given any schedule S, define a relation *before* on the values associated with the operations, as follows:

(1) x *before* x, if some $r:x$ *precedes* $w:x$ or there is no $w:x$.
(2) x *before* y, $y \neq x$, if $op1:x$ *precedes* $op2:y$ for some $op1$, $op2$.
(3) x *before* z, if for some y, x *before* y and y *before* z.

Example 2. Consider the schedule:

$$|w:y\ |w:x\ \ w:y|\ |w:z\ \ w:x|\ |r:y\ \ w:z|\ r:y|$$

Note that x *before* y, because $w:x$ *precedes* $r:y$ and y *before* z because $w:y$ *precedes* $w:z$. Furthermore, x *before* z because x *before* y and y *before* z. Note that *before* is irreflexive for this schedule.

Define the relation *before* to be a *partial order* if it is *irreflexive* (*not* x *before* x, for all x), *antisymmetric* (x *before* y implies *not* y *before* x, for all x, y), and *transitive* (x *before* y and y *before* z implies x *before* z, for all x, y, z). Note that usual definitions of partial order require the relation to be *reflexive*. Our definition is more useful in our context because we expect no read operation to precede the corresponding write operation. Observe also that relation *before* is transitive from definition, and hence it suffices to show that it is irreflexive and antisymmetric to prove that it is a partial order.

THEOREM 1. *A schedule is valid if and only if relation* before, *corresponding to this schedule, is a partial order.*

PROOF. Let *before* be a partial order for some schedule S. Then, from irreflexivity of *before*, for every read operation $r:x$, there is a $w:x$ in the schedule. For each value x, construct the subsequence: $|w:x\ \ w:x|\ |r_1:x\ \ r_1:x|\ |r_2:x\ \ r_2:x|\ \ldots$, where the write operation $w:x$ appears first, followed by all the read operations, if any, whose associated value is x; if $r_j:x$ precedes $r_k:x$ in S, then r_j appears prior to r_k in this subsequence.

Let $x_0, x_1 \ldots$ be a total order of all values in S consistent with the partial order *before*; such a total order can be obtained by a topological sort of the values with respect to *before*. Now construct the desired permutation S' from S by appending the subsequences, corresponding to $x_0, x_1, \ldots$. We claim that S' is a permutation of S which satisfies the validity conditions. Validity condition VC1 is trivially satisfied. Every pair of operations in S' is nonconcurrent, by construction. If $op1:x$ *precedes* $op2:x$ in S, then, from the irreflexivity of *before*, $op2$ is a read, and hence the relative order of $op1$, $op2$ is preserved by the construction. If $op1:x$ precedes $op2:y$, $x \neq y$, in S, then x *before* y holds. The subsequence corresponding to x appears prior to that of y, and hence the order is preserved. Thus VC3 is satisfied. Every read receives the value of the closest preceding write, by construction, and therefore VC4 is satisfied.

Conversely, we show that if S is a valid schedule, then *before* is a partial order. Since S is valid, there exists a permutation S' satisfying the validity conditions. In S', all operations with associated value x must appear contiguously and $w:x$ *precedes* all $r:x$; otherwise some read will not receive the value of the closest preceding write. Therefore, S' defines a total order on the values. This total order must be consistent with the relation *before*. Hence *before* must be a partial order, otherwise no total order would be consistent with it. Irreflexivity of *before* follows trivially. □

Example 3. Schedule S of Example 2 has the following associated S', constructed by the procedure given in Theorem 1. The only total order consistent

with *before* is

$$xyz.$$

From this, we construct S':

$$|w{:}x\ w{:}x|\ |w{:}y\ w{:}y|\ |r{:}y\ r{:}y|\ |w{:}z\ w{:}z|$$

Example 4. Consider the following schedule, which is the one given pictorially in Figure 3.

$$|w{:}x\ w{:}x|\ |w{:}y\ |r{:}y\ r{:}y|\ |w{:}z\ w{:}z|\ w{:}y|\ |r{:}y\ r{:}y|$$

Relation *before* is not a partial order because y before z and z before y. Therefore, this schedule is not valid.

For any schedule S satisfying the axioms, define for any value x,

$$R_x = \{t \mid \text{value of register at point } t \text{ is } x\}.$$

Observation

(1) R_x is an *interval*, that is, if t_1, t_2 are in R_x, then any t, $t_1 < t < t_2$, is also in R_x.
(2) If R_x is nonempty, then for every $op{:}x$ in S there is some point within op which is in R_x.

Proof of Observation. Proof of (1) follows from axiom A4. Proof of (2): if op is a read, then the result follows from axiom A1; if op is a write, pick any point t from R_x and apply axiom A3.

THEOREM 2. *Any schedule satisfying register axioms* A1, A2, A3, *and* A4 *is valid.*

PROOF. We show that *before* is an irreflexive and antisymmetric relation for such a schedule. Since *before* is transitive, using Theorem 1, the result follows.

In S, for each $r{:}x$, there is a $w{:}x$ and $w{:}x$ does not precede $r{:}x$. This is because, from axiom A1, there is some point t' within a read where the register value is x and, from A3, there is some point t within $w{:}x$ such that $t \leq t'$. Therefore, the first rule in the definition of *before* does not apply to any value x.

We now show that *before* is antisymmetric. Consider any two values, x, y, $x \neq y$, for which R_x is nonempty and R_y is nonempty. For $op1{:}x$, $op2{:}y$, some point t_1 within $op1$ is in R_x and some point t_2 within $op2$ is in R_y, from the observation. If $op1$ *precedes* $op2$, then $t_1 < t_2$. Since R_x, R_y are intervals, then every point in R_x is prior to every point in R_y. Therefore, if $x \neq y$, R_x, and R_y are both nonempty and x *before* y, then every point in R_x is prior to every point in R_y; hence *before* is antisymmetric with respect to x, y.

Next, consider some value y where R_y is empty. Then there can be no $r{:}y$ in the schedule because, from axiom A1, register value is y at some point within $r{:}y$, which implies R_y is not empty. Therefore, there can only be a single operation with y, $w{:}y$, in S. Suppose x *before* y and y *before* x. From definition of *before* and the fact that only a single operation $w{:}y$ with value y exists in the schedule, it follows that there are two distinct operations $op1{:}x$, $op2{:}x$ such that $op1{:}x$

precedes $w{:}y$ and $w{:}y$ *precedes* $op2{:}x$. Since there are two distinct operations with value x, there is at least one read operation with value x, and hence R_x is nonempty. From the previous observation, some t_1 within op_1 and some t_2 within op_2 are both in R_x. Since $t_1 < |w{:}y$ and $w{:}y| < t_2$, $w{:}y$ is completely within R_x. This violates axiom A2.

Therefore, *before* is antisymmetric. Also, *before* is irreflexive, from the arguments in the first paragraph and antisymmetry. □

Theorem 2 shows that, in any schedule, if the register values satisfy A1 through A4, then the schedule is valid. We prove the converse in Theorem 3: for every valid schedule it is possible to specify register values at certain points such that axioms A1 through A4 are satisfied. While it is difficult to define the notion of weakest axioms precisely, we can reasonably claim—based on Theorems 2 and 3—that our proposed axioms are the weakest. Theorem 4 shows that the axioms are all independent (i.e., no axiom is implied by the other three).

THEOREM 3. *It is possible to assign register values at certain points in a valid schedule such that axioms* A1 *through* A4 *are satisfied.*

PROOF. For any valid schedule S, there is a permutation S' satisfying the validity conditions. We use S' to assign register values at certain points in S. Let $op_1, op_2, \ldots, op_n$ be the sequence of operations in S' and let v_i be the value associated with op_i. Note that v_i's are not necessarily distinct; however $v_i \neq v_j$, where $i < j$, means that $v_i \neq v_k$, for any k, $k \geq j$. As before, the notation (v_i) within a schedule means that the register has value v_i at that point and continues to have the value v_i until there is some (v_j), $v_j \neq v_i$, in the schedule. We use the following algorithm to assign (v_i), $1 \leq i \leq n$, to some points in the schedule S.

(1) (v_1) is assigned immediately following $|op1$ in S.
(2) (v_i), $1 < i$, is assigned immediately following $|op_i$ or $(v_{(i-1)})$, whichever comes later in S.

Note that $(v_1), (v_2), \ldots$ appear in this order in the schedule.

We first show that (v_i) is between $|op_i$ and $op_i|$ for all i in S, according to our construction. We always assign (v_i) after $|op_i$, and hence we only need to show that (v_i) appears before $op_i|$. We will show, using induction on i, that (v_i) appears before $op_j|$ in S, $j \geq i$. Note that $|op_i$ *precedes* $op_j|$ in S if $j \geq i$, because otherwise S' does not preserve order among op_i, op_j. Now (v_1) appears before $op_j|$, $j \geq 1$, by construction and the fact that $|op_1$ *precedes* $op_j|$. Next, (v_i), $i > 1$, either comes immediately after $(v_{(i-1)})$ and since, according to induction, $(v_{(i-1)})$ appears before $op_j|$, $j \geq i - 1$, the result holds—or (v_i) comes immediately after $|op_i$ and $|op_i$ *precedes* $op_j|$, $j \geq i$.

Axiom A1 is satisfied because within each operation, and hence within each read operation, the register has the associated value. Axiom A2 can be shown to be satisfied as follows. For a write operation op_j, let register value be v_i prior to the write, and hence $v_i \neq v_j$. Register value is v_j at some point during the write (from the construction and arguments of the previous paragraph). Register value at any point after the write is v_k, for some k, $k \geq j$, and since $v_i \neq v_j$, it follows that $v_i \neq v_k$.

Axiom A3 holds because in S', if $op_i{:}x$ is a write and $op_j{:}x$ is a read, then $i < j$; according to our construction (v_i) appears before (v_j) and (v_i) appears within op_i. Axiom A4 holds trivially. □

COROLLARY. *For any schedule there exist register values satisfying axioms* A1 *through* A4 *if and only if the schedule is valid.*

Example 5. We show an example of assignment described in Theorem 3.

Let, S: $|w{:}y\ \ |w{:}x\ \ |r{:}y\ \ w{:}x|\ \ r{:}y|\ \ w{:}y|$
Let, S': $|w{:}x\ \ w{:}x|\ \ |w{:}y\ \ w{:}y|\ \ |r{:}y\ \ r{:}y|$
v_1 is x, v_2 is y, v_3 is y.

Assignment to schedule S leads to

$$|w{:}y\ \ |w{:}x\ \ (x)\ \ (y)\ \ |r{:}y\ \ (y)\ \ w{:}x|\ \ r{:}y|\ \ w{:}y|$$

We now show that the axioms are independent: if we drop any one of the axioms, we can find an invalid schedule with a sequence of register values which satisfy the remaining axioms.

THEOREM 4. *Any three axioms (from* A1 *to* A4*) are satisfied by some assignment of register values to some invalid schedule.*

PROOF. For each combination of three axioms we show an invalid schedule with register values satisfying these axioms.

A1, A2, A3: $|w{:}y\ |w{:}x\ |r{:}x\ (x)\ r{:}x|\ (y)\ w{:}y|\ |r{:}y\ \ r{:}y|\ |r{:}x\ (x)\ w{:}x|\ r{:}x|$

This schedule is invalid because, from the read operations alone, it follows that x *before* y and y *before* x.

A1, A2, A4: $|r{:}x\ (x)\ r{:}x|\ \ |w{:}x\ \ w{:}x|$

This schedule is invalid because a read *precedes* the corresponding write, and hence x *before* x.

A1, A3, A4: $|w{:}x\ (x)\ |r{:}x\ \ r{:}x|\ \ |w{:}y\ \ w{:}y|\ \ w{:}x|\ \ |r{:}x\ \ r{:}x|$

This schedule is invalid because x *before* y and y *before* x.

A2, A3, A4: $|w{:}x\ (x)\ w{:}x|\ \ |w{:}y\ (y)\ w{:}y|\ \ |r{:}x\ \ r{:}x|$

This schedule is invalid because x *before* y and y *before* x. □

We now derive certain properties of schedules satisfying the register axioms A1 through A4.

PROPERTY 1. *If the register value is* x *at all points during a read, then the read returns a value* x.

PROOF. Using A1, no other value can be returned. □

PROPERTY 2. *A read operation* r *receives the value of a preceding write operation* w *if no other write operation executes in the interval* $|w\ r|$.

PROOF. If the write and read have values x, y respectively and $x \neq y$, then, from A1 and A3, register value is y at some point before $|w:x$ and also at some point after $w:x|$. This contradicts A2. □

It follows that a write operation is successful in depositing its value in the register provided it is nonconcurrent with every other write.

PROPERTY 3. *The register value persists in the absence of writes (i.e., if register value is different at points t and t', $t < t'$, then there is a t'', $t \leq t'' \leq t'$, which is within some write operation.*

PROOF. By applying axioms A2 and A3. □

PROPERTY 4. *Let $r:x_1$, $r:x_2$, ... be a set of concurrent read operations. Suppose no write operation is concurrent with any of these read operations. Then, $x_1 = x_2 = \ldots$.*

PROOF. From Property 3, the register value will remain unchanged during all the read operations. The result then follows from Property 1. □

5. SUMMARY AND CONCLUSION

There has been a considerable amount of research on asynchronous concurrent programming in the last 15 years. This paper is an effort to study the applicability of this research in hardware design. We have studied a fundamental assumption—nonconcurrency of simultaneous access—which is basic to most concurrent programming work. We have shown how this assumption can be met by designing registers that meet certain axioms. The axioms seem quite basic and are usually met in practice, at least for flip-flops. The most nonintuitive is axiom A4, which says that the implementation of a write operation must attempt to write its value only once. We have shown that these axioms are sufficient and, in a sense, necessary.

ACKNOWLEDGMENTS

Dr. Harlan D. Mills, of IBM, has pointed out the necessity of elaborate timing analysis if the nonconcurrency assumption is not met [6]. I am indebted to him for suggesting the problem and for his enthusiasm, encouragement, and advice. I am grateful to other members of the Provable Hardware Design Group (Jim Aylor, Ray Hookway, Norm Pleszkoch, John Saunders) of IBM, Federal System Division, for interaction and constructive criticism during the course of this work. Professor Doug Jensen has pointed out that the proposed axioms may be viewed as design rules for constructions of registers. Dr. Leslie Lamport has kindly brought some of his early unpublished work to my attention. Detailed comments from Professors Charles Seitz and Alain Martin, and an anonymous referee are deeply appreciated.

REFERENCES

1. CHANEY, T., AND MOLNAR, C. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. Comput.* (Apr. 1973), 421–422.
2. LAMPORT, L. Concurrent reading and writing. *Commun. ACM 20*, 11 (Nov. 1977), 806–811.

3. LAMPORT, L. A new approach to proving the correctness of multiprocess program. *ACM Trans. Program. Lang. Syst. 1*, 1 (July 1979).
4. LAMPORT, L. A theorem on multiprocess algorithms. Tech. Rep. CA-7503-2011, Massachusetts Computer Associates, Mar. 1975.
5. MARINO, L. P. General theory of metastable operation. *IEEE Trans. Comput. C-30*, 2 (Feb. 1981).
6. MILLS, H. D., AND LOUNSBERY, J. M. Combinatorial analyses of process synchronization. IBM, FSD (internal memo), Aug. 1983.
7. OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs. *Acta Inf. 6*, 4 (1976), 319–340.
8. SAUNDERS, J. M. Engineering description of a flip-flop reader and writer. IBM, FSD (internal memo), Jan. 1984.
9. ULLMAN, J. D. *Principles of Database Systems.* Computer Science Press, Rockville, Md., 1980.
10. WANN, D., AND FRANKLIN, M. Asynchronous and clocked control structures for VLSI—based interconnection networks. *IEEE Trans. Comput. C-32*, 3 (Mar. 1983), 264–293.

Received June 1984; revised June 1985; accepted July 1985